October 1991

Jack 5 User's Guide

Cary B. Phillips  
University of Pennsylvania

Follow this and additional works at: https://repository.upenn.edu/cis_reports

Recommended Citation


This paper is posted at ScholarlyCommons. https://repository.upenn.edu/cis_reports/479  
For more information, please contact repository@pobox.upenn.edu.
Jack 5 User's Guide

Abstract
This chapter is a brief tutorial introduction to using Jack. It demonstrates some of the basic Jack features, illustrating mostly the flavor of how you interact with Jack rather than describing the details of how to use it. The first time you run Jack, you should go through the examples in this chapter. From then on, refer to the later chapters for more details about the various features.

Comments
Jack 5 User’s Guide

MS-CIS-91-78
GRAPHICS LAB 43

Cary B. Phillips

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104-6389

October 1991
Contents

1 An Introduction to Jack

1.1 What is Jack? ............................... 15
  1.1.1 Jack Version 5.8 ..................... 15
  1.1.2 New Features of Jack Version 5.8 from Version 5.5 ...... 15
  1.1.3 Jack Version 5.x (up to Version 5.5) ................. 17
  1.1.4 New Features of Jack Version 5.x from Version 4.x ...... 18

1.2 How to Use This Manual ................. 19
1.3 Technical Support ...................... 19
1.4 DemoJack ................................ 19
1.5 Preliminaries ......................... 19
  1.5.1 Jack and the Window Manager ....... 19
  1.5.2 Using the Mouse .................... 20
1.6 Running Jack ............................ 20
1.7 Reading in an Object .................. 22
1.8 Creating a Human Figure ............... 23
1.9 Changing the View ..................... 25
1.10 Moving a Figure ....................... 26
1.11 Writing the Environment to a File ...... 28
1.12 Getting Help ........................... 29
1.13 Browsing through Jack Commands ...... 30
1.14 Exiting Jack ........................... 30

2 Using the Keyboard ...................... 31
  2.1 The Edit Buffer ....................... 31
    2.1.1 Default Values .................... 32
  2.2 Automatic Completion ................. 32
  2.3 Using the Mouse as an ENTER key .... 32
  2.4 Entering Commands from the Keyboard ...... 33
  2.5 Entering Files from Pop-up Menus ...... 36

3 Peabody Objects ......................... 37
  3.1 Peabody Terminology and Concepts ...... 37
    3.1.1 Peabody Names .................... 37
  3.2 Picking Objects with the Mouse ...... 38
  3.3 Picking Overlapping Objects .......... 39
  3.4 Picking Sites and Joints ............... 40
  3.5 Picking Objects by Name ............... 41
    3.5.1 Picking Objects through Pop-up Menus .... 41
  3.6 Picking the Previous Thing .......... 41
4 Changing the View
4.1 Changing the View Reference Point ........................................ 43
4.2 Snapping the View ............................................................. 45
4.3 The Field of View .................................................................. 46
   4.3.1 A Warning About the Field of View ................................... 46
4.4 Attaching the View to Objects ................................................. 47
4.5 Adjusting the Clipping Planes .................................................. 48
   4.5.1 User Defined Clipping Planes ............................................. 48
4.6 View Stack ............................................................................ 49

5 Moving Figures
5.1 The Movement Operator ......................................................... 51
5.2 Translation ............................................................................ 52
5.3 Rotation ................................................................................. 53
5.4 Terminating the Movement ...................................................... 54
5.5 Changing the Reference of Movement ...................................... 54
5.6 Repositioning the View .......................................................... 55
5.7 Hints ...................................................................................... 55
5.8 Entering Transformations from the Keyboard ......................... 56
   5.8.1 Multiplying Homogeneous Transformations ....................... 56
5.9 Rotating with the Arrow Keys ................................................ 58
5.10 Snapping to Objects ............................................................. 58
5.11 Adjusting Joints .................................................................... 61
   5.11.1 The Human Shoulder ..................................................... 61
   5.11.2 Manipulating Joints Interactively ..................................... 62
   5.11.3 Joints with No Degrees of Freedom ............................... 63
5.12 The Directionality of Joints .................................................... 64
5.13 Entering Joint Angles From the Keyboard .............................. 64

6 Manipulating Human Figures
6.1 Introduction ........................................................................... 65
   6.1.1 Moving Human Figures ................................................... 66
6.2 Bending the Torso .................................................................. 67
   6.2.1 Moving the Torso More Generally .................................... 68
   6.2.2 Torso Behavior ............................................................... 69
6.3 Rotating the Pelvis ................................................................. 70
6.4 Moving the Center of Mass ..................................................... 72
   6.4.1 Controlling the Balance ................................................... 73
   6.4.2 Seated Figures ................................................................. 73
   6.4.3 The Figure Support ........................................................ 76
6.5 Moving the Feet ...................................................................... 77
   6.5.1 The move foot Command ................................................. 77
   6.5.2 Stepping Behaviors ......................................................... 79
   6.5.3 Controlling the Feet ........................................................ 80
6.6 Moving the Arms .................................................................... 80
   6.6.1 The New Shoulder .......................................................... 80
   6.6.2 The move arm Command ................................................ 81
   6.6.3 Controlling the Hands ...................................................... 82
   6.6.4 Attaching Objects to the Hands ....................................... 83
   6.6.5 Reaching with Attached Objects ....................................... 84
   6.6.6 Reaching in General ........................................................ 84
   6.6.7 Articulated Hands with Fingers ....................................... 85
6.7 Moving the Head and Eyes ...................................................... 86
   6.7.1 Fixating the Head ............................................................ 87
### CONTENTS

6.8 Postures .......................................................... 87

7 Specifying and Analyzing Human Figures ................................. 89
   7.1 Specifying Anthropometry ........................................ 89
      7.1.1 Foot Dimensions ........................................ 89
      7.1.2 Leg Dimensions ........................................ 89
      7.1.3 Pelvic Dimensions ...................................... 90
      7.1.4 Torso Dimensions ...................................... 90
      7.1.5 Arm Dimensions ....................................... 90
      7.1.6 Hand Dimensions ..................................... 90
      7.1.7 Head Dimensions ..................................... 90
      7.1.8 Neck Dimensions ..................................... 90
      7.1.9 Eye Coordinates ....................................... 90
   7.2 Measuring Dimensions .......................................... 91
      7.2.1 Rulers .................................................... 92
   7.3 Human Body Measurements ....................................... 92
   7.4 Managing Human Figure Files ................................... 94
      7.4.1 Human Figure Psurf Files ................................ 94
   7.5 Biostereometric Contour Body .................................. 95
   7.6 Upgrading Human Figures ........................................ 95
   7.7 Viewing Analysis .............................................. 95
      7.7.1 Eye View Windows ..................................... 95
      7.7.2 View Cones ........................................... 96
   7.8 Torque and Strength Display .................................... 96

8 The Jack Motion System .................................................. 101
   8.1 The Animation Window ......................................... 102
      8.1.1 Time ...................................................... 103
   8.2 Motion Parameters ............................................. 105
      8.2.1 Time Intervals ......................................... 105
      8.2.2 Weight Functions ...................................... 105
      8.2.3 Velocity Functions ..................................... 105
   8.3 The Motion of Human Figures ................................... 106
      8.3.1 Controlling Behaviors Over Time .......................... 106
      8.3.2 The Center of Mass ..................................... 107
      8.3.3 The Pelvis .............................................. 107
      8.3.4 The Torso ............................................... 107
      8.3.5 The Feet ................................................. 107
      8.3.6 The Arms ................................................. 108
   8.4 Other Types of Motion .......................................... 108
      8.4.1 Joint Motions ........................................... 109
      8.4.2 Figure Motions ......................................... 110
      8.4.3 Path Motions ........................................... 110
      8.4.4 Camera Motion .......................................... 110
      8.4.5 Timed Events ............................................ 110
   8.5 Motion Groups .................................................. 111
   8.6 Time ............................................................... 111
      8.6.1 The Starting Frame ..................................... 111
      8.6.2 Moving Time in the Animation Window .................. 112
   8.7 Changing Motions ................................................ 112
   8.8 Saving Motions ................................................ 113
   8.9 Playing, Saving, and Recording Frames ............................ 113
   8.10 Channels .......................................................... 114
   8.11 (Near) Real-Time Animation Preview .............................. 114
# Contents

8.12 Recording Notes ........................................... 115
8.12.1 Field recording ........................................ 115
8.12.2 Full Scene Anti-Aliasing: ................................ 115

9 Reading Files and Creating Objects 117
9.1 Reading Files ........................................ 117
9.2 Binary Psurfs ............................................ 118
9.2.1 Generating Binary Psurfs ................................ 118
9.2.2 Compressed Files ........................................ 118
9.3 Keeping Track of Jack Files ................................ 118
9.3.1 The Installation File .................................... 119
9.3.2 Installing Files Interactively ......................... 119
9.3.3 The Default "/jack5.install" File ..................... 119
9.3.4 How Jack Reads Peabody Files ....................... 120
9.3.5 How Jack Finds Psurf Files ............................ 120
9.4 The Uses of Peabody and Psurf Files .................... 120
9.4.1 Uses of Psurf Files ..................................... 121
9.4.2 Uses of Figure Files .................................... 121
9.4.3 Uses of Environment Files .............................. 121
9.5 How To Organize Your Data Files ........................ 121
9.6 Getting Information About Files .......................... 122
9.7 Creating Sites ............................................. 122
9.8 Creating Light Sources .................................... 122
9.9 A Note About Surface Attributes .......................... 122

10 Constraints 123
10.1 Introduction ............................................. 123
10.2 Constraints and Motion .................................... 124
10.3 The Definition of a Constraint and its Parameters .... 124
10.4 How Jack Solves Constraints ............................. 129
10.4.1 "Why is the Elbow Over There?" ..................... 130
10.4.2 "Why Isn't That Constraint Satisfied?" ............. 130
10.4.3 Constraint Evaluation Parameters .................... 131
10.5 Controlling Constraints ................................... 132
10.6 Changing Constraints ..................................... 132
10.7 Exploring Relationship Types ............................ 133
10.8 Hold Goals ............................................... 136
10.9 Face Goals ............................................... 137
10.10 Interactive Reaching ...................................... 137
10.11 Rooting Constraints ...................................... 139

11 Changing Objects 141
11.1 Re-rooting Figures ......................................... 141
11.2 Scaling Figures ........................................... 143
11.3 Controlling Degrees of Freedom ........................ 145
11.4 Renaming Objects ......................................... 145
11.5 Deleting Figures .......................................... 145
11.6 Creating Articulated Figures ............................ 145
11.6.1 Creating Joints ...................................... 145
11.6.2 Merging Segments .................................... 147
11.6.3 Deleting Joints and Segments ....................... 147
11.7 Joint Motors ............................................. 148
11.8 Segment Physical Properties .................. 148
# CONTENTS

12 Surface Attributes and Light Sources 151

12.1 Attribute Names .......................................................... 151
12.2 The Jack Color Palette .................................................. 151
12.3 Picking Attributes Interactively ...................................... 152
12.4 The IRIS Lighting Model ................................................ 152
12.5 Attribute Parameters ...................................................... 153
  12.5.1 Ambient and Diffuse Parameters ................................. 153
  12.5.2 Specular Highlights ................................................ 154
  12.5.3 Making Objects Appear Smooth ................................... 155
12.6 Associating Attributes with Surfaces ................................ 156
  12.6.1 Segments with Several Attributes ................................ 156
12.7 Using Light Sources ...................................................... 156
  12.7.1 Light Source Parameters .......................................... 160
12.8 Real-Time Texture Mapping ............................................ 161
  12.8.1 Reflection Mapping ................................................ 164
  12.8.2 Image Formats and Textures ...................................... 164

13 Manipulating Geometric Shapes 169

13.1 Modifying Psurf Geometry ............................................. 169
  13.1.1 Moving Nodes, Edges, and Faces ................................ 169
  13.1.2 Splitting Edges and Faces ....................................... 169
  13.1.3 Creating Nodes and Faces ........................................ 170
  13.1.4 Extruding Nodes, Edges, and Faces ............................... 170
  13.1.5 Deleting Parts of Psurfs .......................................... 172
  13.1.6 Fixing the Orientation of Psurf Faces ......................... 172
13.2 Free-Form Deformations ............................................... 174
  13.2.1 Controlling Deformations ....................................... 174
13.3 CAD Objects .............................................................. 176
  13.3.1 Creating and Manipulating CAD Objects ....................... 176

14 Writing Objects to Files 177

14.1 Writing Environment Files ............................................ 177
  14.1.1 Writing Positions ................................................ 178
14.2 Writing Psurfs ............................................................. 179

15 Windows and Display Parameters 181

15.1 Window Display Parameters ............................................ 181
15.2 Object Display Parameters ............................................ 182
  15.2.1 Transparent Objects .............................................. 185
  15.2.2 Displaying Parts of Psurfs ....................................... 185
  15.2.3 Displaying Sites .................................................. 186
  15.2.4 Turning Objects Off .............................................. 186
15.3 Shading Windows .......................................................... 187
15.4 Generating an Image file of a Window ............................... 187
15.5 Creating New Windows .................................................. 187
15.6 2D Windows ............................................................... 188
15.7 Windows and Cameras ................................................... 189
15.8 Changing Window Parameters .......................................... 189
  15.8.1 Frozen Windows .................................................... 189
  15.8.2 Setting Window Location ......................................... 190
15.9 Changing the Appearance of the Background ......................... 190
## 16 Customizing Jack

16.1 The "/.jack.jcl" Initialization File .......................... 193
16.1.1 Figuring Out the JCL Syntax .......................... 193
16.2 Binding Commands to Keystrokes .......................... 193
16.3 Setting Colors ............................................ 194
16.4 Setting Jack Parameters .................................... 195
   16.4.1 The Scale of Jack Windows ......................... 196
   16.4.2 Setting Units ........................................ 196
   16.4.3 Shading Parameters .................................. 197

## 17 Utilities

17.1 Collision Detection ........................................ 199
   17.1.1 Command Interface to Collision Detection ................. 199
17.2 Tracing Sites and Segments .............................. 202
17.3 Ghost Figures ............................................ 202
17.4 Recording Images to the Video Disk ....................... 202
17.5 Disabling Graphics ....................................... 203
17.6 Remote Commands and the Command Port .................... 203
   17.6.1 Network and Remote commands for Command Ports ............ 204
   17.6.2 Remote Commands for Collision Detection ................. 204
17.7 Paths ..................................................... 205
17.8 Prototype and Contributed Software - The contrib menu .... 205
17.9 Fractals .................................................. 205
   17.9.1 Type of Fractals Implemented ........................ 206
   17.9.2 Fractal Generation and Post-processing Parameters ...... 206
   17.9.3 The Interface ......................................... 206
   17.9.4 Reading and Writing Fractals .......................... 207
17.10 Network Jack ............................................ 207
17.11 Flock of Birds Interface .................................. 208
   17.11.1 Hardware/Operating System Information .................... 209
   17.11.2 Using the FOB in Jack ............................... 210
   17.11.3 Problems .......................................... 210
17.12 Simple Psurf Editor ...................................... 210
17.13 Collision Avoidance Behaviors .......................... 210
   17.13.1 Interactive Manipulation and Motion ....................... 210
   17.13.2 Commands ........................................... 210
17.14 Retina Windows .......................................... 212
17.15 Interactive Mirrors and Shadows .......................... 212

## A The Peabody Object Representation

A.1 Introduction to Peabody .................................... 213
   A.1.1 The Connectivity of Peabody Objects ....................... 213
   A.1.2 The Geometry of Peabody Objects ........................ 214
A.2 The Mechanics of the Peabody Language ...................... 216
   A.2.1 Arithmetic Expressions .................................. 216
   A.2.2 Units .................................................. 217
   A.2.3 Homogeneous Transformations ................................ 217
   A.2.4 Names and Identifiers ................................... 218
A.3 Peabody Construct Declarations ............................. 218
   A.3.1 Figure Declarations ..................................... 219
   A.3.2 Segment Declarations .................................... 220
   A.3.3 Path Declarations ....................................... 220
   A.3.4 Site Declarations ....................................... 220
   A.3.5 Joint Declarations ....................................... 221
CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.3.6 Constraint Declarations</td>
<td>222</td>
</tr>
<tr>
<td>A.3.7 Surface Attributes</td>
<td>222</td>
</tr>
<tr>
<td>A.3.8 Light Source Declarations</td>
<td>223</td>
</tr>
<tr>
<td>A.3.9 Peabody Block Structure</td>
<td>223</td>
</tr>
<tr>
<td>A.3.10 Postures and Posture References</td>
<td>225</td>
</tr>
<tr>
<td>A.3.11 Motion and Motion Group Declarations</td>
<td>226</td>
</tr>
<tr>
<td>A.3.12 The Include Statement</td>
<td>228</td>
</tr>
<tr>
<td>A.4 The Syntax of Psurf Files</td>
<td>228</td>
</tr>
<tr>
<td>A.4.1 Surface Attributes</td>
<td>229</td>
</tr>
<tr>
<td>A.4.2 Face Smoothness</td>
<td>229</td>
</tr>
<tr>
<td>A.4.3 Node Coloring</td>
<td>229</td>
</tr>
<tr>
<td>A.4.4 Big Psurfs</td>
<td>230</td>
</tr>
<tr>
<td>A.4.5 Face Orientation</td>
<td>230</td>
</tr>
<tr>
<td>A.4.6 Binary Psurfs and BPS</td>
<td>230</td>
</tr>
<tr>
<td>B Ray Tracing</td>
<td>231</td>
</tr>
<tr>
<td>B.1 Introduction</td>
<td>231</td>
</tr>
<tr>
<td>B.1.1 Hardware Shading vs. Raytracing</td>
<td>231</td>
</tr>
<tr>
<td>B.1.2 Displaying Images</td>
<td>231</td>
</tr>
<tr>
<td>B.1.3 Designing an Environment in Jack</td>
<td>232</td>
</tr>
<tr>
<td>B.1.4 Lights, Camera, Action</td>
<td>232</td>
</tr>
<tr>
<td>B.2 Raytracing</td>
<td>232</td>
</tr>
<tr>
<td>B.2.1 Parsepea</td>
<td>233</td>
</tr>
<tr>
<td>B.2.2 Rtrace</td>
<td>233</td>
</tr>
<tr>
<td>B.2.3 What Are My Options?</td>
<td>234</td>
</tr>
<tr>
<td>B.3 What Should I Attribute This To?</td>
<td>236</td>
</tr>
<tr>
<td>B.3.1 Ambient, Diffuse, and Specular</td>
<td>236</td>
</tr>
<tr>
<td>B.3.2 Fresnel</td>
<td>237</td>
</tr>
<tr>
<td>B.3.3 Phong Shading</td>
<td>238</td>
</tr>
<tr>
<td>B.3.4 Light Attenuation</td>
<td>239</td>
</tr>
<tr>
<td>B.3.5 Transparency</td>
<td>239</td>
</tr>
<tr>
<td>B.4 Textures</td>
<td>240</td>
</tr>
<tr>
<td>B.4.1 Specifying the Texture Coordinates in the Psurf</td>
<td>241</td>
</tr>
<tr>
<td>B.4.2 Stextures</td>
<td>241</td>
</tr>
<tr>
<td>B.4.3 Normal Mapping</td>
<td>241</td>
</tr>
<tr>
<td>B.4.4 ShrinkWrapping</td>
<td>242</td>
</tr>
<tr>
<td>B.4.5 Face Mapping</td>
<td>242</td>
</tr>
<tr>
<td>B.5 Algebraic Surfaces</td>
<td>243</td>
</tr>
<tr>
<td>C Radiosity</td>
<td>245</td>
</tr>
<tr>
<td>C.1 Introduction</td>
<td>245</td>
</tr>
<tr>
<td>C.2 Environment Modeling</td>
<td>245</td>
</tr>
<tr>
<td>C.3 Radiosity Calculating</td>
<td>246</td>
</tr>
<tr>
<td>C.4 Environment Displaying</td>
<td>246</td>
</tr>
<tr>
<td>D SASS</td>
<td>247</td>
</tr>
<tr>
<td>D.1 Abstract</td>
<td>247</td>
</tr>
<tr>
<td>D.2 Introduction</td>
<td>247</td>
</tr>
<tr>
<td>D.2.1 Using This Manual</td>
<td>247</td>
</tr>
<tr>
<td>D.2.2 For Experienced Users</td>
<td>248</td>
</tr>
<tr>
<td>D.2.3 Installing the Program</td>
<td>250</td>
</tr>
<tr>
<td>D.2.4 Notational Conventions</td>
<td>251</td>
</tr>
<tr>
<td>D.3 A Short SASS Tutorial</td>
<td>251</td>
</tr>
<tr>
<td>D.3.1 Starting the Program</td>
<td>251</td>
</tr>
<tr>
<td>CONTENTS</td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>D.3.2 Scrolling</td>
<td>252</td>
</tr>
<tr>
<td>D.3.3 Creating output files</td>
<td>252</td>
</tr>
<tr>
<td>D.3.4 Saving the spreadsheet contents and Quitting</td>
<td>253</td>
</tr>
<tr>
<td>D.3.5 Advanced Tutorial</td>
<td>253</td>
</tr>
<tr>
<td>D.4 SASS Reference Manual</td>
<td>256</td>
</tr>
<tr>
<td>D.4.1 Anthropometric Spreadsheet</td>
<td>256</td>
</tr>
<tr>
<td>D.4.2 Strength Sheet</td>
<td>262</td>
</tr>
<tr>
<td>D.4.3 SASS-Jack Interface</td>
<td>263</td>
</tr>
<tr>
<td>D.4.4 Database Query Spreadsheet</td>
<td>265</td>
</tr>
<tr>
<td>D.4.5 Exit SASS</td>
<td>271</td>
</tr>
<tr>
<td>D.4.6 SASS - A closer look</td>
<td>271</td>
</tr>
<tr>
<td>D.5 Tricks</td>
<td>274</td>
</tr>
<tr>
<td>D.6 Bibliography</td>
<td>275</td>
</tr>
<tr>
<td>D.7 Appendix</td>
<td>276</td>
</tr>
<tr>
<td>D.7.1 Database Format</td>
<td>276</td>
</tr>
<tr>
<td>D.7.2 Hot Keys</td>
<td>277</td>
</tr>
<tr>
<td>D.7.3 Pull-down menu</td>
<td>277</td>
</tr>
<tr>
<td>D.7.4 Contents by Topic</td>
<td>278</td>
</tr>
<tr>
<td>E CAD Conversion Programs</td>
<td>279</td>
</tr>
<tr>
<td>E.1 Guide to CAD Geometry Conversion</td>
<td>279</td>
</tr>
<tr>
<td>E.1.1 Object Design Within the CAD System</td>
<td>279</td>
</tr>
<tr>
<td>E.1.2 Translating the Data</td>
<td>280</td>
</tr>
<tr>
<td>E.1.3 Grouping and Articulating Your Geometry in Jack</td>
<td>280</td>
</tr>
<tr>
<td>E.2 SDRC Geomod Universal $$\Rightarrow$$ Jack</td>
<td>281</td>
</tr>
<tr>
<td>E.3 AutoCAD DXF $$\Rightarrow$$ Jack</td>
<td>281</td>
</tr>
<tr>
<td>E.4 (Simple) IGES $$\Rightarrow$$ Jack</td>
<td>281</td>
</tr>
<tr>
<td>E.5 Pro/Engineer Render File $$\Rightarrow$$ Jack</td>
<td>282</td>
</tr>
<tr>
<td>E.6 Jack $$\Rightarrow$$ RenderMan RIB File</td>
<td>282</td>
</tr>
<tr>
<td>E.7 Wavefront OBJ File $$\Leftrightarrow$$ Jack</td>
<td>283</td>
</tr>
<tr>
<td>E.8 (Complete) IGES File $$\Leftrightarrow$$ Jack</td>
<td>283</td>
</tr>
<tr>
<td>E.8.1 Conversion</td>
<td>283</td>
</tr>
<tr>
<td>E.8.2 Jack $$\Rightarrow$$ IGES</td>
<td>285</td>
</tr>
<tr>
<td>E.8.3 IGES $$\Rightarrow$$ Jack</td>
<td>285</td>
</tr>
<tr>
<td>E.8.4 User Interface</td>
<td>286</td>
</tr>
<tr>
<td>E.8.5 Supported Iges Entities</td>
<td>288</td>
</tr>
<tr>
<td>E.9 ComputerVision $$\Rightarrow$$ Jack</td>
<td>288</td>
</tr>
<tr>
<td>E.9.1 Introduction</td>
<td>288</td>
</tr>
<tr>
<td>E.9.2 Computervision Basics</td>
<td>288</td>
</tr>
<tr>
<td>E.9.3 What is needed to convert?</td>
<td>289</td>
</tr>
<tr>
<td>E.9.4 Generating a Jack Environment File</td>
<td>290</td>
</tr>
<tr>
<td>E.9.5 Generating a Jack Psurf File</td>
<td>291</td>
</tr>
<tr>
<td>E.9.6 Executing cv2jack</td>
<td>292</td>
</tr>
<tr>
<td>E.9.7 Limitations</td>
<td>292</td>
</tr>
<tr>
<td>F The LISP Interface</td>
<td>295</td>
</tr>
<tr>
<td>F.1 Overview</td>
<td>295</td>
</tr>
<tr>
<td>F.2 The Lisp Interface</td>
<td>295</td>
</tr>
<tr>
<td>F.3 Using XLISP-STAT</td>
<td>296</td>
</tr>
<tr>
<td>F.3.1 The XLISP prompt</td>
<td>297</td>
</tr>
<tr>
<td>F.3.2 Menu Items</td>
<td>297</td>
</tr>
<tr>
<td>F.3.3 Callback Functions</td>
<td>297</td>
</tr>
<tr>
<td>F.4 JCL calls</td>
<td>298</td>
</tr>
<tr>
<td>F.5 Motion Shells</td>
<td>298</td>
</tr>
</tbody>
</table>
List of Figures

1.1 The Jack Screen. .......................................................... 21
1.2 A chair. ............................................................................. 24
1.3 A human figure and a chair. .............................................. 24
1.4 A human figure, a chair and the view reference point. ........... 26
1.5 Moving a chair. .............................................................. 27
1.6 The Help window. .......................................................... 29

2.1 Completion choices in the Jack window. ......................... 33
2.2 Completion choices in the Jack window. ......................... 34

4.1 The view reference point. .................................................. 44
4.2 Jack window with a 90° field of view. .................................. 47
4.3 Attaching the view to a site (instead of the default camera). .... 48
4.4 Adjusting the clipping planes. ........................................... 49

5.1 The chair, with linear movement axis highlighted (global X axis). 52
5.2 The chair, with planar movement axes highlighted (X-Z plane). 53
5.3 The chair, with rotation wheel highlighted (Y axis). ............... 54
5.4 The $\text{xyz}$ Rotation Operator ........................................ 56
5.5 Multiplication of homogeneous transformations (a) right to left ...... 57
5.6 (b) left to right .............................................................. 57
5.7 Adjusting the elbow joint of a human figure. ....................... 63

6.1 Bending the torso via the \texttt{bend torso} command. This example is of flexion. 67
6.2 Bending the torso, with axial twisting. ............................. 68
6.3 Specifying spine bending joints explicitly. ............................. 69
6.4 Rotating the pelvis, with no torso control. ............................ 71
6.5 Rotating the pelvis, with torso control set to \texttt{keep vertical}. ......... 72
6.6 Setting up to make a human sit down. .............................. 74
6.7 Dragging the center of mass down towards the seat. ............... 75
6.8 Rotating the pelvis to relax the upper torso. .......................... 76
6.9 Moving the chair while human figure is attached. .................. 76
6.10 Moving the right foot. ..................................................... 78
6.11 Moving the left foot. ....................................................... 78
6.12 Adjusting the shoulder via \texttt{adjust joint}. ......................... 81
6.13 Attaching a hammer to the left hand. ............................... 83
6.14 Moving the arm using the attached hammer as a reference. ....... 84
6.15 Moving the head by move the focus point. ........................... 87

7.1 Measuring a dimension via the command \texttt{measure distance}. .... 91
7.2 An eye window showing the human figure’s view from between it’s eyes. . 96
7.3 Visualizing a humans visibility via view cones. ....................... 97
7.4 A strength window showing available and required torque along each degree of freedom. 98
LIST OF FIGURES

8.1 The animation window ........................................102
8.2 The animation window zoomed out ............................103
8.3 The animation window zoomed in .............................104
8.4 Scrolling the animation window left .........................104
8.5 Scrolling the animation window right .......................104
10.1 A chain and a small cube, for constraint example ..........126
10.2 The end of the chain is constrained to the cube ........128
10.3 Adjusting a joint that is also under the influence of a constraint 129
10.4 The chain under the influence of two constraints ....130
10.5 Moving the cube ..............................................135
11.1 Adjusting the bottom joint after re-rooting the chain ....142
11.2 The robot after scaling commands ..........................144
11.3 The scaled robot again ........................................144
11.4 The cube and pyramid positioned for the create joint between segments command 146
11.5 Adjusting the newly created joint ..........................147
12.1 The IRIS Lighting Model ......................................152
12.2 Adjusting the color via potentiometers ....................155
12.3 A torus, smooth shaded with specular highlight ..........156
12.4 A torus, flat shaded ...........................................157
12.5 The human figure with a tank top ..........................160
12.6 Supported Image File Formats ..............................164
12.7 A Few Reflection Maps ......................................165
12.8 A Few Plants and Trees ....................................166
12.9 A Few Clouds and Fractals ................................167
12.10 A Few Simple Textures ......................................168
13.1 Extruding one node from a cube ............................170
13.2 Extruding an edge from a cube ...............................171
13.3 Extruding a face from a cube ................................171
13.4 The chair with the deformation lattice appearing around it 173
13.5 Interactively moving a deformation lattice node ........173
13.6 The deformation is not updated until you enter the command evaluate deformation 174
15.1 A shaded window ...............................................183
15.2 After executing make everything shaded. Note the background is wireframe by default 184
15.3 Everything in wireframe, except the head is shaded ....185
15.4 A shaded window on the right, and a wireframe window on the left 188
15.5 The camera for the right window is seen in the left window 190
A.1 .................................................................215
A.2 .................................................................215
A.3 The Precedence of Peabody Arithmetic Operators ....216
A.4 The xyz Rotation Operator ..................................217
A.5 .................................................................228
D.1 Sample Display From SASS: Girth Sheet ..................249
D.2 Anthropometric Spreadsheet Screen Layout ...............256
D.3 Command Menu for Anthropometric Spreadsheet .......258
D.4 Pull Down Menu Window ....................................260
D.5 Dialog Box Display .........................................261
D.6 Strength Sheet Display ......................................263
D.7 Human Figures Created by SASS - Frontal ................265
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.8</td>
<td>Human Figures Created by SASS- Lateral</td>
<td>266</td>
</tr>
<tr>
<td>D.9</td>
<td>Database Query Spreadsheet Screen Layout</td>
<td>266</td>
</tr>
<tr>
<td>D.10</td>
<td>Database Query Sheet Display</td>
<td>267</td>
</tr>
<tr>
<td>D.11</td>
<td>Command Menu for Database Query Spreadsheet</td>
<td>268</td>
</tr>
</tbody>
</table>
Chapter 1

An Introduction to Jack

This chapter is a brief tutorial introduction to using Jack. It demonstrates some of the basic Jack features, illustrating mostly the flavor of how you interact with Jack rather than describing the details of how to use it. The first time you run Jack, you should go through the examples in this chapter. From then on, refer to the later chapters for more details about the various features.

1.1 What is Jack?

Jack is a program which displays and manipulates articulated geometric figures. There are many different aspects of Jack, such as facilities for constructing geometric objects, positioning figures in a scene, performing various types of analyses of the figures, and describing motion of the figures. There are also facilities for specifying lighting and surface property information.

The geometric objects used by Jack are represented by a language called peabody. Peabody has a very powerful and flexible mechanism for representing connectivity between objects, both the joints between individual segments within a figure and arbitrary connections between different figures.

Jack is primarily an interactive system. It is predicated on the belief that geometric operations are best performed interactively and graphically. Most operations in Jack use the mouse, both to pick commands from menus and to specify geometric transformations. Parameters and values may also be entered directly from the keyboard. Whenever possible, the syntax of these values follows the peabody language, reinforcing the idea that Jack is an interactive peabody editor.

Jack may also be controlled non-interactively, through the Jack Command Language, JCL. JCL command files are "scripts" of Jack commands. Most operations which may be performed interactively in Jack may also be performed non-interactively in JCL.

1.1.1 Jack Version 5.8

The current distributed version of Jack is 5.8. This manual originally described Jack Version 5.4 and Version 5.5. In many places it has not changed much. To flag you (the reader) of new features in Version 5.8, there will be a symbol in the margin (the triangle) to alert you that this is a new feature of Jack 5.8. There are also places where features of Jack 5.x are described which differ from Version 4.x. These are marked by the symbol in the margin (the 5 of clubs).

1.1.2 New Features of Jack Version 5.8 from Version 5.5

New Human Figure Jack 5.8 has a new human body model. Your environments which contain human figures from 5.6 or earlier will need to be upgraded. The new human replaces the venerable human5.fig figure. It's most notable attributes are: more human looking; it has proper eyeballs; no more glasses
and hat; better fingers; better feet; you can generate bodies directly from SASS; more geometry in the arms and legs to better approximate the human shape.

**New CAD Translators** A set of new and enhanced CAD file translators are distributed with *Jack* 5.8. In particular, there is a very good IGES translator, both in and out of Jack. There are also guidelines for building geometry in an external CAD system which will be viewed in *Jack*. There are also several new psurf utilities, which can reduce the node and face count in very complex geometry. See Section E and Section H.1.2.

**SASS Version 2.1** SASS has been in development for a while, and this is the first full external release of that program. It helps in creating human figures according to a variety of anthropometric variables. See Section D.1 for details.

**Strength** The torque and strength computations for the human body are much improved. The data for available torque in the human body is from NASA Johnson Space Center. We have the available torque arm values now, and the leg values should be available shortly (this will only be an update to the body definition, all code is already within Jack). See Section 7.8.

**Improved Animation System** The improvements to the animation system include: (1) motion groups, which allow you to group related motions together, and then create motion templates; (2) improved support within Jack for channels, which hold interpolated data, and are more extensible than the older frame structures; (3) motions and channels are now read and written using full Peabody syntax, not JCL. There are also several new motion types which make animating highly articulated figures much easier. See Chapter 8.

**Postures** There are several new commands for dealing with figure postures, as well as extensions to *Peabody* to allow for posture definitions. See Section 6.8 for commands and Section A.3.10 for the *Peabody* reference.

**Articulated Hand** The articulated hand has been improved and the commands for using them are more robust. This will be followed by a more complete treatment of hand controls in the near term future. Commands for controlling the articulated hand are in Section 6.6.7.

**LISP Programming Interface** Jack user's who don't have access to source code can now extend Jack's functionality by writing Lisp code. A lisp interpreter is embedded in Jack, as well as lisp functions to access the internal structures of peabody. This represents a very powerful way for users to extend Jack, and to share those extensions with others. We hope to distribute (in the near future) some sample lisp code which will demonstrate the power and flexibility of this new feature. See Section F.

**JCL Reference Manual** A first cut of a JCL Reference Manual is included. It organizes the JCL commands alphabetically, and gives the full syntax for each command. This should make it easier to build JCL scripts, as well as provide a dense overview of Jack's functionality. See Section G.

**Radiosity** An improved version of the radiosity renderer is included. This version uses a fast over-relaxation progressive refinement algorithm, and will export the finished rendered geometry back into *Jack* keeping the psurf's independent (the older version would create one huge psurf out of an entire environment). A short manual on using the radiosity rendering utilities in included in Section C.

**Improved Video Image Generation** *Jack* can now perform full-scene anti-aliasing and record 60 fields per second. This can dramatically improve image quality generated directly from Jack (when you're not using the ray-tracer). See Section 8.12.1 and Section 8.12.2.

**Interactive Mirrors and Shadows** *Jack* 5.8 includes a very preliminary release of interactive mirror and shadow generation. This should be very useful for analyzing visibility, as well as generating nicer images for presentation. See Section 17.15.

**Reflection Mapping** The texture mapping commands now include a reflection mapping mode, to take advantage of this feature of the SGI hardware. See Section 12.8.1.
1.1. **WHAT IS JACK?**

More Image File Formats Supported *Jack* has a mechanism for supporting arbitrary image file formats (used for texture mapping and saving window images). See Section 12.8.2.

Utah Raster Toolkit 3.0 *Jack* now uses Version 3.0 of the URT. A subset of the URT is distributed with *Jack* and the man pages for those utilities are included in Section II.3. The URT is public domain software, and is included as a convenience to the *Jack* user.

View and Clipping Planes Support for user-definable clipping planes (up to 6) as well as commands for pushing and popping the view associated with a window. *Jack* also will push the view in a window when an automatic viewing adjustment is made, so you can then later pop back to the previous view. See Section 4.5.1.

Rulers You may create rulers which continuously read out the distance between two points in the environment. This is very useful for measuring fit and reach while placing or animating a human figure in a work environment. See Section 7.2.1.

Finer Grain Collision Detection The collision detection routines have been greatly improved. They allow for tracking collisions between any two sets of segments in the environment, as well as choices for different intersection algorithms and display options. See Section 17.1 for details.

Remote Commands *Jack* can receive commands from remote processes through a *command port*. There are also a set of special JCL commands intended for use from remote processes. The Lisp interpreter can also communicate with external process through a *lisp port*. See Section 17.6.

Paths You can now create an explicit path in *Jack* which can then be used for animation or other uses. See Section 17.7.

Joint Motors Joint motors are functions that allow the user to continuously "exercise" a degree of freedom for a particular joint. See Section 11.7.

Real-time Animation Preview *Jack* can now preview your animations at close to 30 frames per second on an Iris workstation, using the SGI utility *movieplayer*. See Section 8.11.

Flock of Birds 6 DOF Sensor Support for Ascension Technology's Flock of Birds 6 DOF Sensors has been added to *Jack*. You can use a bird to control the position and orientation of a figure or site in the Jack environment. We will also distribute the source code for the bird server process, so that you may easily modify it to support other types of sensors. See Section 17.11 for details.

Contributed programs There are several utility functions that are released with *Jack* as "contributed" commands. They aren't directly supported for now, but are released so that you can use them and give us feedback on their usefulness and applicability. Some of them include: (1) commands for using the GL fog parameters, to add fog to your *peabody* windows, (2) fractal geometry generation commands, (3) network commands for connecting two or more *Jack* processes with a shared environment, and (4) some new collision avoidance behaviors for the human figure. See Section 17.8.

Miscellaneous New Commands There are several new commands that were added by request from several *Jack* users. They are documented throughout the manual, and highlighted by the symbol in the margin.

1.1.3 *Jack* Version 5.x (up to Version 5.5)

*Jack* 5 has several new features and enhancements over previous versions. Most notably, *Jack* 5 has a much richer notion of human figures and how to interact with them. You will find commands for interacting with human figures on the human menu. *Jack* 5 also has a new animation system which is based heavily on *Jack's* inverse kinematics algorithm. The old animation system in *Jack* Version 4 is not available in Version 5.

If you are already familiar with *Jack* Version 4, you will find that the most basic things have not changed, although there are many new facilities. These are described throughout this manual, but they are highlighted below. This manual highlights the new features of *Jack* Version 5 with the symbol in the margin, like the one
1.1.4 New Features of Jack Version 5.x from Version 4.x

Almost everything regarding human figures in Jack Version 5 is new or improved. Chapters 6, 7, and 8 of this manual are completely new, and you should read them even if you are already familiar with Jack. The following is a list of small and miscellaneous changes and improvements that are described in greater detail in other sections.

Pop-up menus stay popped up You can now click the mouse and get the pop-up menus to stay on the screen without holding the mouse button down. This is described in Section 1.5.2.

Pop-up menus for reading files You can now choose files from pop-up menus. This is described in Section 2.5.

Arrow keys in edit buffer The left and right arrow keys now work in the edit buffer. This is described in Section 2.1.

Arrow keys for rotation You can now rotate figures by 90° at a time using the arrow keys while executing move figure. See Section 5.9.

New snapping options There are some new snapping options for use in move figure and other movement commands. See Section 5.10.

A better way of attaching figures to other figures The attach figure command works much better than rooting constraints. See Sections 6.4.2.1 and 6.6.4.

Free-form deformations Free-form deformations allow psurfs to be stretched, bent, and twisted. See Section 13.2.

Articulated figure creation New commands make it easy to create articulated figures and add joints to old figures. See Section 11.6.

Distance measuring The measure distance command allows you to measure distances between objects. See Section 7.2.

New surface attribute generation Jack 5 has a slightly new way of handling surface attributes which gives them more intuitive names. See Section 12.1. It also has a new and better mechanism for picking attributes interactively, described in Section 12.3.
1.2 How to Use This Manual

This manual is structured like a tutorial, with many tutorial examples. Each chapter addresses a particular aspect of Jack, through descriptions of its functionality and tutorial examples which step you through using its features. The material in this example proceeds logically from the most basic features to the more advanced ones, so you should generally proceed through the manual from beginning towards the end. Occasionally, you may be required to know about some feature which is described later on, but in these cases, you will be directed to those sections explicitly.

This chapter is a very thin overview of using Jack. It presents several of the most basic Jack commands to give you a flavor of how Jack works. This chapter is not meant to be a comprehensive presentation of each of the features it discusses.

1.3 Technical Support

If you are having problems with Jack and you are a licensed user, you may call the Computer Graphics Research Lab at the University of Pennsylvania at 215-898-1488, or the preferred method, send e-mail to jack@graphics.cis.upenn.edu.

To get licensing information about Jack call the Center For Technology Transfer at 215-898-9585.

There is an Anonymous FTP site at ftp.cis.upenn.edu (or 130.91.6.8). This is where I place additional Jack-related programs and data files between formal Jack releases. Currently, there are several useful files there. To log in, you would do something like the following:

`% ftp ftp.cis.upenn.edu
...user-id: anonymous
...password: your-login-name (e.g. I put granieri@graphics.cis.upenn.edu)
ftp> binary
ftp> cd pub/graphics
ftp> ls
ftp> get demojack.tar.Z
ftp> quit`

1.4 DemoJack

There is a pre-canned demonstration version of Jack called demojack, which is distributed with Jack (in the $JACK/demojack/ directory). It has two environments, one with a human connected to the cockpit of an Apache helicopter, and one with a human sitting in the driver’s seat of a car. This is useful for giving quick demos of Jack.

You must start demojack from the $JACK/demojack/ sub-directory on the distribution. There is an extra menu item on the top of the main menu, which is built from the file .jack.menu which sits in this directory. You can use your own .jack.menu file for creating your own demos also. Take a look at the file after running demojack.

1.5 Preliminaries

1.5.1 Jack and the Window Manager

Jack uses the window manager for its windowing facility and pop-up menus. If you are familiar with other programs on the IRIS, you will recognize the “look and feel” of the windows and menus. You may generally manipulate the windows on the screen with the commands in the “title bar” menu, which pops up when you press the right mouse button down when the mouse cursor is in the title banner of the window. These commands allow you to move and reshape windows, as well as change their ordering on the screen. You also
get the title bar menu when you press the right button down when the cursor is along the border of the window.

With 4Dwm, the default window manager on the SGI IRIS Workstations, the input focus "follows the mouse." In other words, the input focus belongs to the window which is currently under the mouse cursor. The window which currently has the input focus has a light brown border. Normally, the border is light gray. To "attach" to a particular window, simply move the mouse over that window. Its border should turn brown.

Jack allows you to have multiple windows simultaneously, each with a different view of the geometric environment. These windows are sometimes referred to as "Jack windows," since they are created by Jack. Technically, these are "peabody windows" since they display peabody objects. Jack has other kinds of windows which it allows you to create, such as "meter" windows with potentiometers for adjusting color values. We use the term "Jack window" to describe any window created by Jack and to distinguish them from ordinary text windows created elsewhere.

When Jack has more than one window, it always directs its operations to the current window, i.e. the one which has the input focus. For instance, changing the view will change the view for the currently highlighted window. If you want to change the view for another window, you must first attach to that window.

NOTE: There is a conflict between the default key bindings for 4Dwm and the keys that Jack uses for local rotations. In your 7.4Dwmrc file, you should see a section which looks something like:

Buttons 4DwmButtonBindings
{
   Shift Ctrl<Btn1Down> window f.raise_resize
   Shift Ctrl<Btn2Down> window f.raise_move
   Shift Ctrl<Btn3Down> window f.post_menu
}

These lines should be commented out by placing a # character at the beginning of each line.

1.5.2 Using the Mouse

Jack uses the mouse for most of its input. You should arrange the mouse pad next to you in a comfortable position. It is best to grip the mouse from its side so that you have two fingers available to press down the buttons. You will often have to hold down more than one button at a time while you move the pad around. Also, make sure that the optical pad doesn’t slide around on the table.

It is also important when using an optical mouse to keep the mouse squarely on the pad. The sensor in the mouse will not work properly if it is sitting diagonally on the pad. As you move the mouse around, be sure that it stays at right angles with the pad.

We will sometimes refer to the little red arrow on the screen as the mouse, but technically the mouse is the device on the table beside you. Since the mouse controls the position of the cursor, they seem like the same thing. When we say “move the mouse to the upper part of the screen” we actually mean to move the mouse so that the red arrow moves to the upper part of the screen.

Jack has a new way of using the mouse to select commands from the pop-up menus. If you press down a button and hold it down, the pop-up menu will appear as usual. However, you may also click the mouse quickly and the menu will appear after your release the button. The menu will then stay there and let you peruse the menus without having to hold down the mouse buttons. If you do it this way, click the mouse again over the command you want to execute.

1.6 Running Jack

To run Jack simply type the command jack. The Jack screen will appear with a large graphics window, a small text window below it, and a one-line status window across the bottom of the screen.

The graphics window shows the standard blue ground plane. The status window is very important because this is where Jack displays information about what it is doing and what it expects you to be doing.
Figure 1.1: The Jack Screen.
You interact with Jack by executing commands, which you can select through the mouse and keyboard. In its normal state, Jack is waiting for input from you. To select a command through the pop-up menus, press down the right mouse button and hold it down. The main Jack menu will appear.

Menu items which have an arrow on the right hand side are sub-menus. When you slide the arrow off that side of the menu, another menu will appear. To execute a command, release the mouse button while the item is highlighted. You may want to take some time to explore the menus. Most commands require you to enter other arguments, so at this point you should not execute any commands. If you accidentally execute a command and do not know what to do, you can abort the command by hitting Ctrl-C.

Important Note! In order to execute Jack commands, the mouse cursor must be in either the graphics window or the status window. If the mouse cursor is in the message window, Jack will not respond!

There are also several options you may specify on the command line when starting Jack. The Jack command line is of the form:

```
jack [options] <files ...>
```

where [options] are of the form:

- `-4` 4-panel screen
- `-B` black and white windows
- `-E <cmd>` execute the JCL command `<cmd>`
- `-H <colors>` color indices for help window
- `-L <colors>` color indices for log window
- `-M <left,right,bottom,top>` message window location
- `-P` don't display projections
- `-S` make window shaded
- `-T <r,g,b>` text color
- `-W <left,right,bottom,top>` window location
- `-j` don't read init file (`~/jack.jcl`)
- `-s <scale>` set window scale parameter
- `-w` open smaller window (1/2 screen size)
- `-A <left,bottom,nc,nr,fontname>` open log window at `<left,right>` with `nc` columns and `nr` rows with font `fontname`.

### 1.7 Reading in an Object

Jack is a system for manipulating and analyzing geometric objects. Object models in Jack are stored in files
1.8. Creating a Human Figure

Called peabody files, Peabody is a language for describing the geometric properties of objects. Jack creates objects by reading these files. It also stores your work by generating peabody files which can be read later on. Jack represents both geometric objects and human figure models through peabody.

These files can come from several places. There are several conversion utilities which convert objects designed in other CAD systems to peabody. If you have generated a peabody file through one of these conversion programs, then you will want to read in that peabody file.

Jack also has several sample peabody files which come with its installation. For the purposes of this demonstration we will read one of the standard files which describes a chair, called chair.pss.

\[\text{TUTORIAL 1.1: READING A FILE}\]

1. Execute the command read file
   The read file command is on the create menu.

2. Enter the name of the file to read: chair.pss
   When you have entered chair.pss, hit ENTER. If you make mistakes, you can correct them by hitting the DELETE key.
   When you hit ENTER, Jack begins reading the file. It will display some messages on the screen. These messages give information about the model being read. The message window gives a status report as Jack reads the file. This model is not very complex, so it does not take much time to read.

1.8 Creating a Human Figure

Jack 5 models figures in a more complex way than earlier versions. The command create human figure is now on the human menu. It creates a human figure and the constraints which model the human figure's behavior.

\[\text{TUTORIAL 1.2: CREATING A HUMAN FIGURE}\]

1. Execute the command create human figure.
   It will prompt you to enter the name of the figure. It will provide a default name, but you may name it anything you like. The name may be descriptive, like male50, or it can be more humanistic, like maynard or herb. You could also call him jack, but remember that Jack is the name of the program, not the figure!

2. Enter a name for the figure.
   In this example, just accept the default name by hitting ENTER.
   Jack will begin reading the file. It will display some messages in the message window about which files it is reading. When it finishes, the figure should appear:
   If this process didn't work or the figure looks substantially different from this one, you may not have the human figure model properly installed. Refer to the directions for installing Jack, which are distributed with it.
CHAPTER 1. AN INTRODUCTION TO JACK

Figure 1.2: A chair.

Figure 1.3: A human figure and a chair.
1.9. CHANGING THE VIEW

You may notice that this figure looks as if he has clothes on. Actually, they aren’t clothes: his segments are just colored to look that way. You may alter this definition with the commands on the anthropometry menu, described in Section 7.1.

1.9 Changing the View

*Jack* uses a camera metaphor to represent the view in the graphics window: imagine looking into the window through the lens of a camera. To change the view, we move the camera.

**TUTORIAL 1.3: CHANGING THE VIEW**

1. **Execute the command** `change view`  
   This command is on the `view` menu.

   ![Menu](image)

   When you do this, a yellow cross-hair will appear in the middle of the screen and the message window will display a special message describing what options are available.

   The yellow cross-hair is the view reference point, an imaginary point directly in front of the camera along its line of sight.

   You can change the view by moving the mouse. The three mouse buttons have different meaning.

2. **The left button controls horizontal swing**  
   When you hold down the left mouse button and move the mouse back and forth horizontally, the camera swings around the reference point on a circular track at the same elevation.

3. **The middle mouse button controls vertical swing**  
   When you press down the middle mouse button and move the mouse up and down vertically on its pad, the camera swings up and down around the view reference point at the same longitude. You may go over the top and turn upside down if you want.

4. **The right button controls zoom**  
   When you press down the right mouse button and move the mouse up and down vertically on its pad, the camera moves in and out, getting closer or farther away from the view reference point. You cannot, however, go through the reference point.
5. Try pressing two buttons at once

You can swing horizontally and vertically at the same time by holding both the left and middle mouse buttons down at the same time.

The right mouse button controls zoom, i.e. the distance of the camera from the view reference point. When you hold down the right button and move the mouse up and down, the camera moves farther away from or closer to the reference point. You can’t zoom through the reference point.

6. Hit ESCAPE to stop

To terminate the view change process with the new view, hit the ESCAPE key. This will save the view in the current location and Jack will return to its normal state.

There are other ways to move the view as well. These are described in Chapter 4.

1.10 Moving a Figure

Jack lets you move objects interactively with the mouse. Its movement technique provides a way of interactively manipulating of the geometric transformations which define the position, orientation, and even the shape, of peabody objects.

All moving commands use the same basic approach. In this example, we will explore moving a figure. Make sure that you have already read in a figure as described above.

♦ Tutorial 1.4: Moving a Figure

Do this after you have done the previous tutorials. You should have a human figure and a chair.

1. Execute the command move figure

The move figure command is on the move menu, which is a sub-menu of the edit menu.
1.10. MOVING A FIGURE

This command will prompt you in the message window to pick a figure to move.

| Pick a figure to move | ^K keyboard | mouse -> pick | ^C,ESCAPE abort |

2. Click the mouse on the chair
   Place the mouse cursor over the figure and click the mouse. When you have selected the figure, a yellow cross-hair will appear at the origin of the figure.

3. Press and hold down one of the mouse buttons
   You can now begin moving the figure with the mouse. The left button translates the figure along its x axis. The middle mouse button translates the figure along the y axis, and the right button translates along z. When you select axis by pressing one of the buttons, a red arrow appears to illustrate the axis you have selected. This arrow also indicates the direction you should move the mouse on the screen.

4. To stop moving, hit the ESCAPE key
   This will leave the figure in its current position. If you want to erase the movement you have just done and return the figure to its previous position, you can abort by hitting ^C.

5. Repeat the tutorial with the human figure.
   You can move human figures just like other types of figures. However, if you are already familiar with how to move figures in Jack, and you know how to rotate figures, you will
notice that the human figure rotates differently. You can rotate him around the y axis just like before, but if you try rotating him around the x or z axis, strange things happen. This behavior is described in Section 6.1.1.

The movement technique is described in greater detail in Chapter 5.

1.11 Writing the Environment to a File

*Jack* allows you to save the geometric environment in files called *peabody environment* files. You can read this file back in later on with the *read file* command.

** Tutorial 1.5: Writing an Environment File**

1. **Execute the command** *write environment*

   This command is on the write menu.

   ```
   write
   write environment
   write figure definition
   write psurf
   write psurf globally
   write figure psurfs
   write environment to psurf
   write positions
   write figure position
   write global positions
   write frames global positions
   save window setup
   write JCL log
   ```

   This command will prompt you to enter the name of a file:

   ```
   environment file: /project
   ```

2. **Enter the name of the file you want to create**

   The initial value in the buffer is the current directory. Unless your name is "smith," it will likely be something else. You must enter the name of the file by appending it to the end of the directory name. The file may have any name you wish, although it must have the suffix .env. In this example, enter *test.env*, so the contents of the buffer are something like:

   ```
   environment file: /project/test.env
   ```

   Then hit ENTER. If *Jack* asks you any questions, enter the default answers. These are some options that will be discussed later.

3. **Exit Jack**

   This is described in the next section.

4. **From the Unix shell, type ls.**

   You should see the file *test.env* there.

5. **From the Unix shell, type more test.env.**

   This will type type file. You can see that the file is quite readable. This is the peabody language. It is described in Appendix A.

6. **Run Jack again, and execute the read file command.**

   Read the file *test.env*. The human figure and the chair should appear again just like they were before.
1.12 Getting Help

*Jack* has an on-line help system which lets you get information about individual commands and about how to do certain things. Most of the information in this User's Guide is available on line, although in slightly different format.

The On-line help is triggered with the question mark key ?. Whenever you press the question mark key, *Jack* will create a *help window* in the middle of the screen, and it will post a bottom-up description of what you are currently doing. You can page through the help information to read what is of interest to you. There are usually several help screens active at once, each pertaining to a different aspect of the current operation.

![Figure 1.6: The Help window.](image)

**Tutorial 1.6: Getting On-line Help**

Before doing this tutorial, make sure you have read in at least one figure, like the human figure, for example.

1. Execute the command `move figure`.
2. Hit the question mark key ?
   When *Jack* prompts you to pick a figure, hit the question mark key ?. A help window will appear in the middle of the screen and message in the upper left corner of the graphics window will say:

   - current help screens:
   > 1. Picking Figures
2. The "move figure" command

The current help screen describes the technique for picking figures. If you page to the next help screen, it will give you information about the move figure command itself.

3. Page through the help screens by following the instructions in the prompt.
When you are done, the help window will disappear, and the status window message will change back to the figure-picking prompt.

4. Pick the human figure
You are now moving the figure.

5. Hit the question mark key ?
The help window will re-appear and this time the message will say:

```
current help screens:
  > 1. Moving Objects Interactively
  2. The "move figure" command
```

Now the current help screen describes the moving technique. This information roughly follows Chapter 5.

1.13 Browsing through Jack Commands

You can browse through the on-line help screens with the command browse. When you execute this command, Jack will let you select commands or enter them from the keyboard, but it will not execute them. Instead, it will just print help screens which describe what the commands do. To stop browsing, execute the command stop browsing. These commands are on the command menu.

★ Tutorial 1.7: Browsing through Jack Commands

1. Execute the command browse.
You can either execute the command from the command menu, or just type browse. Jack will display some special messages on the screen reminding you that you are browsing.

2. Execute the command move figure.
Instead of prompting you to pick and move a figure, Jack will display the help screen about the move figure command in the help window.

3. Execute a few other commands to get the feel for it.
Some commands have several layers of information.

4. To stop browsing, execute the command stop browsing
It is on the menu.

1.14 Exiting Jack

To exit Jack, execute the command quit on the main menu. This will ask you to confirm your action, so that you will not accidentally exit the program and lose your work.

```
quit jack?
don't quit
```
Chapter 2

Using the Keyboard

This chapter describes how to interact with Jack through the keyboard. Jack is primarily an interactive system, and it derives most of its input from the mouse. However, from time to time, Jack will ask you to enter values from the keyboard. When it does, it allows you to edit the values that you enter. Not only can you erase characters, but you can back up into the string with the arrow keys and insert new characters. This allows you to alter a string once it is in the buffer. For this reason, the message window is sometimes referred to as an edit buffer.

2.1 The Edit Buffer

In addition to the left and right arrow keys, the following control characters have special functions in the edit buffer.

- A move cursor to the beginning of the line.
- B move cursor backward one character.
- C abort the entry.
- D delete the character underneath the cursor.
- E move cursor to the end of the line.
- F move cursor forward one character.
- G move cursor forward one word
- H move cursor back one word.
- K delete characters from the cursor to the end of the line.
- U delete the entire line.

left arrow move the cursor backwards one character.
right arrow move the cursor forwards one character.

The process of entering a value from the keyboard is terminated by hitting ENTER, and the entire string as it appears is entered, even if the cursor is currently in the middle of the line. You may abort the process by hitting "C. What happens when you abort depends upon what you are entering, but it will usually mean that the current operation will be aborted as well, and Jack will return to its previous state.
2.1.1 Default Values

Many operations which request input from the user provide default values. These are values which are placed in the buffer before you begin entering the value from the keyboard. If this value is the one you want, you may simply hit ENTER, and the value will be entered as if you had typed it. You may also edit this value, or delete it entirely by hitting "U" and enter a completely new value. Many times, the default value provided for a certain parameter will be its current value.

2.2 Automatic Completion

Sometimes Jack requests that you enter one of a fixed number of strings from the keyboard. When the set of legal entries is fixed, Jack can automatically complete the selection from a unique substring of the entry. To do this, hit the Tab key. The buffer will be filled out with as many characters as can be determined from the current string and the set of legal entries. A screen message will also be printed giving the completion choices.

Note: In Jack Version 5.8, the completion key is Tab, not the space bar. We changed it because space bar tended to confuse a lot of new users, and we thought Tab is a better choice.

The automatic completion is performed on file names as well. When Jack prompts you to enter a file name from the keyboard, you may enter part of the path and then use the Tab key to complete the name as uniquely as it is defined.

▲ Tutorial 2.1: Automatically Completing File Names

1. Execute the command read file
   This command is on the create menu. It will prompt you to enter the name of the file you want to read:

   file:

2. Hit the Tab key
   Don’t hit enter yet. You will get a listing of files which you can read in. They are a combination of the ones in the current directory and the ones which you have installed. These are the files you can enter by name alone. You can read in any file you like by entering its complete path name.
   Your list of files may be different from this one because you may have other files installed. This is ok.

3. Hit the letter c and then the Tab key again

   file: c

   You will get a listing of all files which begin with the letter c

4. Hit the letter h and then the Tab key again
   You will get a listing of all files which begin with the letters ch. However, if there is only one file which begins with ch, such as chair.pss, then Jack will automatically fill in the rest of the name. This is automatic completion.

5. Hit ENTER to read the file, or "c to abort

2.3 Using the Mouse as an ENTER key

When Jack prompts you with a default value, you may accept this value by hitting the ENTER key. Alternatively you may accept the value by clicking the left mouse button. When you press down and release the button, Jack will ask you to confirm that you do in fact want this entry. If you do, click once again.

1 See Section 9.3 for a discussion of installed files
2.4. ENTERING COMMANDS FROM THE KEYBOARD

The completion choices are:

```
./
./
ground.pss  groundmesh.pss  human.fig
rueuwcone.pss  scallcube.pss  sphere.pss
torus.pss  camera.pss  chair.pss  cube.pss  cylinder.pss
```

Figure 2.1: Completion choices in the Jack window.

### Tutorial 2.2: Using the Mouse as an ENTER Key

1. Execute the **read file command** and enter `human5.fig`.
   
   Jack will prompt you to enter a name for the figure. It will provide you with a default name `human5`.
   
   ```
   figure name: human5.
   ```
   
   If you have read in more than one figure, the name may be slightly different, since all figures in Jack must have unique names. This name is taken from the name of the file from which the figure was read. This name is acceptable, although Jack gives you the opportunity to give it another name.

2. Click the left mouse button
   
   Press it down and release it. This is similar to hitting the ENTER key except that Jack will ask you to confirm your entry:
   
   ```
   you want 'human5'?
   ```

3. Click the left mouse button again
   
   This confirms the entry and Jack begins reading the file.

4. If you don't want that entry, hit the Tab key
   
   This will abort left mouse click and return you to editing the string.

When Jack prompts you to select from the keyboard among a few specific values, you may also enter the value via a pop-up menu which you get by pressing the right mouse button. Pick the value from the menu just like the other menus.

### 2.4 Entering Commands from the Keyboard

Typically, you execute Jack commands by selecting them from the pop-up menus and picking the arguments with the mouse. You may also execute commands by typing them in directly from the keyboard. To do this, just type away! When Jack displays the message,
press right button for main menu

it is waiting for you to do something. If you press a mouse button, it expects you to select a command from the pop-up menus. If you press a key on the keyboard, it expects you to enter a command, and the prompt will immediately change.

As an example, execute the change view command from the keyboard:

♠ Tutorial 2.3: Executing a Command from the Keyboard

1. When Jack is displaying the default prompt, hit the letter c
   The message in the message window will change to an edit prompt:
   
   ![Completion choices]
   
   At this point, you may either continue typing the rest of the command or you may use automatic completion by hitting the Tab key.

2. Hit the Tab key
   You will get a listing of all of the commands which begin with the letter c.

3. Hit the letter h and then hit the Tab key again
   The text in the buffer will automatically expand to:
   
   ![Completion choices]
   
   This is because there other are no commands which begin with ch except ones which begin with change. Since there is nothing else that you could legally type except for change, Jack filled it in for you.

4. Hit the Tab key again
   You will get a listing of all of the commands which begin with the letter change.

5. Type v and then hit the Tab key
   The text will expand to
   
   ![Completion choices]
   
   ![Completion choices]
2.4. ENTERING COMMANDS FROM THE KEYBOARD

6. Hit ENTER and Jack will begin executing the command
7. Hit ‘U and the entire line will disappear

Jack commands have arguments, which are the objects and parameters which the command operates on. You are required to enter or pick these after executing the command. When you enter a command from the keyboard, you may enter all or some of its parameters at the same time. If you enter just the name, you will be prompted to pick or enter the needed arguments.

♣ TUTORIAL 2.4: ENTERING COMMAND ARGUMENTS

1. Type read file from the keyboard and hit ENTER
   Jack will prompt you to enter the name of the file you want to read.

   file:

2. Type human5.fig and hit ENTER
   Jack will then prompt you to enter a name for the figure.

3. Hit ENTER to accept the default name
   Jack will read the figure.

4. Type read file human5.fig and hit ENTER
   This in effect enters the file name ahead of time. Jack will then only prompt you for the name of the figure.

5. Type read file human5.fig fred and hit ENTER
   This enters the command name and all its arguments, so Jack proceeds with reading the file without asking for any more information.

You may recall previous commands by hitting ‘P when you are editing this buffer. When you hit ‘P, the buffer will be filled with the previously executed command and its arguments. Hit ‘P again and you move to the command executed before that. If you hit ‘N, you move to the next most recently executed command. To experiment with this facility, execute several Jack commands such as read file, change view, and move figure.

♣ TUTORIAL 2.5: RECALLING PREVIOUSLY EXECUTED COMMANDS

1. From the default prompt, begin entering a command by typing any letter
   Jack will display the letter you typed in the edit buffer.

2. Hit ‘P
   In the edit buffer will appear the previously executed command, with a slight change in syntax. If the previous command which you executed was a change view you will get something like:

   change.view("jack.sindou",xyz(-21.80deg,36.60deg,13.42deg) * trans(400.00cm,300.00cm,500.00cm));

   This is the syntax in which Jack remembers the commands you enter. It is called JCL, or Jack Command Language. Jack can read commands from files in this syntax. The blank spaces in command names are replaced by underscores, the arguments to the command are surrounded by parentheses, and text strings are enclosed in double quotes. When you enter commands from the keyboard, you do not need to follow this syntax directly.

3. Use the arrow keys and control keys to alter the string
4. Hit ‘P again
   The next previous command will appear.
5. Hit `N
The change view command will reappear: it is the next command.

This gives you a way of re-executing previous commands. Jack allows you to re-execute the single previous command by hitting the `! key, but it is often convenient to re-execute commands which were issued several commands ago. Remember that the message window is an edit buffer, so you may alter the commands any way you like to change the arguments or omit the arguments altogether.

2.5 Entering Files from Pop-up Menus

Jack allows you to enter files from the pop-up menus as well as keyboard. In the read file command, when Jack prompts you to enter the name of the file you want to read, click the right mouse button. A menu will appear:

<table>
<thead>
<tr>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td>environment files</td>
</tr>
<tr>
<td>figure files</td>
</tr>
<tr>
<td>psurf files</td>
</tr>
<tr>
<td>JCL files</td>
</tr>
</tbody>
</table>

You can read environment files, figure files, psurf files, or JCL files. These files are the ones which you can enter by name alone. They are the ones which are either in the current directory or are installed. You can read any file you like by entering its complete path name, but you can only choose the installed ones from the pop-up menus.

Jack remembers all file names that you enter and it allows you to retrieve them using `N and `P just as with entering commands. Anytime Jack prompts you to enter a filename, either for reading or writing, you can hit `P to retrieve the name of a file you entered previously.
Chapter 3

Peabody Objects

This chapter describes the general structure of peabody objects and how to interact with them. It gives the details for how pick object with the mouse, using the commands on the info menu as examples.

3.1 Peabody Terminology and Concepts

A peabody environment consists of figures, each of which is a collection of segments. The segments are the basic building blocks of the environment. Each segment has a “geometry.” It represents a single physical object or “part”, which has shape and mass but no movable components. The geometry of each segment is represented by a psurf, which is generally a polyhedron or a polygonal mesh.

Jack is different from some other modeling and animation programs in that it has a very well-developed notion of articulation, i.e. geometric objects are connected together. Jack begins with the idea that objects will have joints, the basic element of articulation. Joints are Jack’s way of maintaining a hierarchy, but it does so in a very general way.

Joints in Jack connect segments within the same figure. Joints may have specific degrees of freedom, which describe the transformation between the connected segments as rotations and translations around specific axes. The axes may also have upper and lower limits.

Joints connect segments through attachment frames called sites. Sites are reference frames relative to each segment. Each segment generally has several sites. In addition to serving as the attachment points to joints, sites also act as “handles” at notable landmarks on figures. Many operations in Jack refer to sites because they are convenient way of referencing specific points on the figure.

The term “figure” applies not only to articulated, jointed figures such as a human body: any single “object” is a figure. It need not have moving parts. A figure may have only a single segment, such as a coffee cup, or it may be composed of several segments connected by joints, such as a robot. We sometimes use the term “object” to refer to any generic part of the peabody environment.

The term psurf refers only to the representation for the geometry of a segment, which is the graph of nodes and edges typically drawn as the wireframe or shaded image of the segment. In the case of a figure with a single segment, it is sometimes convenient to refer to it as a “psurf”, but that is not technically correct.

Peabody objects are defined and stored in files in the syntax of the peabody language. The details of the syntax of the language are described in Appendix A.

3.1.1 Peabody Names

Each part of the peabody environment has a name, which you can use to refer to it. Most of the time, you can refer to objects by pointing at them with the mouse, but sometimes you will need to know the names of
objects, such as when you edit a peabody file.

The names of peabody objects must be unique, so that no two objects have the same name. For instance, it is not possible to have two figures with the same name. Segments, sites, and joints have local names, which are their names within the figure to which they belong. To uniquely identify these objects by name, you must qualify the local name with the name of the figure. It is possible, and even common, to have segments with the same name in different figures, such as two human body figures each with a segment named left_upper_arm. The same applies to joints. Sites are a component of segments, so their names must be qualified with both the name of the segment and figure.

3.2 Picking Objects with the Mouse

When Jack performs operations on geometric objects, it requires that you select the objects that you want to operate on. Jack has a way of picking objects interactively by pointing at them with the mouse. It allows you to select among several things which may overlap on the screen. The same basic principle is used to pick all kinds of objects: figures, segments, sites, and joints, as well as individual nodes, edges, and faces associated with each segment.

Whenever Jack is waiting for you to pick something, it will display a prompt in the message window. To pick a figure, position the mouse over the object and press any mouse button, and hold it down. You must position the mouse over the image of the object. If the object is drawn in wireframe, point the mouse at an edge of the object; if it is shaded, point the mouse at the interior of a face. The figure will appear highlighted, a peabody description of it will appear at the top of the window, and the message window will ask you if that is the figure you intend to pick. If it is, then release the mouse button.

As an introduction to the peabody representation, explore the commands on the info menu

These commands prompt you to pick a peabody object, and then they display some information about it. If you need information about several objects, you can re-execute the command by hitting the exclamation point !.

Before exercising these commands, create a human figure model with the command create human figure.

* Tutorial 3.1: Getting Information about Figures

1. Execute the command figure info

   This command is on the info menu. You will get the "pick figure" prompt:
3.3. PICKING OVERLAPPING OBJECTS

Pick a figure -> keyboard mouse -> pick ESCAPE abort

2. Move the mouse to the figure and press down one of the buttons
   Hold the button down. The figure will appear highlighted, and a description of the figure
   will appear in the upper left corner of the graphics window.

   ```
   figure ["human5.fig"] human {
     root = body_root_floor;
   }
   figure human->location = trans(0.00cm,0.00cm,0.00cm);
   ```

   This description is in the syntax of the peabody language. The message window will display
   the message:

   ```
   figure 'human'?
   ```

   This is to confirm that this is the figure that you want.

3. Release the mouse button
   If you pressed the mouse button when the mouse cursor was not over the image of a figure,
   you will get a message:

   ```
   that's not a figure!
   ```

   Release the button and you can pick again.

3.3 Picking Overlapping Objects

If objects overlap on the screen, it is not as easy to pick them with the mouse because you may not be
able to point at them uniquely. If multiple objects lie under the cursor, Jack generates a “pick list” of all
the objects underneath the cursor and allows you to select the desired object by cycling through the list.
When this occurs, the message window will inform you of how many objects have been picked and which
one is currently selected. The currently selected one will appear highlighted. If this is the desired object,
release the button. If it is not, then click another mouse button while holding the original button down.
This will cycle through the list allowing you to arrive at the desired object. This is particularly important when
picking sites and joints: since they are points in space, it is sometimes impossible to point at them uniquely.

♣ TUTORIAL 3.2: GETTING INFORMATION ABOUT SEGMENTS

1. Execute the command segment info
2. Press one of the mouse buttons with the cursor over the head
   The head will appear highlighted and a description of it will appear in the graphics window:

   ```
   segment human.bottomhead {
     psurf = "head.pss" * scale(10.00,7.85,22.70);
     site proximal->location = trans(0.00cm,0.00cm,0.00cm);
     site lsocket->location = trans(9.80cm,3.10cm,11.60cm);
     site rsocket->location = trans(9.80cm,-3.10cm,11.60cm);
   }
   ```

3. Release the mouse button
4. Press the left mouse button with the cursor over the shoulder
   Hold the button down. Use the left button this time because you will need to get your finger
   on another button while still holding this one down.
   Depending upon exactly where you pointed, you will get a message like:

   ```
   3 segments picked... 1: segment 'human.center.torso'?
   ```
CHAPTER 3. PEABODY OBJECTS

Jack has generated a pick list, or a list of the segments which lie underneath the cursor. The currently selected segment is the one described in the message window. It is highlighted as well. You can cycle through the list by clicking another mouse button.

5. While holding the left button down, click the middle button
With another finger, press and release the middle button. The right button will work, too.

The prompt will change to:

```
3 segments picked... 2: segment 'human.left.clavicle'?
```

This cycles through the pick list.

6. While still holding the left button down, click the middle button again
The last of the three selected figures will become the current one.

```
3 segments picked... 2: segment 'human.left.upper.arm'?
```

7. Click the middle button one more time
You have cycled through the entire list and there are no more segments:

```
no more segments to choose from!
```

8. To pick a particular segment, release all mouse buttons while it is selected

3.4 Picking Sites and Joints

Some operations in Jack require you to pick sites interactively, but since sites are not physical parts of the peabody environment, they can be difficult to pick using the mouse unless you know exactly where they are. For this reason, Jack allows you to pick sites by pointing at the segment on which the site lies. The general picking mechanism is the same as with segments, except the "pick by segment" facility is in effect when the CONTROL key is held down. To pick in this way, hold down the CONTROL key and press down a mouse button while the mouse cursor is over the segment which contains the desired site. All of the sites on that segment will then be placed in the "pick list" similar to the list of overlapping segments described above. You may then cycle through the sites just as before, by clicking another mouse button while holding the original button down.

♠ Tutorial 3.3: Getting Information about Sites

1. Execute the command site info
2. Press the mouse button with the cursor near the tip of the hand
   There are several sites in the tip of the hand. You may asked to select among them.
3. Execute the site info command again
4. Hold down the CONTROL key
5. Press the mouse button with the cursor over the hand
   This takes all sites which belong to the hand and puts them in the pick list as described above. You should get a message:

```
3 joints picked... 1: joint 'human.left.hand.distal'?
```

If you release the button, you will have picked this site. To cycle through the other sites, click another mouse button.

Joints may be picked in a similar manner. If you know exactly where a joint is, you may pick it by pointing the mouse at it. If you do not know the exact location of the joint, you may pick it by pointing at a segment and then holding the CONTROL key down while you press down the original mouse button. All joints which are linked to sites on that segment are then entered in the pick list.

♠ Tutorial 3.4: Getting Information about Joints
3.5. **PICKING OBJECTS BY NAME**

1. Execute the command `joint info`
2. Press the mouse button with the cursor near the shoulder
   
   The joint center is at the upper end of the lower arm. You must place the mouse cursor very close to the exact location in order to hit it. This can be hard to do. Keep clicking and you will find it.
3. Execute the `joint info` command again
4. Hold down the CONTROL key
5. Press the mouse button with the cursor over the upper arm
   
   This takes all joints which are connected to the upper arm and puts them in the pick list as described above. You should get a message:

   | 2 joints picked... 1: joint 'human.left.shoulder'? |

   If you release the button, you will have picked the shoulder. To cycle through the other joints, click another mouse button.
6. Click another mouse button
   
   The elbow should appear highlighted and the message will change:

   | 2 joints picked... 2: joint 'human.left.elbow'? |

**3.5 Picking Objects by Name**

Normally, Jack allows you to pick objects interactively by pointing at them with the mouse, but there are times when this is not convenient, such as when an object is not visible. Therefore, you may also pick objects by entering their names from the keyboard. To do this, when Jack is prompting you to pick something, hit `-K` and you will get a message such as:

| figure: |

Just enter the name. You may use automatic completion to help type the name. If you do not know the complete name of the thing you want to pick, hit the Tab key. This will give you the list of the possible things to enter.

Since the names of segments and joints are local to each figure, and names of sites are local to each segment, their names must be fully qualified, using a period between the names. For example, a segment named `left.upper.arm` which belongs to a figure named `fred` would be referenced as `fred.left.upper.arm`. A site named `distal` on fred's left upper arm would be referenced as `fred.left.upper.arm.distal`.

Nodes, edges, and faces may be entered using a numerical index after the name of the associated segment, separated by a period. The nodes, edges, and faces are given as indices into the tables associated with the psurf. The indices start at 1. For example, node 1 of the psurf associated with the `left.upper.arm` segment of `fred` would be referred to as `fred.left.upper.arm.1`.

**3.5.1 Picking Objects through Pop-up Menus**

When you hit `-K` to pick an object from the keyboard, you may also pick it from the pop-up menus by pressing the right mouse button. If you press down the right mouse button when Jack is prompting you to enter the name of the object, a menu of all possible objects will appear. To choose the one you want, release the mouse button when the cursor is over the appropriate item.

Alternatively, you may **click** the right mouse button quickly, and the pop-up menu will appear and stay in place. To select the item, click the right button again over the appropriate entry.

**3.6 Picking the Previous Thing**

In some situations, you may want to execute several commands on the same object, such as the same figure, segment, joint, node, edge, or face. Jack remembers the things you pick, and the picking facility allows you to pick the previous “thing” by hitting the `!` key instead of pointing at things with the mouse.
Chapter 4

Changing the View

This chapter describes how to manipulate the view in the graphics window. The commands on the view menu allow you to do this. The most common of these commands is change view. This command is bound to ^V, so you can either execute it by selecting it from the pop up menu, simply hit ^V, or typing change view as a keyboard command.

Jack uses a camera metaphor to represent the view in the graphics window: imagine looking into the window through the lens of a camera. To change the view, you move the camera.

▲ Tutorial 4.1: Changing the View

1. Execute the command change view

   This command is on the view menu.

   When you do this, a yellow cross-hair will appear in the middle of the screen and the message window will display a special message describing what options are available.

   The yellow cross-hair is the view reference point, an imaginary point directly in front of the camera along its line of sight.

   You can change the view by moving the mouse. The three mouse buttons have different meaning.
When you press down any of the mouse buttons, a message will appear at the bottom of the graphics window describing the view reference point, center of projection, and "view up" vectors:

- center of projection: \((400,300,500)\)
- view reference: \((0,100,0)\)
- view vector: \((-0.19,0.95,-0.23)\)

These vectors describe the global location of the camera, the location of the view reference, and the world coordinate vector which lies along the vertical axis of the screen.

2. **The left button controls horizontal swing**
   When you hold down the left mouse button and move the mouse back and forth horizontally, the camera swings around the reference point on a circular track at the same elevation. While holding down only the left mouse button, moving the mouse up and down vertically has no effect.

3. **The middle mouse button controls vertical swing**
   When you press down the middle mouse button and move the mouse up and down vertically on its pad, the camera swings up and down around the view reference point at the same longitude. You may go over the top and turn upside down if you want. While holding down only the middle mouse button, moving the mouse side to side horizontally has no effect.

4. **The right button controls zoom**
   When you press down the right mouse button and move the mouse up and down vertically on its pad, the camera moves in and out, getting closer or farther away from the view reference point. While holding down only the right mouse button, moving the mouse side to side horizontally has no effect.

5. **Press the left and middle mouse buttons down together**
   This causes the camera to sweep both horizontally and vertically. The horizontal motion of the mouse causes the horizontal camera swing. The vertical motion of the mouse causes
the vertical camera swing. Notice that you stay the same distance from the view reference point.

6. **Now, press the left and right mouse buttons down together**
   This causes the camera to move horizontally and zoom in and out at the same time. The horizontal motion of the mouse causes the horizontal camera swing. The vertical motion of the mouse controls the zoom.

7. **Hit ESCAPE to stop...**
   To terminate the view change process with the new view, hit the ESCAPE key. This will save the view in the current location and Jack will return to its normal state.

8. **Or, Hit ^C to abort and go back to the previous view**
   When you hit ^C, Jack will revert back to the view which you had before you executed the change view command.

### 4.1 Changing the View Reference Point

The sweeping operation used so far moves the camera in a way such that it is always focused on the same reference point. It is also possible to do the opposite: to keep the camera position fixed and move the reference point.

To pan the view, hold down the **CONTROL** key. Just as with the sweeping operation, the left mouse controls the horizontal swing and the middle mouse controls vertical swing. The right mouse controls zoom, but in this case the view reference point stays the same distance from the camera, effectively allowing the camera to be pushed and pulled back and forth in space. This is particularly beneficial as a way of positioning the reference point near an area of interest.

**Tutorial 4.2: Panning the View**

1. **While executing the change view command, hold down the CONTROL key**
2. **Press down the left button and move the mouse side to side**
   This time, the camera stays in the same position in space but it pivots side to side. The view reference point moves in a circular arc up and down.
3. **Press down the middle button and move the mouse up and down**
   Now the camera pivots up and down. The view reference point moves in a circular arc up and down.
4. **Press down the right button and move the mouse up and down**
   Now the camera moves in and out, but this time the view reference point goes with it by staying the same distance away. This gives you a way of moving the view reference point around.

### 4.2 Snapping the View

The sweeping mechanism provides a good means of looking at the objects clustered around the view reference point, but it can be difficult to use the panning mechanism to change the location of the reference point. Frequently, we wish to look at a particular object. You can position the view reference point at a particular peabody site while you are moving the view by hitting ^S. Jack will then prompt you to pick what you want to snap to and then ask you to pick the appropriate object. You may snap to sites and nodes. When you do, Jack will shift the view reference point to that object.

**Tutorial 4.3: Snapping the View**
1. **While executing the change view command, hit ^S**
   *Jack* will prompt you to enter the type of object you want to snap to:
   
   ![Snap Options](#)

2. **Press the right mouse button to display the options menu**

3. **Pick the site item**
   *Jack* will prompt you to pick a site.

4. **Pick a site somewhere on the human figure model**
   When you do so, the view reference point will "slide" to it. Then you can change the view as before with the view centered on that site.

5. **Repeat the process but pick the node item**
   *Jack* will prompt you to pick a node. Pick a node somewhere on the human figure model. When you do so, the view reference point will "slide" to it. Then you can change the view as before with the view centered on that node.

### 4.3 The Field of View

*Jack* defines the view in the graphics window using a *field of view* angle. This angle measures the vertical dimensions of the viewing frustum. The horizontal angle depends upon the aspect ratio of the window. The default field of view is 40°. You can change the field of view with the command `set field of view`.

#### Tutorial 4.4: Setting the Field of View

The effect of the field of view is easier to see if there are some objects in the scene. Read in a human figure.

1. **Execute the command** `set field of view`
   *Jack* will prompt you to enter an angle in degrees. The default is 40°.

2. **Enter 90°**
   The objects in the scene will appear smaller or farther away and somewhat warped. Actually, the objects aren't farther away. *Jack* is just fitting more space onto the screen, which means they look smaller. The warping is due to the exaggerated effect of the perspective.

3. **Execute the command** `set field of view`

4. **Enter 10°**
   The objects in the scene will appear much closer. If you back away, you can see that there is less of an effect of the perspective.

#### 4.3.1 A Warning About the Field of View

Do not use the field of view as a means of *zooming* in and out on objects. With extreme angles, the view can be difficult to manipulate. It is much better to move the camera itself with the change view command.

Also, do not set the view of view angle low in an attempt to turn off perspective and get a 2D window. *Jack* has 2D windows for this very purpose. They are discussed in Section 15.6.
4.4 Attaching the View to Objects

Jack represents the view in each window through a site, as described in detail in Section 15.7. Jack allows you to set this site to any site in the environment. This is most useful for visualizing the view from the eyes of a human figure. Jack 5 has special commands for creating eye view windows. The same thing may be accomplished with the attach view to site command as described below. See Section 7.7.1 for a description of creating eye view windows.

Tutorial 4.5: Attaching the View to a Site

This example attaches the view in a window to a site in the eye of a human figure. Make sure you have a human figure model. This example assumes you know how to create another graphics window, using the commands on the window menu. See Section 15.5 for details. This example also assumes that you know how to rotate joints, described in Section 5.11.

1. Create another window
   Do this with the command create ordinary window, then resize and reshape the two graphics windows so that they don't overlap too much on the screen.

2. Position the view close to the left eye of the figure
   You will need to see the orientation of the coordinate frame of the sites in the eye.

3. Execute the command attach view to site

4. Pick the site with the +z axis pointing back into the head
   This is important! The view is defined to look down the -z axis of the site you attach it to. If the orientation is sideways, the view will look out the side of the head.
   The view in the current window will switch to the eye. Most likely, you won't see anything. This is because the figure is probably looking off into space.
   When you execute the attach view to site, Jack redefines the view for the "current" window, i.e. the one the mouse is in when you execute the command. This is the same as for the change view command.

5. Adjust the figure's neck (via the command adjust joint)
   Pick the neck joint by pointing at it in the "other" graphics window, i.e. not the one attached to the figure's eye. You must manipulate the joint from that window as well, because you must be able to see it. Bend the neck down so that the figure can "see" the ground.
6. Execute the command `reset view to camera`.

This command re-attaches the view to its original place on the camera figure. The view should revert to where it was before you executed the `attach view to site` command.

### 4.5 Adjusting the Clipping Planes

*Jack* allows you to adjust the front and back clipping planes to perform "cutaway" views of a scene. The command `adjust clipping planes` allows you to do this using potentiometers to move the clipping planes forwards and backwards.

The front and back clipping planes are always perpendicular to the line of sight. After adjusting the clipping planes, they stay at the same relative distance from the view reference point, so that if you change the view and zoom in and out, the objects should still be clipped the same.

You can reset the clipping planes to their default values with the command `reset clipping planes`. Other commands related to the front and back clipping planes are:

- `snug clipping planes` will move the near and far clipping planes so they closely fit the current environment.
- `set clipping mode` will affect the behavior of the front and back clipping planes. There are two values for the clipping mode: `static` and `snug to environment`. Static is the default mode; clipping planes will not be adjusted automatically. When `snug to environment` is selected, the clipping planes will be snugged, as in the previously mentioned command; then a flag will be set that will cause *Jack* to continuously update the clipping planes to account for camera zooms. For example, zooming the camera away from the scene will push the clipping planes back to keep their position relative to the environment. **NOTE:** this mode will not yet account for camera rotations.

#### 4.5.1 User Defined Clipping Planes

Most SGI workstations allow for 6 user-defined clipping planes to be used in rendering the scene. Support for these has been added to *Jack*. Clipping planes are represented by a wireframe square icon. Commands are provided for enabling and disabling these clipping planes; once a plane has been enabled it can be manipulated using the standard *Jack* animation and manipulation commands. Clipping planes are not saved as part of the environment. Commands relating to user defined clipping planes are:
4.6. VIEW STACK

- enable user clipping plane prompts the user for the clipping plane to enable (0 through 5), then enables this plane and adds the icon to the environment.
- disable user clipping plane prompts the user to pick a clipping plane icon, then deletes the icon, disables and resets the clipping plane associated with it. Alternatively, delete figure may be used to delete the icon.

Clipping planes are active in every window; they clip all objects in the environment, including the background grid but excluding the stars. Projections are not clipped. The user defined clipping planes are very useful when attempting to create cutaway views, or for un-obstructing the view in a cluttered environment.

4.6 View Stack

Each peabody window in Jack maintains a stack of views. The command push view will push the current view, then execute a change view. The command pop view will pop the view stack, and restore that view to the window. These commands are very handy if you are changing the view recursively. For example, you may execute move arm, then X push view, move the arm a bit, then X push view, move the arm a bit more, then pop view twice to restore the view to what it was before you started the move arm. Also, you may execute pop view after Jack makes an automatic viewing adjustment (Jack warns you before doing the automatic view adjustments) to restore the previous view.

There is also a command unobstruct view, which will move the view to get an un-obstructed view of whatever you are manipulating. This is useful if you are working in a shaded environment.
Chapter 5

Moving Figures

This chapter describes how to move figures and adjust joints using the keyboard and mouse. It describes the basic interaction mechanism by which Jack allows you to move geometric objects, using the move figure command as an example. Jack uses the same basic technique to manipulate other kinds of geometric objects, such as psurf geometry. These other commands work in a manner similar to the move figure command.

This chapter introduces the Jack movement technique through tutorials which involve moving a chair. It is best to do these examples with a simple object like this rather than a human figure model because in Jack Version 5, the human figures behave somewhat differently when you move them with the move figure command. This is described in Section 6.1.1.

5.1 The Movement Operator

Jack has several commands for moving objects interactively, and they all use the same basic technique. The movement operator provides direct interactive control over the translational and orientational components of an object using the three-button mouse. It allows you to translate or rotate an object along or around either its own local coordinate axes or the coordinate axis of the world coordinate frame. The axes of rotation and translation are activated by pressing down some combination of mouse buttons and keyboard keys. The translation or rotation is caused by moving the mouse in the proper direction. The required direction of motion of the mouse corresponds to the direction of movement of the object itself.

The movement operator is best learned by example, and we'll explain it here with an example using the move figure command. When you execute the move figure command, you begin the process of moving the object. The three mouse buttons are your primary control over the movement. The movement begins when you press down a button, and it continues until you release the button. The moving process continues in this fashion until you terminate it by hitting the ESCAPE key. Alternatively, you may hit ^C to abort the move, and the object you are moving will snap back to its original position. For convenience, the move figure command is bound to ^F.

TUTORIAL 5.1: MOVING A FIGURE

For the examples in this chapter, use the chair model from the file chair.pss.

1. Execute the move figure command
2. Pick the figure you want to move
   A yellow cross-hair will appear at the origin of the figure which shows its x, y, and z axes. These are the axes along which you can translate the figure. When you press a mouse button you will begin moving the figure.
5.2 Translation

Each button is associated with an axis: the left button controls translation about the \( x \) axis, the middle button controls the \( y \) axis, and the right button controls the \( z \) axis. Pressing down any combination of the three buttons enables translation along the corresponding axes, but you can't translate along three axes at once.

\[ \text{TUTORIAL 5.2: TRANSLATING A FIGURE} \]

Continue executing the move figure command.

1. While executing move figure, press down the left mouse button

A red arrow will appear showing you the orientation of the figure's \( x \) axis. This also shows you the direction you should move the mouse on the screen.

![Figure 5.1: The chair, with linear movement axis highlighted (global X axis).](image)

2. With the left button down, move the mouse in the direction of the red arrow

The figure will move only in the direction of the arrow.

3. Try moving the mouse in some other direction

The figure will not move easily! The figure may move a little, but its motion is only due to the component of your motion which is along the \( x \) axis.

4. Release the left button and experiment with the middle and right

When you press and hold down the middle button, you translate along the \( y \) axis. Move the mouse up, and the object should rise. Move the mouse down and the object should come down. Use the coordinate axis projections as a cue for the position of the object. The right mouse button controls translation along the \( z \) axis. When you press and hold down the right button, you translate along the \( z \) axis.

5. To translate the figure in a plane, press any pair of buttons simultaneously

When you press a pair of buttons at the same time, you enable translation along both axes simultaneously, or in a plane. The two arrows show you which axes are enable, just like before.

6. When an object moves to the edge of the screen, it won't go any further

In order to move an object, its origin must be on the screen. If it is not, you will get a message like:
5.3 Rotation

Rotation is controlled through the CONTROL key. When you are moving an object, the default transformation is translation, described above. When you hold down the CONTROL key, the transformation becomes rotation and the mouse buttons and mouse movements have different meaning. For rotation, the three mouse buttons control rotation around the x, y, and z axes, respectively. Holding down a single button enables rotation around that axis. It is not possible to rotate around two axes simultaneously, so holding down more than one button has no effect.

To demonstrate rotation, continue executing the move figure command and move the object back to the coordinate origin.

**TUTORIAL 5.3: ROTATING A FIGURE**

Continue executing the move figure command.

1. While executing move figure, hold down the CONTROL key

2. Press the middle mouse button

   Hold the button down. A red wheel should appear centered about the y axis. This wheel describes the axis around which the rotation is taking place, which is currently the y axis. Imagine the plane in world coordinates in which the red wheel lies: the plane extends in all directions to the horizon and may cover the entire screen. Look at where the mouse cursor lies on the screen, and imagine the point on that plane which lies under the cursor. There is a green line drawn from that point to the origin of the wheel. This line lies in the same plane as the red wheel!

3. Move the mouse around the origin of the red wheel

   Imagine the green line as an extra spoke on the wheel. As you move the mouse, the line always points towards the cursor, and the wheel (and the object) remains fixed with respect to it. This will be difficult to see if the cursor is very far away from the center of the wheel. It is best to keep the cursor fairly close to the perimeter of the wheel.
4. Release the middle button and press the left button
   The left button rotates around the global z axis.
5. Release the left button and press the right button
   The right button rotates around the global z axis.
6. You can't rotate along an axis perpendicular to the line of sight!
   If you get a message
   \[
   \text{can't rotate around that axis from this view!}
   \]
   then the red wheel is perpendicular to your line of sight and cannot be displayed effectively.
   You must reposition the view before rotating around that axis.

5.4 Terminating the Movement

To get out of the move figure command, hit either ESCAPE or ^C. ESCAPE will leave the figure in its current position. ^C will abort the movement you have just done, and the figure will revert back to its position before you executed the move figure command.

5.5 Changing the Reference of Movement

The rotation and translation operations described so far operate along the global coordinate axes. It is sometimes convenient to translate or rotate along an axis local to the figure itself. You can do this by holding down the SHIFT key. When you hold down the SHIFT key, the rotations and translations take place with respect to the local coordinate axes of the figure you are moving. The local axes are illustrated by the yellow transform cross-hair at the origin of the object. You can rotate around the local axes by holding down both the CONTROL and the SHIFT keys simultaneously.

\* Tutorial 5.4: Local Transformations

1. While executing the move figure command, hold down the SHIFT key
5.6 REPOSITIONING THE VIEW

2. Rotate the figure around the $y$ axis
   Do this by holding down the CONTROL key and pressing the middle mouse buttons. The figure will rotate around the yellow axis labeled $y$. If you haven't rotated the figure before, this axis will still be aligned with the vertical axis.

3. Rotate the figure around the $x$ axis
   Do this by holding down the CONTROL key and pressing the left mouse button. The figure will rotate now around the yellow axis labeled $x$.
   Compare this to ordinary rotation without holding down the SHIFT key: the ordinary rotation takes place around the global axis, which is not necessarily aligned with the global axes.

5.6 Repositioning the View

When you are rotating a figure, it is necessary to position the view so that it is not perpendicular to the axis of rotation. When this situation does arise, the rotation wheel is perpendicular to the line of sight and the mouse has no "leverage." It is necessary to reposition the view so you can see the wheel. You can do this from within the move figure command by hitting $^v$, which executes the change view command. When you terminate the change view command, you will automatically return to moving the figure as before.

TUTORIAL 5.5: REPOSITIONING THE VIEW

1. While executing the move figure command, hit $^v$
   You will begin changing the view with the change view command. Reposition the view a little.

2. When you are done, hit ESCAPE
   The prompt will change back to the move figure command and you may resume moving the figure just as before.

   Actually, you can execute any Jack command from inside the move figure command, as long as it is bound to a key.

5.7 Hints

After rotating figures away from their original orientation, it is easy to lose track of which axis is which. There is a simple way of keeping track of this information. Holding the mouse still, press down the left button while you hold down the CONTROL key. The wheel displays the $x$ axis. If that's the axis around which you want to rotate, begin rotating. If not, press down the middle button to find the $y$ axis, or the right button to find the $z$ axis. The information about which axis is which is not visually available on the screen at all times, but if you need to know it, just press down the mouse buttons in succession.

5.8 Entering Transformations from the Keyboard

As you are moving an object, you may also enter values for the transformation directly from the keyboard. To enter the value from the keyboard, hit $^k$ while moving the object. The message window will prompt you with a message like:

```
transform: trans(68.71cm,23.46cm,67.43cm)
```

You may enter the transform by editing this value. The default value is the current value of the transform: pressing ENTER without modifying the value does nothing. When a default transform is given which is not pure translation, a term involving rotation is printed first, followed by a translation term.
Jack accepts transformations in the syntax of the Peabody language. Peabody relies heavily on the specification of homogeneous transformations, and the Peabody language has a rather simple mechanism for describing such transformations. Transformations may be expressed as a sequence of simpler, primitive transformations such as rotation and translation.

The primitive transformations are concatenated in an arithmetic expression. The multiplication operator * multiplies two transformations. In addition, the numerical values of the operations can themselves be arithmetic expressions. You may also specify units for the numbers. The default units are centimeters for distance and degrees for rotation. If you omit the unit specification, these are assumed.

The translation operator is \texttt{trans}, and it takes three arguments, giving the translation in \( x \), \( y \), and \( z \). Rotation may be described with the \texttt{xyz} operator, which specifies rotation in terms of angles around the local \( x \), \( y \), and \( z \) axes, in that order. For example,

\[
\texttt{t = xyz(100deg,20deg,30deg);}
\]

describes a rotation transformation which is formed by a rotation of 10° around the \( x \) axis, followed by a rotation of 20° around the \( y \) axis, followed by a rotation of 30° around the \( z \) axis. This operator can be used for simple rotations around a single coordinate axis by using zeros for two of the angles.

\[
\texttt{xyz(\theta_x, \theta_y, \theta_z)}
\]

Figure 5.4: The \texttt{xyz} Rotation Operator

5.8.1 Multiplying Homogeneous Transformations

Peabody represents homogeneous transforms as 4 \times 4 matrices in the form:

\[
\begin{bmatrix}
  z_0 & x_1 & x_2 & 0 \\
  y_0 & y_1 & y_2 & 0 \\
  z_0 & z_1 & z_2 & 0 \\
  p_0 & p_1 & p_2 & 1
\end{bmatrix}
\]

The translation vector \((p_0, p_1, p_2)\) is in the bottom row of the matrix.

When transformations are multiplied, the product may be interpreted in two ways. The most intuitive way corresponds to local transformations when applied right to left. For example, the transformation given by

\[
\texttt{t = xyz(90deg,0,0) * trans(0,100cm,0);}
\]

may be interpreted as first a translation of 100 cm along the \( y \) axis, followed by a rotation of 90° around the \( x \) axis. The ordering is critical, since transformations don’t commute.

Alternatively, the product of transformations may be interpreted in global coordinates when read left to right. The above transformation may be interpreted as a rotation of 90° around the \( z \) axis, followed by a translation of 100 centimeters along the \( y \) axis.

When Jack writes a transformation, it writes the rotation part followed by the translation part, expressed in terms of the \texttt{xyz} operator, no matter how the expression was originally specified. This product can be
5.8. ENTERING TRANSFORMATIONS FROM THE KEYBOARD

Figure 5.5: Multiplication of homogeneous transformations (a) right to left

\[ \text{xyz}(90,0,0) \times \text{trans}(0,100,0) \]

interpreted as a rotation followed by a translation with respect to the base coordinate frame, or alternatively
a translation followed by a rotation around the \textit{translated} axes.

The exponentiation operator may be applied to transforms, with the usual meaning. Raising a transform
to the power of -1 yields the inverse of the transform.

The keyboard facility provides a means of manipulating transforms in several convenient ways:

1. Absolute transformations may be entered by deleting the default value and entering the
   complete transformation.
2. Incremental transformations may be applied by simply editing the supplied default values
   and adding the incremental value to it. This can be done by either adding the numbers in
   your head or adding them with a plus sign in the expression. Recall that the values may be
   arbitrary arithmetic expressions, not just numbers.
3. Local incremental transformations may be applied by appending a term to the end of the
   supplied current expression. For example, appending the term \textit{xyz} (1\text{deg}, 0, 0) to the end
   of the current transform expression effectively rotates the object around its \textit{local} x axis by
   1° from its current orientation.
4. Global incremental transformations may be applied by inserting a term in front of the
   supplied current expression. This can be done by hitting \textit{A} to back the cursor up to the
   beginning of the buffer. For example, inserting the term \textit{xyz} (1\text{deg}, 0, 0) before the the
   current transform expression effectively rotates the object around the \textit{x} axis of the world
   coordinate frame by 1° from its current location.
5. You can invert the transformation by raising the current transformation to the \textit{-1} power.
   The exponentiation operator is \textit{^}, and it has a higher precedence than the other arithmetic
   operators, so you should enclose the complete transformation in parentheses.

\textbf{TUT\textit{O}}R\textit{I}AL 5.6: ENTERING TRANSFORMATIONS FROM THE KEYBOARD

1. While executing the \textit{move figure} command, hit \textit{K}

   The message window will prompt you with the current location of the figure:
CHAPTER 5. MOVING FIGURES

| transform: trans(0.00cm,0.00cm,0.00cm) |

If you have translated or rotated the figure, then your numbers may vary.

2. Hit the right arrow key to move into the string

3. Replace the middle 0.00cm with 100.00cm

   The arrow keys allow you to move left and right through the string. The DELETE rubs out characters to the left. "D rubs out characters to the right. To insert characters, just type.

4. When you are done, hit ENTER

   The origin figure will move to the point (0, 100, 0).

5. Hit "K again

   This time, the default value will be (0.00cm, 100.00cm, 0.00cm). Notice that the cursor is at the beginning of the line. If you start typing, the text will be inserted into the beginning of the line.

6. Enter an x rotation of 90 degrees

   To do this, type xyz(90deg,0,0) * The string should look like:

   | transform: xyz(90deg,0,0) * trans(0.00cm,100.00cm,0.00cm) |

   Don't forget the '*'! This is what tells Jack to concatenate the transformations.

   When you hit ENTER, the figure will rotate 90° around its local x axis. The axis is local because we inserted the rotation on the left.

7. Hit "K and enter a z rotation of 90 degrees at the end of the string

   To do this, first hit "E to move the cursor to the end of the string. Then type * xyz(0,0,90deg).

   The string should look like:

   | transform: xyz(90deg,0,0) * trans(0.00cm,100.00cm,0.00cm) * xyz(0,0,90deg) |

   Don't forget the '*'!

   When you hit ENTER, the figure will rotate 90° around the global z axis. The axis is global because we inserted the rotation on the right.

5.9 Rotating with the Arrow Keys

When you are moving an object, you can rotate it by 90° at a time by hitting the arrow keys. The left and right arrow keys rotate around the global y axis, which is vertical. The up and down arrow keys rotate around either the x or z axis, depending upon which one is closest to being perpendicular to the line of sight. The behavior is such that if you are looking down the z axis, the up and down arrow keys will rotate around the z axis, causing the object to rotate towards or away from you.

5.10 Snapping to Objects

The Jack direct manipulation movement operator is convenient for gross movements, but it can be difficult to adjust things precisely in this way. Many such adjustments involve moving things to tangency and aligning objects with other objects. Jack provides a means of snapping objects to tangency and alignment with other objects during the course of a moving operation.

Snapping is a sub-option of moving; as you are moving an object, hit "S. Jack will prompt you to enter the type of thing you want to snap to. Press the right button for a pop-menu of choices.
When you snap to the orientation of something, Jack finds the coordinate axis of the thing you are moving which is closest to the orientation of the thing you want to snap to and then aligns these two axes. The orientation of the object you are moving is described by the yellow crosshair.

The options are:

**site**
*Jack* will ask you to pick a site, and it will then snap the object you are moving to both the position and orientation of the site.

**site position**
*Jack* will ask you to pick a site. When you pick a site, the object you are moving will snap to the location of the site, keeping the same orientation.

**site orientation**
*Jack* will ask you to pick a site. When you pick a site, the object you are moving will snap to the global coordinate axes of the site, with as little rotation as possible. It matches the axes depending on the current orientation of the object. Therefore, before you snap, you should rotate the object so that it is fairly close to the orientation of the site. Then the snap will clean up the orientation.

This orientation is different from that in the **site** option, in which the orientation matches that of the site exactly.

**node position**
*Jack* will ask you to pick a node. When you pick a node, the object you are moving will snap to the location of the node, keeping the same orientation.

**edge position**
*Jack* will ask you to pick an edge. When you pick an edge, the object you are moving will snap to a point on the edge, along the shortest possible path. This will either be along the path perpendicular to the edge, or to one of the endpoints of the edge. The orientation of the object will not change.

**edge line**
*Jack* will ask you to pick an edge. When you pick an edge, the object you are moving will snap to the line of the edge, along the shortest possible path, i.e. the path perpendicular to the edge. The object origin will then lie along the line defined by the edge. The object may or may not lie between the endpoints of the edge. The orientation of the object will not change.

**edge orientation**
*Jack* will ask you to pick an edge. When you pick an edge, the object you are moving will snap to the orientation of that edge. The snapping is accomplished by determining the
coordinate axis of the object you are moving which is closest in orientation to the direction of the edge and rotating it so that it becomes aligned with the edge. Therefore, before snapping to the edge, make sure that you have aligned the object in approximately the correct orientation. The snap then re-adjusts the orientation.

**face position**

*Jack* will ask you to pick a face. When you pick a face, the object you are moving will snap to it, along the shortest possible path. This may be the path perpendicular to the face, or it may be the path to an edge or node of the face, depending on the relative location. The object origin will then lie in the plane of the face within its boundary.

**face center**

*Jack* will ask you to pick a face. When you pick a face, the object you are moving will snap to the center of the face. The orientation of the object will not change.

**face plane**

*Jack* will ask you to pick a face. When you pick a face, the object you are moving will snap to the plane in which the face lies, along the shortest possible path. The object origin will then lie in the plane of the face, but it may not lie within its boundary.

**face orientation**

*Jack* will ask you to pick a face. When you pick a face, the object you are moving will snap to the orientation of that face. The snapping is accomplished by determining the coordinate axis of the object you are moving which is closest in orientation to the normal of the face and rotating it so that it becomes aligned with the edge. Therefore, before snapping to the face, make sure that you have aligned the object in approximately the correct orientation. The snap then readjusts the orientation.

**square orientation**

*Jack* will snap the object to the coordinate axes which most closely match the object's current orientation. This means that the global orientation of the object will be composed entirely of 90° rotations around coordinate axes.

**ground plane**

*Jack* will lower the object to the floor. The object will move downwards until a vertex of the object is located in the $y = 0$ plane.

---

**Tutorial 5.7: Snapping to Tangency and Alignment**

1. Make sure that you have two fairly simple objects to experiment with
   Try experimenting with the cube and the pyramid.

2. While you are executing the move figure command, hit $-$S
   *Jack* will then prompt you to specify what you want to snap to. You can snap to either the position or orientation of a site, node, edge, or face.
   You can enter the appropriate value from the keyboard, or do so from the pop-up menu by pressing the right mouse button.

3. Press the right mouse button and select the node position item from the menu

4. Pick a node on the other figure
   Click the mouse on one of the nodes on the figure you are not moving.
   After picking the node, the object will slide towards the node you have picked.

5. Experiment with the other things to snap to
5.11 Adjusting Joints

The angles at the joints of a figure define its posture. These joint angles may be manipulated in Jack with the adjust joint command. For convenience, this command is bound to ^E.

Joints connect sites on different segments within a figure. Joints in peabody may have specific degrees of freedom. A degree of freedom is a rotation around a specific axis. This rotation describes the relative orientation of the two sites which the joint connects. Joints may also be prismatic, in which case they translate along the axis. The transformation between the sites which a joint connects is formed by the product of the simple rotations and translations associated with each degree of freedom in order.

The degrees of freedom of a joint are specified in the syntax of the peabody language by the type field, whose value is an arbitrary expression composed of primitive rotation and translation operators. The rotation operator is R, and the translation operator is T. Each rotation and translation specifies an axis. The axis must be a coordinate axis of unit length: (1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), or (0,0,-1).

The current angle associated with the axis comprises the displacement of the joint and is not part of the type. A simple joint may be defined as:

```
joint left.elbow {
    connect left.upper.arm.distal to left.lower.arm.proximal;
    type = R(0.00,1.00,0.00);
    llimit = (0.00deg);
    ulimit = (159.00deg);
}
```

This joint rotates only around the z axis. The displacement field specifies that the transformation is a 45° rotation around the z axis.

Joints may have up to six degrees of freedom by multiplying primitive operators. In this case, the displacement field has the same number of arguments as the type expression has primitive elements. The complete transformation at the joint is the product of each primitive operator instantiated with the appropriate angle. As in the case of homogeneous transforms, the operators should be interpreted right to left as local transformations, i.e. with respect to the local, or current, transform. Alternatively, the displacement at the joint may interpreted from left to right as primitive transformations with respect to the parent coordinate frame, that is, the site on the "from" side of the joint.

For example, the transformation at the joint defined by:

```
joint left.wrist {
    connect left.lower.arm.distal to left.palm.base;
    type = R(0.00,0.00,1.00) * R(1.00,0.00,0.00) * R(0.00,1.00,0.00);
    llimit = (0.00deg,-47.90deg,-78.00deg);
    ulimit = (135.00deg,36.70deg,94.80deg);
    displacement = (10.00deg,20.00deg,30.00deg);
}
```

may be interpreted as a rotation of 30° around the y axis of site left.lower.arm.distal, followed by a rotation of 20° around the rotated x axis, followed again by a rotation of 10° around the rotated y axis. It may alternatively be interpreted as a rotation of 10° around the z axis of the site left.lower.arm.distal, followed by a rotation of 20° around the y axis of the same frame, followed by a rotation of 30° around the x axis of the same frame as well.

5.11.1 The Human Shoulder

JackVersion 5 has a much more sophisticated representation of the human shoulder joint than it does for the other joints in the body. Because of this, the shoulder moves in a different way, and it is not so easy to describe through specific rotational degrees of freedom. If you have been using previous versions of Jack, then you will notice some differences when you adjust the shoulder joint. The shoulder joint and how to manipulate it are described in Section 6.6.1.

For the purposes of these examples, experiment with the other joints in the human figure, such as the elbow and wrist. The definition of these joints uses only the basic features of the peabody object.
representation.

5.11.2 Manipulating Joints Interactively

Joints are manipulated in Jack using the same general direct manipulation facility, although the transformations are encoded in the mouse differently. During adjusting a joint, the mouse buttons control the transformation along successive degrees of freedom of the joint. Typically, all these degrees of freedom are rotational, but it is also possible to define prismatic joints, which translate along an axis.

The adjust joint command binds the degree of freedom axes to the mouse buttons, so that the left button controls the first degree of freedom, the middle button controls the second degree of freedom, and the right button controls the third degree of freedom. Pressing any individual button activates that degree of freedom. You cannot transform the joint along more than one axis at a time, so pressing more than one button at a time has no effect. If a joint has only one degree of freedom, pressing the middle or right button has no effect. If a joint has only two degrees of freedom, pressing the right button has no effect.

If the degree of freedom is rotational, the rotation wheel appears and you may rotate the joint along that axis. In addition, if the joint has limits, the limits are displayed as sections in the wheel. The green section defines the range within the limits; the red section defines the ranges outside the limits. The green and red sections appear relative to the cursor spoke, not the segment which the joints connects. You cannot move the mouse into the red zone.

Joints with more than three degrees of freedom are highly uncommon, but in this case, the additional degrees of freedom are bound to the three mouse buttons with the SHIFT key held down. Joints should not have more than six degrees of freedom. With joints of more than one degree of freedom, it is important to see exactly which axis is being manipulated. Therefore, it is very helpful to see the coordinate axis projections of the rotation wheel while adjusting the joint.

\* TUTORIAL 5.8: ADJUSTING A JOINT

1. Read in a human figure model and position the view close to its left elbow
2. Execute the adjust joint command
   Pick the left elbow. You then begin adjusting the joint. This joint has only one degree of freedom, so only the left mouse button is active. If you press the middle or right buttons, you will get a message

   \[No degree of freedom associated with that button!\]

3. Press the left mouse button
   The rotation wheel will appear, but this time it has sections of red and green. The green section defines the range through which you may rotate the mouse. As you rotate the mouse, the wheel rotates but the red and green sections remain fixed. When you reach the boundary, the joint stops at its limit.
   The red and green sections always appear relative to the position of the line between the cursor and the origin of the wheel. You should think of these sections as defining the legal range of movement for the mouse. If you position the mouse near the distal end of the segment which the joint moves, then the sections will illustrate the joint limits.

\* TUTORIAL 5.9: ADJUSTING A MULTIPLE DEGREE OF FREEDOM JOINT

Before doing this example, adjust the shoulder and leave it at 90°. This will make the wrist more easily visible.

1. Execute the adjust joint command
2. Select the left wrist of the human figure model
   When you press a mouse button, the definition of the joint will appear in the lower left corner of the screen.
5.11. ADJUSTING JOINTS

Figure 5.7: Adjusting the elbow joint of a human figure.

The numbers for the limits for your model may be different. The type field describes the degrees of freedom, which in this case can be interpreted as a rotation around the y axis of the arm, followed by a rotation around the local x (the x axis of the arm rotated around the y axis), followed by a rotation around the local, doubley rotated, z axis.

3. Press left mouse button
The wheel appears around the y axis of the arm, which extends horizontally outward from the wrist. The red and green sections of the wheel illustrate the legal range of motion. Notice that the displacement value changes as you move the mouse. This illustrates which degree of freedom you are rotating around. The left button controls the first degree of freedom, which is the right-most one in the DOF expression.

Before releasing the left button, rotate the first degree of freedom back to 0°.

4. Release the left button and press the middle button
This rotates around the next degree of freedom, which is the x axis, which extends upwards when the shoulder is in its default position and the elbow is bent 90°.

The axis is local, which means that it is rotated by the first degree of freedom.

5. Release the middle button and press the right button
This rotates around the third degree of freedom, which is the z axis, which extends down the length of the hand. This axis is local, too, meaning it has been rotated by the first and second degrees of freedom.

5.11.3 Joints with No Degrees of Freedom
Joints may have no degrees of freedom, which means that no axes of rotation or translation have been defined. Such joints are just arbitrary transformations between segments. When you adjust such a joint.
you can manipulate the transformation across the joint the same way as with the move figure command.

5.12 The Directionality of Joints

The direction in which joints are defined affects how they behave during manipulation. The transformation across a joint is defined in the direction of how the sites appear in the connect statement, regardless of how the figure is rooted.

5.13 Entering Joint Angles From the Keyboard

You may enter joint angles directly from the keyboard by hitting "K while adjusting the joint. You will be prompted to enter the joint angles with a message like:

```
joint angles: (73.41deg,78.91deg,23.78)
```

The default value gives the current joint angles. The number of values required depends upon the number of degrees of freedom of the joint. This is given in the joint description. The number of default values also specifies the correct number of values. If a joint has only one degree of freedom, the parentheses around the angle vector may be omitted.
Chapter 6
Manipulating Human Figures

*Jack* Version 5 has a much richer notion of human figures and how to interact them. Previous versions of *Jack* have provided certain means of manipulating and positioning figures in general, but had few special features to interacting with human figures. In *Jack* Version 5, these new commands are on the human menu.

### 6.1 Introduction

Human figures are modeled in *Jack* through peabody, just like all other types of objects, but there special ways of manipulating them and describing their motion. The commands for manipulating human figures are on the human manipulation menu and the human behavior menu. Commands for describing the motion of human figures are on the human motion menu and are described in Chapter 8.

A human figure in *Jack* is represented as a peabody figure, which is a collection of segments connected by joints. In addition, human figures are controlled through a set of kinematic constraints that make the figure behave in a certain way. The constraints are "handles" through which you can cause the figure to move while maintaining its balance and other conditions. These constraints are maintained automatically by *Jack* so they are transparent to the user. The commands on the human manipulation menu provide a means of interacting with these figures and their constraints. The commands on the human behavior menu change certain properties of the constraints.

*Jack* makes a distinction between interactive manipulation and the description of motion, although it is sometimes easy to confuse the two. As you manipulate a figure interactively by moving it or by moving some part of it, you are describing its posture. The only significant result of the manipulation process is the final posture that you achieve, not the motion that the figure goes through to achieve it. If you are concerned
principally with motion that a figure goes through, then you should describe it through the motion system, described in Chapter 8.

For example, in analyzing a reaching task, you may only be interested in the posture of the figure as its hand is touching a particular control. To do this, you can use the move arm command on the human manipulation menu to interactively drag the hand towards the control and leave it in the desire position. However, if you want to illustrate the motion which the figure goes through in moving its arm to a particular control, then you should use the create hand motion on the human motion menu. This command allows you to set up the time interval over which the reach will take place and then generate a smooth motion sequence of the moving arm.

The behavioral constraints which Jack uses to control the figures do capture some of the characteristics of how human bodies move. The philosophy of Jack is to represent the figures as much as possible as “real human figures” which behave in human-like ways and allow you interact with them in ways suggestive of how we describe actual human movement. The movement which is generated by the interactive manipulation process is meant as a quick substitute for the motion which can be described through the motion system.

\* Tutorial 6.1: Human Figures

1. Create a human figure with the command create human figure.
   This command reads the description of the figure from a peabody file, and it then creates the set of constraints which model the behavior of the figure in its natural state.

2. Position the view so your are looking at the figure's side.

3. Execute the command bend torso
   This command will prompt you to select a type of bending. You may choose to bend the torso from the waist, from the neck, or anywhere in between.

4. Accept the default value by hitting ENTER
   Jack will then display a potentiometer window which allows you to bend the torso of the figure forwards and backwards, side to side, or twist along its axis.

5. Adjust the flexion potentiometer, slowly.
   Press down the left mouse while the mouse cursor is in the left meter, and hold it down while moving the mouse up and down. The torso of the figure will bend forwards and backwards, but hips will shift so that the figure remains balanced.
   This is an example of a behavior. The figure behaves in such a way as to maintains its balance, even as you manipulate it. This is true of all types of manipulation, not just the torso manipulation. You can see the same effect to a lesser degree by rotating the shoulder so that the arm moves forward. However, since the arm is relatively light compared with the rest of the body, the hips to not have to shift very much to maintain balance.

6. To stop moving the torso, hit ESCAPE.

6.1.1 Moving Human Figures

When you move a human figure using the move figure command in Jack Version 5, its behavioral constraints change with the figure location so that the posture of the figure doesn't change. This is different from previous versions of Jack in the presence of constraints. Historically, when you moved a figure which had “hold” constraints, the goal of the constraints remained fixed.

This rule of behavior is not entirely true in the case of rotating the figure, however, in which case the balance of the figure may be disturbed. You may freely rotate the figure along the vertical y axis. In this case, it rotates around the location of its current support, which is either the left or right foot or the lower torso. The figure support is described in Section 6.4.3. However, if you rotate around the z or x axis, strange things happen. The exact behavior depends upon the settings of the behavior functions described below.
6.2 BENDING THE TORSO

The introductory tutorial demonstrated manipulating the torso of a human figure. This is a very powerful positioning tool which allows the posture of the torso to be described in terms of total bending angles in the forward/backward, lateral, and axial directions, along with a set of weighting parameters which describe how the total bending angle is distributed to the individual vertebrae. The model of the spine and the technique for manipulating it were developed by Gary Monheit.1

The command prompts you to select a type of bending. You can choose one of:

- Curl from neck
- Bend from waist
- Input parameters

When you bend the torso, the entire torso will bend, but you can control which vertebrae bend the most. The first two choices provide very common sets of parameters. With the neck curl, most of the bending will take place in the neck, while the waist will remain stiff until the figure reaches an extreme angle. The waist bend is just the opposite: the waist will bend freely while the neck remains stiff.

TUTORIAL 6.2: MOVING THE TORSO WITH A NECK CURL

To do this example, make sure you have human figure. Also, make sure it is standing up straight.

1. Execute the command `bend torso`.
2. Select the curl from neck option.
3. Adjust the flexion forwards and backwards
   Notice how the motion originates from the neck, and how the waist initially remains stiff.
   If you rotate it to the extreme, however, the neck will bend as well.
4. Adjust the lateral and axial angles with the middle and right potentiometers.
   They exhibit the same type of behavior.

---

5. To terminate, hit ESCAPE.

\* TUTORIAL 6.3: MOVING THE SPINE FROM THE WAIST

To do this example, make sure you have human figure. Also, make sure it is standing up straight.

1. Execute the command `bend torso`.
2. Select the bend from waist option.
3. Adjust the flexion forwards and backwards
   Notice how the motion originates from the waist, and how the neck initially remains stiff.
   If you rotate it to the extreme, however, the neck will bend as well.

![Figure 6.2: Bending the torso, with axial twisting.](image)

4. Adjust the lateral and axial angles with the middle and right potentiometers.
   They exhibit the same type of behavior.
5. To terminate, hit ESCAPE.

6.2.1 Moving the Torso More Generally

These two options are really just special cases of the third option, select parameters, which allows you to specify the explicit parameters of the bending. Using this mechanism, you can select a specific range of joints, possibly limiting the motion to a subset of the torso, along with the parameter joints which describe the weighting distribution for the bending. You can select these parameter joints through the spine window.

The spine window shows four parameter joints: the top joint, bottom joint, initiator joint, and resistor joint. The torso manipulation procedure operates on a set of vertebrae which can include the entire spine or only a portion of it. This is designated by the top and bottom joint. For most applications, the top joint should be the neck and the bottom joint should be the waist. This allows the entire torso to move.

The initiator joint is the joint which rotates the most. It is where the rotation is initiated. Likewise, the resistor joint is the joint which remains stiff during the manipulation. The resistance parameter is which Jack prompts you for after selecting the joint parameters gives a weight to the stiffness, in terms of a percentage.

These joints may be any joints in the spine, but there are two common selections which give distinctly different behavior. The first case has the initiator at the waist and the resistor at the neck. As you bend the figure forward in this case, it begins bending at the waist, keeping its neck straight until it reaches an
extreme angle. The other case is just the opposite: the neck is the initiator and the waist is the resistor. The motion in this case begins with a neck curl which gradually works its way down the spine to the waist.

\section*{Tutorial 6.4: Bending the Torso More Generally}

To do this example, make sure you have human figure. Also, make sure it is standing up straight.

1. Execute the command `bend torso`. 
   \textit{Jack} will display the spine window.
2. Click on the neck icon for the top joint
3. Click on the waist icon for the bottom joint
4. Click on the waist icon again for the initiator joint
5. Click on the neck icon again for the resistor joint
6. Enter a resistance of 50
7. Enter y to the question “interpolate through resting position?”
8. Adjust the flexion forwards and backwards
   Notice how the motion originates from the waist, and how the neck initially remains stiff.
9. To terminate, hit \textsc{escape}.

\subsection*{6.2.2 Torso Behavior}

The `bend torso` command allows you to actively manipulate the torso of a human figure, but the torso may also take part in \textit{passive} behaviors, in which it bends automatically as you move other parts of the figure. This is controlled through the command `set torso control` on the human behavior menu. This command allows you to choose an option from the pop-up menu:
The keep vertical torso behavior is the default behavior for a human figure. This behavior keeps the torso of the figure vertical. This means that as other parts of the figure move or change orientation, the torso will automatically adjust itself so that the head is vertical. If you select this option, the command will prompt you to choose a type of torso bending, just like the move torso command. Jack uses this type of bending to move the torso to keep it vertical.

The hold orientation torso behavior holds the torso in its current orientation. When you subsequently manipulate another part of the body, such as the legs or the pelvis, the upper chest will attempt to maintain the same global orientation. The hold orientation behavior is sensitive to twist as well and forward/backward and side to side orientation. This is different from the keep vertical behavior because, which is not sensitive to vertical twist.

The none option disables both of these behaviors, allowing the torso to move with the rest of the body. The keep vertical torso behavior is disabled automatically by the bend torso command. This is necessary, since the bend torso command specifies explicit angles for the torso. After executing the bend torso command, the torso will no longer automatically adjust itself, until you specifically instruct it to do so with the set torso control command again.

The bend torso command automatically sets the torso behavior to be hold orientation. Thus, whenever you bend the torso, it will automatically stay in the orientation into which you position it until you specify otherwise. The torso behavior is also affected by the move arm command. The keep vertical behavior is also disabled when you move an arm of the figure from the waist. The move arm command is described in Section 6.6. When you move an arm from the waist, the spine needs to be able to bend. It bends according to the parameters which it currently has, as set by set torso control. You can change the parameters for their effect with the move arm command without turning on the keep vertical behavior with the set parameters option. The move arm command does not affect the hold orientation option. With this set, the torso will attempt to maintain its orientation even as the arm and chest move.

### 6.3 Rotating the Pelvis

The lower torso region of the body is controlled through two constraints. One is a positional constraint on the center of mass, and the other is an orientational constraint on the pelvis. The pelvis constraints provides control over the orientation of the pelvic area. We use the terms pelvis and lower torso interchangeably. The command rotate pelvis allows you to interactively rotate the pelvis.

The effect of rotating the pelvis depends very much on the behavior of the torso. This can be controlled through the command set torso control on the human behavior menu. When the torso is maintaining vertical, rotating the pelvis causes the hips to curl under the body. When the torso is released, rotating the pelvis is similar to moving the torso from the waist, except that the motion is from the hips, not the waist.

#### ♠ Tutorial 6.5: Rotating the Pelvis

Make sure you have a human figure. Start with the human figure standing erect.

1. Execute the command set torso control.
   Select the none option. This will allow the torso to rotate freely, without automatically righting itself.

2. Execute the command rotate pelvis.
   If there is more than one human figure, Jack will prompt you to pick which figure you want to manipulate.

3. Press the left mouse button to rotate forwards or backwards.
   When you press down the left mouse button, the rotation wheel should appear, allowing you to rotate along an axis running through the side of the figure. As you do so, the figure’s hips should curl forwards or backwards. The figure will maintain its balance, so the hips should move forwards or backwards as well. Compare this motion to the move torso command: the movement comes from the hips.

4. Press the middle mouse button to twist vertically.
When you press down the middle mouse button, the rotation wheel should appear, allowing you to rotate along the vertical axis. As you do so, the figure’s hips should twist around to the right or left. The feet should stay planted in place on the floor.

Don’t rotate too far, or you will reach an orientation which the hips cannot achieve.

5. **Press the right mouse button to twist side to side**

When you press down the right mouse button, the rotation wheel should appear, allowing you to rotate along an axis running through the figure front to back. As you do so, the figure’s hips should tilt, and the figure’s knees should bend one way or the other. The feet should stay remain planted in place on the floor.

It is especially important not to rotate too far in this case, because there is not much mobility in this direction with the feet close together.

6. To terminate, hit **ESCAPE**.

Leave the figure in a more or less upright posture.

7. **Execute the command** set torso control again.

This time, select the keep vertical option.

8. **Execute the rotate pelvis command again.**

9. **Press the left mouse button to rotate forwards or backwards.**

As you rotate forwards, the hips will curl just like before, but the torso will automatically bend forwards or backwards so that the head remains vertical.

The orientational behavior of the pelvis can be set with the set pelvis control command, which provides these options:

<table>
<thead>
<tr>
<th>type of control:</th>
</tr>
</thead>
<tbody>
<tr>
<td>hold orientation</td>
</tr>
<tr>
<td>follow feet</td>
</tr>
<tr>
<td>none</td>
</tr>
</tbody>
</table>

The hold orientation option holds the pelvis in its current orientation. The follow feet option causes the pelvis to rotate automatically around the global vertical axis so that it maintains the same orientation.
relative to the orientation of the feet. In the simplest case, the forward direction of the pelvis is determined in terms of the average of the orientation of the two feet. Therefore, twisting the feet will automatically cause the pelvis to rotate.

### 6.4 Moving the Center of Mass

You can shift the weight of a figure between the feet with the move center of mass command. This command allows you to translate the balance point of the figure. The balance point is just the vertical projection of the figure’s center of mass onto the ground plane. Normally, human figures in Jack maintain their balance at a point halfway between the feet, although this may be controlled through the set balance control command, as described in Section 6.5.

This command does not restrict the center of mass to the support polygon of the figure, so it is possible to “unbalance” figure. If this happens, the figure does not automatically fall over. It might, however, take a step in order to maintain its balance, as described in Section 6.5.2.

When you move the center of mass with the move center of mass, Jack displays the support polygon of the figure, along with a vertical projection of the center of mass onto the ground plane. If the balance point is within the support polygon, the polygon is green, indicating balance. If the balance point is not inside the polygon, the polygons is drawn red, indicating the figure is not balanced. You can enable the display of the support polygon all the time with the command display support polygon.

Normally, Jack allows the center of mass of the figure to rise or fall as necessary as the feet move. The balance behavior actually controls only the horizontal location of the center of mass. However, the move center of mass command gives you control over the elevation of the center of mass as well. Therefore, you can translate the center of mass up or down, causing the figure to squat or stand on its toes.

**♣ Tutorial 6.6: Moving the Center of Mass**

Make sure you have a human figure, and move its feet apart with the move foot command.

1. Execute the move center of mass command.
   
   If there is more than one human figure, Jack will prompt you to pick which figure you want to manipulate.
   
   A yellow cross-hair will appear on the floor at the figure’s balance point.
   
2. Press a mouse button to translate the balance point.
You move the center of mass just like moving a figure with the move figure command, using the mouse buttons to translate and the control key to rotate. Rotating the center of mass has no effect, because the orientation of the body is controlled through the pelvis, not the center of mass. Also, translating the balance point up and down has no effect either, because the constraint on the center of mass only specifies its location in the ground plane.

3. To quit, hit **ESCAPE**.

When you quit, the center of mass remains where you left it. It will remain there until you move the feet again.

6.4.1 Controlling the Balance

Normally in *Jack*, the distribution of weight between the feet remains fixed even as the feet move. This can be controlled with the set balance control command, which provides these options:

<table>
<thead>
<tr>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>follow feet</td>
</tr>
<tr>
<td>hold current position</td>
</tr>
<tr>
<td>hold current elevation</td>
</tr>
<tr>
<td>release elevation</td>
</tr>
<tr>
<td>seated</td>
</tr>
</tbody>
</table>

The default option is follow feet. This means that as the feet move, the balance point automatically adjusts to maintain the same location relative to the feet. This is true of both lateral displacement and displacement forward and backward of the *balance line*, the line between the balls of the feet. The move center of mass command described above allows you to change the location of the balance point relative to the feet. After doing so, the center of mass will automatically adjust to maintain the same placement relative to the feet.

Another option for balance is hold current position, in which the balance point will hold its current global position, even if the feet later move. With this, it is possible to unbalance the figure, and it will not fall over. Neither of these behaviors control the elevation of the center of mass. *Jack* normally allows the center of mass to float up and down as necessary as the feet and body move. You can control this elevation behavior with the set balance control command. The option hold current elevation will cause the elevation to remain at its current value. The release elevation option allows the elevation to rise and fall as necessary, the default behavior. Note that these options are not mutually exclusive with the other balance controls.

Another option is seated. This is described more fully below.

6.4.2 Seated Figures

The behavior of a figure in a standing and seated position is quite different, particularly related to the balance parameters since balance is not an issue when a figure is sitting. *Jack* requires that you specify when you want a figure to be seated. You can do this with the seated option of the set balance control command.

The seated option does not actually make the figure sit down, but it prepares it to do so. It sets the lower torso as the root of the figure hierarchy. After setting the balance control seated, you can raise, lower, and shift the center of mass of the figure using the move center of mass.

The seated behavior also changes the way the feet are handled. When a figure is standing, its feet are controlled through the toes. When you move a foot, you move it from the toes. This suggests the way a standing person tends to direct his or her feet towards a particular place: typically, the toes go first. When a figure is seated, the heel serves as the primary reference point on the foot. After setting the balance behavior to seated, when you move a foot, you move it from the heel instead of the toes.

🔺 TUTORIAL 6.7: SEATING A FIGURE

Make sure you have a human figure.
1. Read in the file chair.pss
   This is an example chair. In this example, the chair is for looks only. You can use another object if you like.

2. Move the human figure to the front of the chair.
   Use the move figure command. Position the figure so that its feet are in the proper place in front of the chair. This is not absolutely critical because the position can be adjusted later on, either by moving the figure again or by moving the feet. However, it is best to begin with the figure's feet in approximately the correct location.

3. Execute the command set balance control.
   Choose the seated option. Nothing will change visibly, but the subsequent behavior of the figure will be different.
   You may alternatively execute this command before moving the figure.

4. Execute the command move center of mass.
   This allows you to translate the lower torso, to which the center of mass is associated. Translate the center of mass backwards, over the seat of the chair, then downwards onto the chair. The feet will remain in place on the floor. If you translate back too far, the feet may come off the floor. If this happens, move the center of mass forwards or downwards to a place where the feet can reach the floor.

5. Rotate backwards
   Executing the rotate pelvis command to rotate the figure backwards. With the torso behavior set to keep vertical, the chest will automatically bend, so the figure appears to slump into the chair.

6.4.2.1 Attaching a Figure to a Seat
   Once you have positioned the figure into a seated posture in the chair, you can attach the figure to the chair so that if the chair moves, the figure will move with it. You can do this with the attach figure command. This command is actually very general and allows you to attach any object to any other object. It will prompt you to pick a figure to attach and then pick a site to attach it to. After the attachment is made, the relative location of the attached figure and the site will remain fixed even if the site moves.
6.4. MOVING THE CENTER OF MASS

Figure 6.7: Dragging the center of mass down towards the seat.

Figure 6.8: Rotating the pelvis to relax the upper torso.
Tutorial 6.8: Attaching a Figure to a Seat

Use the human figure and chair from the previous example.

1. **Execute the command** `attach figure`
   This command is on the object menu. It will prompt you to pick a figure to be attached.

2. **Pick the human figure**
   It will then prompt you to pick a site to attach the figure to.

3. **Pick a site on the chair**
   Any site on the chair will do. There is one in end of the front left leg. After selecting the site, the human figure will be attached to the chair.

4. **Move the chair with move figure**
   The figure will remain seated in the chair, but its feet will remain fixed on the ground.

![Figure 6.9: Moving the chair while human figure is attached.](image)

5. **Execute the move center of mass command.**
   You can still move the center of mass of the figure to reposition it in the chair. This changes the placement of the figure relative to the chair, but the placement will remain fixed as you leave it.

To un-attach the figure from the chair, execute the unattach figure command.

6.4.3 The Figure Support

Previous versions of Jack used the term support to describe how the figure is connect to the world coordinate frame. In Version 5.x, the figure support is controlled automatically, so there is no longer any need to specify it explicitly. Therefore, there is no longer any need for the set figure support command. As a hierarchy, the figure is rooted through either one of the toes, or through the lower torso. Which one the system selects depends on the posture of the figure. Jack makes the decision based on two criteria:

- It roots the figure through a foot whenever the weight of the body is more than 60% on that foot. This ensures that if the figure is standing with more weight on one leg than the other, the supporting leg serves as the root. It also ensures that if the figure is standing with weight equally between the two legs but possibly swaying side to side that the root doesn’t rapidly change between the legs.
6.5. MOVING THE FEET

If the height of the center of mass above the feet dips below 70% of the length of the leg, then theoot changes to the lower torso. This predicts that the figure is sitting down. Heuristically, this proves
to be a good choice even if the figure is only squatting, because the constraint on the non-support leg
tends to behave badly when both knees are bent to their extremes.

This is all handled internally by Jack, so there is no need for you to worry about it as a user.

6.5 Moving the Feet

You can move the legs of a human figure with the command `move foot`. This command asks you to select the
left or right foot, and it then lets you move it, subject to the balance constraint, just as with the pelvis and
torso.

6.5.1 The `move foot` Command

🐶 TUTORIAL 6.9: MOVING THE FEET

Make sure you have a human figure.

1. Execute the command `move foot`.
   
   If there is more than one human figure, Jack will prompt you to pick which figure you want
to manipulate.
   
   This command will then ask you to select the left or right foot. It will display a message:

   ```
   side (left/right): left
   ```

   Press the right mouse button and select `right` or `left` from the pop-up menu.
   
   When you make the selection, a yellow cross-hair will appear at the toes of the corresponding
foot.

2. Press a mouse button to translate or rotate the toes
   
   You move the toes just like moving a figure with the `move figure` command. The leg comes
along with it.

3. Move the foot forward of the body
   
   Notice that the entire figure moves. The balance point of the figure floats halfway between
the feet, so as you move one of the feet, the balance point moves.

4. To terminate, hit ESCAPE.
   
   When you quit moving, the foot stays where it you left it.

5. Execute the command `set balance control`.

   Select the `follow feet` option. This will prompt you to move the center of mass just like the
`move center of mass` command.

6. Translate the balance point to the left foot
   
   Move it until the balance point is close to the left foot, so that the figure is balanced 80%
on the left foot and 20% on the right.

7. Translate the balance point forward
   
   Watch the support polygon as you move, and move it until it is near the front edge of the
support polygon.

8. Hit ESCAPE to stop moving the center of mass.

9. Execute the `move foot` command again.

   As you move the foot, the figure will maintain the same proportion of support between the
feet, both side to side and front to back.
10. **Execute the command** set balance control.
   This time, select the hold current position option. This will cause the figure to keep its balance point at its current location, regardless of what happens to the feet.

11. **Execute the move foot command again.**
    Select the right foot and move it forward. Notice that the center of mass does not shift.
    Leave the foot forward of the body.

12. **Execute the move foot command once again.**
    This time, select the left foot and move it forward. Notice the feet come out from under the body but the figure does not fall down. *Jack* assumes that you know what you are doing and that you have some reason for doing this.
6.5.2 Stepping Behaviors

*Jack* Version 5 has some predictive behaviors which are designed to automatically position the feet when the figure is close to losing its balance. There are special cases of the more general notion of active behaviors. They are controlled through the set stepping behavior command, which provides these options:

<table>
<thead>
<tr>
<th>type of control:</th>
</tr>
</thead>
<tbody>
<tr>
<td>take step before losing balance</td>
</tr>
<tr>
<td>take step when pelvis twisted</td>
</tr>
<tr>
<td>off</td>
</tr>
</tbody>
</table>

Selecting either of the stepping options enables that type of behavior. Selecting the off option disables them both.

The take step before losing balance behavior is active while moving the center of mass. When this is on, moving the center of mass outside of the support polygon will cause the figure to take a step to regain its balance. The heuristics it uses are:

- If one foot is significantly in front of the other, then:
  - if the center of mass is forward, then step with the back foot,
  - if the center of mass is backward, then step with the front foot.
  
- If the feet are in line with each other, then step with the foot which bears the least weight.

- If the center of mass is to the side of the support polygon, then the weight all on one foot, so:
  - if the non-support leg is fully extended, step with it.
  - otherwise, step with the support leg. This is like a hop.

The placement of the stepping foot is always such that the new location for the feet has the center of mass in the middle of the support polygon.

The take step when pelvis twisted behavior is active while rotating the pelvis. When this is on, if the pelvis becomes very twisted relative to the feet, the figure will automatically make a stepping turn, as if turning around. This behavior measures the orientation of the pelvis relative to the feet by projecting the forward axis of the pelvis onto the ground plane and finding the angle between this axis and the axis down the center of feet, from heel to toe. The angle is always measured counter-clockwise, looking down at the feet. Therefore, a negative angle for the left foot means that the pelvis is pointing farther to the left than the left toes, and likewise a positive angle for the right foot means that the pelvis is pointing farther to the right than the right toes.

The heuristics which the pelvis stepping behavior uses are:

- If the angle between the pelvis and the left foot is greater than $45^\circ$, then swing the left foot forward.
- If the angle between the pelvis and the right foot is greater than $45^\circ$, then swing the left foot forward.
- Otherwise, if the angle between the pelvis and the left foot is less than $-20^\circ$, then step with the left.
- Otherwise, if the angle between the pelvis and the right foot is less than $-20^\circ$, then step with the right.

Both of these behaviors can be turned on at the same time.
6.5.3 Controlling the Feet

The feet controlled through constraints on the toes and heels. The constraints on the toes can either hold the feet in a fixed orientation or they can be allowed to pivot on the floor. This can be controlled with the command set foot control. This command prompts you to select which foot you want to control. You can choose the left, right, or both. It then prompts you to enter a type of control. You can choose the type by selecting an entry from the pop-up menu with the right mouse button:

- release
- pivot
- hold global location
- hold local location

If you select the pivot type, the foot will be allowed to rotate along a vertical axis through the ball of the foot. It may rotate any time the figure moves, such as when you move the other foot.

When you move a foot with the move foot command, you move its complete orientation, so you have control over its twist as well. So if a foot is allowed to pivot and you subsequently move it, you can move not only its position by its orientation. When you finish moving it, its behavior will be set to hold global location.

The hold global location option is the default foot behavior. The toes remain fixed in position and orientation.

The hold local location option fixes the location of the toes relative to a segment, which the command prompts you to select. If you subsequently move the segment, the foot automatically moves as well.

The constraint on the heels is a point-to-plane constraint which controls the elevation of the heel. Normally, the heel is constrained to the floor.

6.6 Moving the Arms

You can move the arms of a figure with the move arm command. This command is similar to the interactive reach command. It lets you interactively move a desired point for the hand around in space. This command can move either the arm fixed at the shoulder or the entire arm and torso, fixed at the waist.

6.6.1 The New Shoulder

Jack Version 5 has a much more sophisticated representation for the shoulder than previous versions. In Version 5, the shoulder and clavicle joints are coupled together so that the two move in unison in a biomechanically reasonable way. Since the joints are coupled, they cannot be adjusted independently. Whenever you adjust either the shoulder or the clavicle using the adjust joint command, you adjust the shoulder-clavicle complex as a unit. It is best to think of this shoulder-clavicle complex in Jack 5 as a single joint, although technically there are two.

The Version 5 shoulder also has true spherical joint limits, as opposed to the upper and lower limits on each degree of freedom used by earlier versions of Jack. This greatly improves the realism because the limits to the range of movement more closely match the actual human shoulder.

The new shoulder model requires a special figure definition, but when Jack 5 reads in an old human figure model, it automatically converts it to the new representation. When this happens, it is best to write the modified definition of the figure to a new file.

♣ Tutorial 6.10: Adjusting the Shoulder

Make sure you have a human figure model, and position the view so that you are looking down on the left shoulder, from the front of the figure. Also, adjust the elbow of the figure so that it is rotated 90°. This will help to illustrate the angle of the shoulder.

1. Execute the adjust joint command.

Pick the left shoulder. The three mouse buttons allow you to rotate the shoulder joint using the rotation wheels just like any other joint, although they behave slightly different.
2. **Press the left button**

The left button controls elevation. This is the angle the arm makes with the vertical axis of the body. Raise the arm about 90°.

![Figure 6.12: Adjusting the shoulder via adjust joint.](image)

Note that the shoulder and clavicle both move.

3. **Press the middle button**

The middle button controls the longitudinal angle of the arm. This always rotates around the vertical axis of the body, no matter what the elevation. If you do this with the arm by the side, then the arm twists because this is a singular position.

It is not possible rotate the shoulder forwards from its downward position with a single rotation. You must first raise the elevation and then change the longitude. Of course, to do this you should really use the move arm command anyway.

4. **Press the right button.**

The right button controls twist. The arm will twist along its axis.

### 6.6.2 The move arm Command

This command first prompts you to choose between the left and right arms. It then asks you to pick a reference point:

<table>
<thead>
<tr>
<th>reference point:</th>
</tr>
</thead>
<tbody>
<tr>
<td>forearm</td>
</tr>
<tr>
<td>palm</td>
</tr>
<tr>
<td>attached object</td>
</tr>
</tbody>
</table>

You can move the arm from the palm, from the forearm, or from an object attached to the hand. If you select the forearm option, the wrist will not move. This is the best way to move the arm, since it ensures that the wrist will not get kinked.

The command then asks you for the fixed joint. This can be either the shoulder or the waist. You specify this information through the pop-up menus, not by pointing at the figure. The reference point in the hand is in the center palm. You can change both the position and orientation of the hand as you move.
TUTORIAL 6.11: MOVING THE ARMS

Make sure you have a human figure.

1. Execute the move arm.
   If there is more than one human figure, Jack will prompt you to pick which figure you want to manipulate.
   Jack will then prompt you to choose either the left or right arm.

2. Choose either left or right.
   Jack will then prompt you to choose the reference point. You do this through the pop-up menu or the keyboard, not by pointing at the arm. The choices are:

<table>
<thead>
<tr>
<th>reference point:</th>
</tr>
</thead>
<tbody>
<tr>
<td>forearm</td>
</tr>
<tr>
<td>palm</td>
</tr>
<tr>
<td>attached object</td>
</tr>
</tbody>
</table>

3. Choose the forearm option.
   Jack will then prompt you to choose the fixed joint. You do this through the pop-up menu or the keyboard, not by pointing at the joint. The choices are:

<table>
<thead>
<tr>
<th>starting joint:</th>
</tr>
</thead>
<tbody>
<tr>
<td>shoulder</td>
</tr>
<tr>
<td>waist</td>
</tr>
</tbody>
</table>

4. Choose the shoulder.
   A yellow cross-hair will appear at the figure's hand.

5. Move the desired position of the hand with the mouse buttons.
   You can move the goal around just like with the move figure command. As you translate, notice that the hand attempts to keep the same orientation. You can explicitly change the hand's orientation by rotating it.

6.6.3 Controlling the Hands

Normally in Jack, the joints of the arms do not move unless you explicitly move them. Since the arms stay stiff, they move along with the rest of the body. For example, when you bend the torso forward, the arms appear to bend backward, only because the joints in the arm don't move. Sometimes it is convenient to have the hands positioned automatically.

You can attach the hands of the figure to certain reference points with the set hand control command. This command lets you specify a type of control for the left or right hand, or both hands together. It will ask you to choose an option from the following:

<table>
<thead>
<tr>
<th>type of control:</th>
</tr>
</thead>
<tbody>
<tr>
<td>hips</td>
</tr>
<tr>
<td>knees</td>
</tr>
<tr>
<td>hold global location</td>
</tr>
<tr>
<td>hold local location</td>
</tr>
<tr>
<td>site</td>
</tr>
<tr>
<td>release</td>
</tr>
</tbody>
</table>

The hips option places the hands on the hips as one would do when standing. The knees option places the hands on the knees as one would do when sitting and leaning slightly forward. It's not a good idea to select this option unless the figure is already in a sitting position. Otherwise, the hands won't be able to
reach the knees. The hold global location option holds the hand in its current place in space relative to the world coordinate system. The hold local location option holds the hand in its current place relative to another segment. It will prompt you to pick the segment. If that segment later moves, the arm will move so as to maintain the same relative relationship to the segment. The release option frees the hand from any previous control.

The hips and knees options prompts you to choose a reference point, either the forearm or the palm. The hold options also allow you to select an attached object reference point. This is described in the next section.

The site option attaches the hand to a site. The reference point on the hand is in the palm. The site you select must have the proper orientation, which matches the orientation of the site in the palm of the hand. This orientation is with the +z extending outward from the palm, the +y axis extending outward from the fingers, and the +x axis extending to the right of the hand as you look at the palm side. This type of control can only be used with the palm.

### 6.6.4 Attaching Objects to the Hands

You can attach objects to the figures hands with the attach figure command, just like you attached the figure to the chair in Section 6.4.2.1. The command works in the same way: pick the figure that you want attached and the pick the site you want it attached to. In the case of the hands, the attachment site will be in the palm.

#### Tutorial 6.12: Basic Carpentry Skills, Part I

1. **Read in the file hammer.fig**
   
   This file is on the Jack distribution tape in the tools subdirectory.

2. **Move the right arm in front of the body**
   
   Use the move arm command.

3. **Move the hammer with move figure**
   
   Move it so that the handle is in the proper location relative to the hand, i.e. so the hand is holding the handle.

4. **Execute the attach figure command**

![Figure 6.13: Attaching a hammer to the left hand.](image)
Pick the hammer, and then pick the palm site as the attachment site. The hammer is now attached to the hand.

5. Adjust the elbow joint
The hammer will move with the hand.

6. Move the hammer
You can still move the hammer relative to the hand, but it stays attached.

Note how this relationship is different from attaching the hand to an object using set hand control. When the hand is attached to an object using set hand control, moving the object drags the hand with it. Moving the arm, on the other hand, destroys the connection between the two.

6.6.5 Reaching with Attached Objects

When an object is attached to the hand, you can move the hand with the move arm command in two ways. You can either move the arm from the palm or from a reference point on the attached figure. This allows you to position the arm and the object simultaneously.

* TUTORIAL 6.13: BASIC CARPENTRY SKILLS, PART II

Continue with the human figure holding the hammer as before.

1. Execute the move arm command
When it prompts you for the reference point, select the attached figure option.

2. Pick the site on the head of the hammer
You will begin moving the hammer from its head, and the hand will follow

Figure 6.14: Moving the arm using the attached hammer as a reference.

6.6.6 Reaching in General

There are three ways in Jack to perform a "reaching" operation. The first way is to use the move arm command to drag the hand into position interactively. You can then use the set hand control command to keep it there as the rest of the body moves and repositions itself. The second way is to create an arm motion using the create arm motion command described in Section 8.3.6. This command will prompt you to move the hand into its final place much like move arm, but it will then let you incorporate this into an animation and
play back the motion. Finally, the reach for site command allows you to select a reference site, and it moves the hand to the site gradually in a reaching movement. Once the hand gets there, it says there as if you had issued a set hand control command with the site option.

The reach for site command currently positions the palm of the hand only, as opposed to the fingertips. The orientation of the site is significant: the hand will approach the site so that the site in the palm of the hand will match its orientation. This orientation has the $+z$ axis extending out of the palm of the hand, with the $+y$ axis extending out of the fingers, and the $+x$ axis on the right hand and the $-x$ axis on the left hand extending toward the thumb side of the hand.

The reach for site actually operates just like the create arm motion command. It creates a motion which lasts for a specific duration. However, when the movement is complete, the motion disappears. This means that the motion cannot be recorded and cannot be repeated automatically. Therefore, the reach for site command should be used mostly for positioning.

The command create hand site creates a site which can be used for a reach site. It creates it in the current location and orientation of the hand. It will prompt you for a segment on which to create the site.

### 6.6.7 Articulated Hands with Fingers

The human figure has been augmented to include fully articulated hands. Commands for controlling the hand are on the human $\Rightarrow$ hand menu. The fingers on the hand can be positioned for grasping, and a grasp can be executed. The grasping routine works by positioning the fingers for a particular grasp, positioning the hand (via the palmcenter site) near the segment to grasp, then incrementally closing the fingers and monitoring when they collide with the segment you are trying to grasp. When all fingers have collided, the grasping routine stops. The hand then appears to be grasping the segment. You can then execute a set hand control to keep the hand in that relative position to the segment. If the segment then moves, the hand will go along with it. The commands for controlling the hand with fingers are:

- **shape hand**: This command allows you to initialize the joint angles of the fingers for a grasp. The possible grasps are:
  - **power**: Like holding a baseball bat or flashlight. The fingers are extended fully outwards.
  - **precision**: Like holding a pen. The thumb, index, and middle finger are extended, and the third and fourth fingers are curled inwards.
  - **disk**: Like opening a jar. Again, the fingers are positioned as in the power grasp.
  - **tripod**: Similar to the precision grip, but thumb, first and second finger are fully extended.
  - **small disk**: Like holding a bottle cap with your finger tips. Fingers are initialized as in the disk grip.

- **reach and shape hand**: This command shapes the hand as in shape hand, and also prompts you for a segment to grasp. Once the segment is chosen, the command prompts for a grasp location type, which can be either a predefined site or a calculated transform. For precise hand positioning, it is best to define grasping sites on the object you wish to grasp. These sites should be positioned so the palmcenter lines up correctly for the desired hand position and orientation (you create these sites using the create hand site command). The calculated transform attempts to determine the shape of the object and line the hand up accordingly. This works well with simple segments, but not with more complex ones. Finally, a reach is executed to position the hand in the right position, in preparation for closing the fingers.

- **close hand**: This command prompts for a grasp type, and a segment to grasp. It then closes the fingers until they collide with the segment, or the palm.

- **grasp**: This command combines the individual steps of the previous 3 commands. That is, it will (1) shape the hand for a particular grasp, (2) move the hand to the segment to grasp, and then (3) close the fingers on the object.
- **hand collision options**: You may set the type of collision detection used when the fingers are closed on an object. They are:
  - **full collision - hand with object**: uses full geometry of the object for determining collision.
  - **bounding box collision - hand with object**: uses only the bounding box of the object for determining collision.
  - **bounding box collision - hand with object and self**: same as above, but also checks the hand for collision with itself (i.e. the fingers entering the palm, or one another).

- **check full collision**: This command allows you to just check if two segments collide (using their complete geometry).
- **check bbox collision**: Same as above, but using only the bounding boxes of the two segments. These commands are useful if you are positioning the fingers manually, and want to make sure the segments are not touching.

### 6.7 Moving the Head and Eyes

You can position the head of a human figure with the move head command. This allows you to position the head by moving a reference point which the head looks at. You can also position the eyes of the figure using the move eyes. However, these manipulation procedures function independently. The eyes are positioned relative to the current orientation of the head. Moving the eyes rotates the eyeballs in their sockets by allowing you to move a focus point, similar to the head movement. The head itself doesn’t move. You should orient the head first and then the eyes.

The head moves by rotating the neck joint. The neck, however, is a part of the spine and it moves along with the vertebrae in commands like bend torso and with the keep vertical torso behavior. When you want to control the torso and the head at the same time, *Jack* automatically adjusts the torso parameters to ignore the neck.

You can alternatively move the head of a human figure by simply adjusting the neck joint with the adjust joint command, but this is not as intuitive because it is not always easy to see what direction the figure is looking. Also, when the neck is coupled with the rest of the joints in the spine, you cannot adjust it independently. If you try to do so, *Jack* will issue a message:

```
That joint is coupled: can't adjust it interactively!
```

If you do indeed want to adjust it, then set the torso parameters so that the torso control does not include the neck joint.

The move head command attempts to keep the head oriented vertically, so it is not possible to cause the head to twist along its line of sight.

🌟 **Tutorial 6.14: Moving the Head**

Make sure you have a human figure.

1. **Execute the move head.**
   - If there is more than one human figure, *Jack* will prompt you to pick which figure you want to manipulate.
   - A yellow cross-hair will appear in front of the figure’s head, and *Jack* will draw the line of sight of the figure, between the reference point and another reference point between the figure’s eyes.

2. **Press the mouse buttons to translate the focus point.**
   - As you move, the neck will automatically rotate so that the head looks at the focus point.
   - If you move the reference point to an extreme position so that the figure cannot rotate the head to it, it will rotate only as far as the neck joint limits allow.
3. Hit ‘S’ to snap the reference point to a particular object.

You can use the snapping mechanism to have the figure look at a particular object.

The move head command does not adjust orientation of the eyes relative to the head, but this may be done with the move eyes command. This command works in a similar fashion, by moving a reference point and having the eyes focus on it.

6.7.1 Fixating the Head

You can fixate the figure’s line of sight on an object with the command set head control. This command will prompt you to choose a type of control:

- **fixate**
- **release**

The fixate option prompts you to select a “goal,” which is the thing you want the figure to look at. It can be a site, a node, or a global transform. The figure’s line of sight will remain directed at the object even if the figure or the object move. The release option disables the fixation. The fixation is also disabled by moving the head with move head.

6.8 Postures

Jack allows you to save and restore postures, and associate postures with figures. The commands for doing so are on the utility ⇒ posture menu:

- **posture**
- **save environment posture**
- **set environment posture**
- **save figure posture**
- **write environment with postures**
The `save environment posture` command records all figure locations and joint angles. It prompts you to enter a name for the posture, which can be used as an argument for the `set environment posture` command to restore the posture later. *Jack* automatically saves the posture of the figures whenever you read a Peabody environment file or a figure file, and it names the posture according to the name of the file. These two commands actually create a set of channels, which hold the joint angles and figure positions for all figures in the environment (at the time the command is executed). These commands are equivalent to the old *Jack* commands `save posture` and `restore posture`.

The command `save figure posture` prompts for a figure, and then a posture file name. It will create a posture file and associate it with the posture name you provide. The posture file will contain all joint angles, the figure root and position, and if the figure is human, all the associated behaviors. It will also store all constraints that are currently defined on the figure (for posturing non-human figures with constraints). The command `set figure posture` will prompt for a figure, then prompt for a posture name (one from the list of currently defined postures for that figure). It will set the posture of the figure by actually reading and interpreting the posture definition file.

The command `write environment with postures` is similar to the command `write environment`, except it will add the correct posture references to all figure blocks at the end of the environment file. For the format of the posture reference, see Section A.3.10.

**Note:** When you read an environment file that has posture references, the respective posture definition file is **not** read at that time. This is important, since the posture definitions may reference things that are not currently defined in the environment at the time the posture reference is read.
Chapter 7

Specifying and Analyzing Human Figures

7.1 Specifying Anthropometry

The human figure created by the create human figure command has the dimension of the 50th percentile male human figure based on data from the Society of Automotive Engineers. These dimensions, however, simply provide a starting point for a human figure of whatever dimensions you require. The commands on the anthropology menu allow you to change the anthropometric dimensions of the figure to arbitrary values.

Currently, the anthropology commands in Jack allow you to specify dimensions in joint-center to joint-center distances. It is not possible to directly enter external dimensions from anthropology literature.

Each of the commands on the anthropology menu prompt you to enter a set of dimensions, which are measurements of length, width, height, or breadth. The prompt given with each command describes what the particular dimensions are. The current dimensions of the figure are provided as the default value. One of the dimensions is always the dominant one, and its name is capitalized in the prompt. To change the dimensions, enter the new dimensions vector. You may alternatively enter a single value which will be interpreted as the dominant dimension. The other dimensions of the body part will be scaled similarly so that the segment remains the same relative shape.

Each of these commands modifies the definition of the figure, so the figure must be written to a new figure file. The write environment command will automatically prompt you to write out the figure definition if its dimensions have been changed.

7.1.1 Foot Dimensions

The set foot dimensions command sets the dimensions of the feet of a human figure. The dimensions are the width, the height from the floor to the ankle, the length from the heel to the toe. The default values provided are the current dimensions. This command sets the dimensions both feet.

7.1.2 Leg Dimensions

The set lower leg dimensions sets the dimensions of the lower legs of a human figure. The dimensions are the thickness front to back, the width side to side, and the length from the knee to the ankle. The default values provided are the current dimensions. This command sets the dimensions both legs.

The set upper leg dimensions command sets the dimensions of the upper legs of a human figure. The dimensions are the thickness front to back, the width side to side, and the length from the hip to the knee. The default values provided are the current dimensions. This command sets the dimensions both legs.
7.1.3 Pelvic Dimensions

The set pelvic dimensions command sets the dimensions of the pelvis, or lower torso, of a human figure. The dimensions are the thickness front to back, the width side to side between the pivot points of the hips, and the height from the hip joint to the waist. The default values provided are the current dimensions.

7.1.4 Torso Dimensions

The set torso dimensions command sets the dimensions of the torso of a human figure. The dimensions are the chest thickness front to back, the width side to side between the pivot points of the shoulders, and the height from the waist to the shoulders. The default values provided are the current dimensions.

7.1.5 Arm Dimensions

The set lower arm dimensions command sets the dimensions of the lower arms of a human figure. The dimensions are the thickness front to back, the width side to side, and the length from the elbow to the wrist. The default values provided are the current dimensions. This command sets the dimensions both arms.

The set upper arm dimensions sets the dimensions of the upper arms of a human figure. The dimensions are the thickness front to back, the width side to side, and the length from the shoulder to the elbow. The default values provided are the current dimensions. This command sets the dimensions both arms.

7.1.6 Hand Dimensions

The set hand dimensions command sets the dimensions of the hands of a human figure. The dimensions are the thickness through the palm, the width including the thumb, and the length from the wrist to the figure tips. The default values provided are the current dimensions. This command sets the dimensions both hands.

7.1.7 Head Dimensions

The set head dimensions command sets the dimensions of the head of a human figure. The dimensions are the thickness back to front, the width side to side, and the height from the neck to the top of the head. The default values provided are the current dimensions.

7.1.8 Neck Dimensions

The set neck dimensions command sets the dimensions of the neck of a human figure. The dimensions are the thickness front to back, the width side to side, and the length from the height of the shoulder joint to the bottom of the head. The default values provided are the current dimensions.

7.1.9 Eye Coordinates

The command set eye coordinates sets the coordinates of a figure's eyes relative to the origin of the head. The coordinates are given in terms of the height from the neck joint, the distance forward from the body center plane, and the interpupillary distance. This command sets the location of both eyes.

7.2 Measuring Dimensions

You can measure dimensions of a human figure, or any figure for that matter, with the measure distance command on the info menu menu. This command lets you stretch a measuring line through space to measure distances between points. It will first prompt you to position the first endpoint of the line. Once this is done, it will hold the first endpoint there and then allow you to move the other end of the line to another point in space as it prints the distance between the two points. The distance is measured as the euclidean distance between the points as well as the distances in the x, y, and z axes. This mechanism is most useful in conjunction with the snapping option described in Section 5.10.
1. **Execute the **measure distance command.**

A yellow cross-hair will appear at the origin. You can move it just like any object. In particular, you can snap it to objects.

2. **Hit -S**

*Jack* will prompt you with the snap menu:

```
<table>
<thead>
<tr>
<th>snap to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
</tr>
<tr>
<td>site position</td>
</tr>
<tr>
<td>site orientation</td>
</tr>
<tr>
<td>node position</td>
</tr>
<tr>
<td>edge position</td>
</tr>
<tr>
<td>edge line</td>
</tr>
<tr>
<td>edge orientation</td>
</tr>
<tr>
<td>face position</td>
</tr>
<tr>
<td>face center</td>
</tr>
<tr>
<td>face plane</td>
</tr>
<tr>
<td>face orientation</td>
</tr>
<tr>
<td>square orientation</td>
</tr>
<tr>
<td>square orientation</td>
</tr>
</tbody>
</table>
```

3. **Choose the node position option**

Then select a node on the outside of the left upper arm. The cross-hair will snap there.

4. **Hit ESCAPE**

Another cross-hair will appear near the first one, with a line between the two. You can now move the second one. A message in the upper left corner of the screen gives the dimensions of difference between the points. The distance is the straight-line distance between the two. The vector of dimensions measures the difference along the global $x$, $y$, and $z$ axes.

![Figure 7.1: Measuring a dimension via the command measure distance.](image)
5. Hit "s again.

This time, select a node on the outside of the upper right arm. The second cross-hair will snap there and you may then read off the distance between the external surface of the arms.

### 7.2.1 Rulers

Several commands in Jack allow the user to create rulers that continuously display the linear distance between points in the environment, even when the points are moving. Rulers are drawn as a white and red line between the points. The distance is displayed in the center of the ruler, using the current distance units. Two commands are available for ruler creation:

- create ruler will prompt the user to select a segment, then create a transform at the base site of the selected segment and allow the user to move this reference point. This procedure will then be repeated for the second segment. The ruler will be created between these two reference points.

- create ruler between sites is a shortcut to the above procedure. It allows the user to create a ruler between two pre-defined sites.

Rulers are deleted with the delete ruler command.

Ruler “ticks” are represented by red and white line segments. By default, a tick is displayed every 10 units. This can be changed with the set ruler parameters command. Distances are reported using the current distance units defined in the environment; these can be changed with the set distance units command.

### 7.3 Human Body Measurements

There are several commands for specifically measuring a human figure. They are in the human menu, then the measurements menu. Note that when you request a measurement of a figure, the figure is repositioned, so use save posture and reset posture when using these commands.

A list of the dimensions considered follows. Each measurement command is listed, followed by the description of the measurement taken.

- measure standing height - Distance from floor to top of head
- measure eye height - Distance from floor to level of inner core of right eye
- measure shoulder height - Distance from floor to outer point of right shoulder
- measure kneecap height standing - Distance from floor to top of right kneecap
- measure kneecap height sitting - Distance from floor to top of right kneecap
- measure crotch height - Distance from floor to crotch
- measure sitting height - Distance from sitting surface to top of head
- eye measure sitting eye height - Distance from sitting surface to level of inner core of right
- measure popliteal height - Distance from surface of footrest to underside of right knee
- measure thigh clearance height - Distance from top of sitting surface to junction of abdomen and thigh
- measure hip breadth standing - Maximum breadth across the hips
- measure shoulder breadth - Maximum breadth across the shoulders
- measure interpupillary breadth - Distance between the center of the pupils of the eyes
- measure foot breadth - Maximum breadth of right foot
- measure hand breadth - Maximum breadth across base of fingers
• measure hip breadth sitting - Maximum breadth across the hips
• measure interscy breadth - Distance along surface of back between armpit creases
• measure shoulder circumference - Maximum circumference of shoulder
• measure chest circumference - Average circumference of chest during normal breathing
• measure waist circumference - Circumference at level of umbilicus with abdomen relaxed
• measure hip circumference - Maximum circumference of hips at level of greatest buttock protrusion
• measure calf circumference - Maximum circumference of right calf
• measure neck circumference - Maximum circumference of neck
• measure vertical trunk circumference - Circumference of trunk, buttock-shoulder-crotch
• measure biceps circumference - Maximum circumference of biceps
• measure thigh circumference - Maximum circumference of thigh
• measure head circumference - Maximum circumference of head
• measure forearm circumference - Maximum circumference of forearm
• measure wrist circumference - Circumference at wrist
• measure shoulder-elbow length - Distance from outer point of shoulder to elbow
• measure forearm-hand length - Distance from elbow to tip of middle finger
• kneecap measure buttock-knee length - Distance from rear-most buttock projection to front of right
• measure buttock-popliteal length - Distance from rear-most buttocks projection to back of right knee
• measure hand length - Distance from wrist to tip of middle finger
• measure foot length - Distance from back of right heel to tip of longest toe
• measure shoulder length - Distance from outer point of neck to outer point of shoulder
• measure palm length - Distance from wrist to base of middle finger
• measure head length - Maximum thickness of head
• measure anterior arm reach - Distance from back of shoulder to the tip of middle finger
• measure functional reach - Distance from back of shoulder to tip of thumb
• measure overhead reach - Arm extended above shoulder; distance from floor to tip of middle finger
• measure chest depth - Maximum thickness of chest
• measure arm span - Arms extended out of shoulder; distance between tips of middle fingers
• measure hand circumference - Maximum circumference of hand
7.4 Managing Human Figure Files

The command `create human figure` creates a figure of default dimensions. If the dimensions of the figure are of special importance to you, you should begin your work with `Jack` by modifying this figure with the anthropometry commands to yield a figure with your own dimensional data. Once you have done so, you should write the figure to a new file and “install” it for future use.

Human figures are represented in `Jack` through `peabody` and are stored in `peabody` figure files. The `create human figure` command reads the file `human.fig`, which is provided on the `Jack` distribution tape. This file must be properly installed in order for the `create human figure` command to work properly.

Once you have used the anthropometry commands to give specific dimensions to the figure, you must write the figure to a new figure file. You can do this in two ways. You can use the command `write figure definition` command on the write menu. This will prompt you to select a figure and then enter a file name in which to store it. Alternatively, you may just wait until you save your work with the `write environment` command. This command will detect that the human figure has been modified from its original definition and automatically prompt you to write the figure definition to a file, just like with the `write figure definition` command.

You should not write the figure to the file `human.fig` because this will change the dimensions of future figures which you create with the `create human figure` command. Instead, you should write the figure to a file with a descriptive name, probably one which matches the data for the dimensions you have applied to the figure. For example, if your data is NASA 50% percentile astronaut population data, you may choose the file name `nasa50.fig`. Or if your data is from a specific individual, you may with to name the file after the individual, such as `norman.fig`.

It is also important where you write the file, i.e. in what directory. If you intend to use this human figure model only for the current project, you can write it out to the current directory. From then on, if you want to use this human figure model, you must go to the directory in which the model is and run `Jack` from there.

If you intend to use this particular human figure model on several projects, you should choose a centrally located directory and place the file there. You should then `install` the file. You can either use the command `install file` interactively in `Jack` or you may just add a line to your `/jacks5.install` file giving the complete path name of the figure file. This will make it so that you can read in the file from any directory on your system.

7.4.1 Human Figure Psurf Files

The human figure definition has two parts: the figure definition in the `peabody` figure file and the geometric description of the individual segments in the psurf files which accompany the figure file. Psurf files may be stored in two ways. They may be stored as ordinary individual files or they may be stored collectively in an archive file, which has the suffix `.a`. Archive files are created by the Unix utility `ar`.

Figure files refers to an archive in which `Jack` will look for the individual psurf files. However, `Jack` always looks for the individual psurf files for the figure in the directory of the figure file before checking the archive. The rules for where `Jack` looks for files are described in detail in Section 9.3.5.

The archive file has the advantage of organization: all psurfs are stored in one file. However, it takes `Jack` considerably longer to read psurf files from an archive than from individual files not in an archive. Therefore, the best way to organize figure files is to place them in the same directory as the psurf files and then “install” them in `Jack`. This process is described in Section 9.3. You can either use the command `install file` interactively in `Jack` or you may just add a line to your `/jacks5.install` file giving the complete path name of the figure file.

The `Jack` distribution tape comes with a directory `$JACK/jacklib/human` of data files. Here is where the file `human.fig` and the accompanying psurf files are stored. There are two versions of the psurf files. Each file exists as an individual file, in both its text (.pss) version and binary (.bps) version. In addition, there is an archive file `human-5.8.a`. The psurfs in the archive are identical to the ones in the ordinary files. The archive is provided here to ensure that
7.5 Biostereometric Contour Body

There are three commands dealing specifically with the contour bodies. They are on the contour body menu, off the human menu. The contour body figure behaves exactly (and is structured the same) as the human5 figure. The only difference is the bounding geometry and size. It has about 20,000 polygons (human5 has about 1,400 polygons).

The contour figure is in the file contour.fig. The three commands are:

- create contour figure: This reads the file contour.fig, which is installed by default.

- create stick figure: This reads the file contourstick.fig, which should be an installed file. The stick figure is structurally the same as the contour figure, but each segment is replaced with a box, reducing the polygon count to about 1,900. This is for faster interactive positioning.

- switch figure: This command will switch the geometry of a contour figure with a stick figure, or vice-versa. You can position the figure, or create an animation, using the stick figure, then switch to the full contour geometry for full rendering.

7.6 Upgrading Human Figures

Note: This section is for people upgrading humans from Jack Version 4.9 or earlier.

The human manipulation and motion commands in Jack Version 5 require a special human figure definition. The older body models used with previous versions of Jack will still function as ordinary figures but they cannot be used with the new human figure features. If you have environment files which you have created with older versions of Jack, you will need to "upgrade" the human figure model in order to take advantage of the new features. This may be done with the command upgrade human figure. This may take some patience, and you may in the end have to resort to tweaking the figure's posture or even starting from scratch.

To upgrade a human figure model, first read your old environment file into Jack Version 5. Then execute the upgrade human figure command. It will prompt you to pick a human figure model. The upgrade process begins by creating a new figure and adjusting its dimensions to match the old one. It will then replace the old figure with the new one, trying its best to maintain the same posture.

The new human figure model has a hat and glasses. You can get rid of these by deleting the segments using the delete segment and then writing the figure definition as mentioned above.

The new human figure model has a redesigned neck which is much thinner than the previous models. The necks of the previous models at Penn had abnormally large necks. Unfortunately, when you upgrade a human figure, the new model has the same neck dimensions as the old one, and it looks very bad. You can solve this problem by executing the set neck dimensions command described above to change the neck thickness and width.

The upgrade process destroys any constraints on the original human figure. Since the new human figure model has internal constraints to describe its posture, any old constraints would overlap and cause problems.

7.7 Viewing Analysis

7.7.1 Eye View Windows

Jack allows you to perform a type of visibility analysis by creating a graphics window whose center of projection is located at the eyes of a human figure. This may be done with the command create eye window. This command creates a new graphics window. The window will display what the figure "sees." This command will prompt you to select a reference point:
The left eye and right eye options center the line of sight down the individual line of sight of the appropriate eye. The between eyes option centers the view forwards of the head from a reference point located between the eyes.

Figure 7.2: An eye window showing the human figure's view from between it's eyes.

### 7.7.2 View Cones

*Jack* allows you to visualize the visibility of a human figure through transparent view cones. These cones emanate from the eyes of a figure and help to demonstrate what the figure can see. You can create view cones with the command `create view cones`. This command will prompt you to enter an angular dimension and a length. The angular dimension measures how fat the cones are. The default is 40°. The length measures how long the cones are. The length of the cones is purely for appearance.

The shape of the view cone is approximates the occlusion of the nose, but this is just sample data so the shape should not be considered valid. The view cones are actually just transparent psurfs attached to the figure's eyes. The cones are read from the psurf files `rviewcone.pss` and `lviewcone.pss`.

The view cones may be deleted with the command `delete view cones`.

### 7.8 Torque and Strength Display

1. *Jack* has a basic form of static strength display, through strength windows created with the command `create strength window`. A strength window displays strength information on a degree of freedom basis, based on strength equations given in the peabody file defining the human figure. The window displays available torques at each degree of freedom, and the required torque necessary to exert a given force at the hand.

We apply Denavit-Hartenberg notational convention to build the underlying robotic model that represents the human body's peabody definition. Then a recursive (Newton-Euler) method is used to compute the joint torques on an arbitrarily loaded figure. The computation is order $O(n)$ where $n$ is the degree of freedom.

---

1. Hyeongseok Ko
of the human figure. Thus the static torques can be computed in real time while manipulating the human body.

Several visualization techniques are used in displaying the computed torque and the strength. A torque window displays the torque together with the strength information on a degree of freedom basis, based on strength equations given in the peabody file defining the human figure. The window displays the strength at each degree of freedom (purple bar), and on top of that the exerted torque - green for the static torque and blue for the dynamic torque. Once the exerted torque exceeds the joint strength, that top portion of the bar is colored red to alert you of an overloaded condition. At the same time, the joint is colored red. Also, a torque icon can be created at each joint, which is an arrow that grows or shrinks according to the magnitude of the torque.

**Tutorial 7.2: Static Torque Display Windows**

Create a human figure.

1. Execute the command `create strength window`
   The command will prompt you to select a joint group:

<table>
<thead>
<tr>
<th>type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>left arm</td>
</tr>
<tr>
<td>right arm</td>
</tr>
</tbody>
</table>

2. Choose the left arm
   A window will appear:
   The window displays a bar graph for each of the seven degrees of freedom of the arm: three for the shoulder, one for the elbow, and three for the wrist. The graphs display the available torque in the positive and negative directions along each degree of freedom. The positive direction is above the middle line; the negative direction is below.
   The secondary bar graph to the right of the available torque display is the exerted torque. If this value exceeds the available torque in either the positive or negative value, the exceeding part is colored red instead of green.

3. Move the left arm
Figure 7.4: A strength window showing available and required torque along each degree of freedom.

The current configuration of the arm (pointing down) produces very tiny exerted torque. Use the move arm command. As you move the arm, the values of the meters will change. This is because the exerted torques as well as the available torques differ at different angles. Try to identify the torque arrows. To see the drastic torque changes, you can attach an object (e.g., a small cube) at the hand, and move the arm.

The strength equations are specified in the peabody figure file. To specify new equations, you must edit the figure file and replace the old equations with new ones. The format of the equations is rather simple. For each joint in the arm, the \texttt{pstrength} and \texttt{nstrength} fields give a vector of expressions for the positive and negative strength respectively for each degree of freedom. The length of the vector should match the number of degrees of freedom of the joint: the shoulder should have three, the elbow one, and the wrist three. Each element of the vector is an arithmetic expression which can involve several terms. Each term may reference a joint angle through the \texttt{displacement} operator. For example, the strength equation for the shoulder are:

\begin{verbatim}
joint right_shoulder {
  pstrength = (19.656 +
    0.0267 \times \text{right_shoulder->displacement}[1] -
    0.0551 \times \text{right_shoulder->displacement}[2],
    18.945 +
    0.0436 \times \text{right_elbow->displacement}[0] -
    0.031 \times \text{right_shoulder->displacement}[0] -
    0.0248 \times \text{right_shoulder->displacement}[2],
    7.726 +
    0.0398 \times \text{right_shoulder->displacement}[0] +
    0.046 \times \text{right_shoulder->displacement}[1]);
  nstrength = (18.856 +
    0.0134 \times \text{right_elbow->displacement}[0] +
    0.011 \times \text{right_shoulder->displacement}[1] -
    0.046 \times \text{right_shoulder->displacement}[2],
    12.449 -
    0.0134 \times \text{right_shoulder->displacement}[1] +
    0.00072 \times \text{right_shoulder->displacement}[2] +
    0.00083 \times \text{right_shoulder->displacement}[2],
    12.948 +
    0.0352 \times \text{right_shoulder->displacement}[0] -
    0.00025 \times \text{right_shoulder->displacement}[0] +
    0.106 \times \text{right_shoulder->displacement}[1] -
    0.000825 \times \text{right_shoulder->displacement}[1]);
}
\end{verbatim}
The Jack Animation system is built around the concept of a motion, which is a change in a part of a figure over a specific interval of time. A motion is a rather primitive notion. Typically, a complex animation consists of many distinct motions, and several will overlap at each point in time. Motions are created interactively through the commands on the motion menu and the human motion menu. There are commands for creating motions which control the placement of the feet, center of mass, hands, torso, head, etc.

The animation system describes postures through constraints, and the motions control the movement of the constraints which define the postures. The motion system is designed to describe the motion of human figures, but it may describe motions of other types of figures as well.

**Tutorial 8.1: An Example Animation**

Jack Version 5 comes with a set of demo files. The files are in a subdirectory called $JACK/jacklib5/demo on the Jack distribution tape. This tutorial involves reading one of the files. Make sure that you know where the demo files are on your system.

1. **Read in the file step.env and step.motions.env**
   
   This files are in the $JACK/jacklib5/demo/animation directory on the Jack distribution tape. They are .env files which contain the motions for creating an example animation. The animation involves the human figure taking a step forward. Read the files with the read file command, just like any .env file.
   
   As the file is being read in, a small window will appear with some text on the left and a grid on the right.
   
   The names in the left column show the parts of the body which are moving. Time line beneath the grid shows time on the horizontal axis. The lines and curves in the grid show the individual motions and when they are active. The shape of the lines and curves and the neighboring text is explained below.

2. **Execute the command go**
   
   This command is on the time menu. This starts the generation of the animation sequence. The figure will begin to move and a red line will begin to move from left to right in the animation window. The red line indicates the current time.
   
   After executing the go command, you may proceed to execute other Jack commands if you wish, but for this example, just watch the figure for now. When the time line gets to the right side of the window, it will stop and the figure will stop moving. This animation is completely generated.

3. **Look at the commands on the human ⇒ motion menu.**
This example animation was generated using the commands on this menu. Each motion was created by executing one of these commands. The commands let you describe small changes in the different parts of the figure. These commands are describe later.

4. Execute the command play frames
This command is on the animation menu window. As Jack is generating the animation, it stores the position of the figure internally at each time step. The play frames command replays the frames as quickly as possible, showing you the motion as fast as your machine can draw the objects.

5. Continue with the other tutorials below

8.1 The Animation Window
Jack displays motions in an animation window. This window shows time on a horizontal axis, with a description of the the parts of each figure which are moving arranged vertically. The time interval over which each motion is active is shown as a segment of the time line. Each part of the body gets a different track. The description shows body the name of the figure and the name of the body part which is moving.

The mouse buttons have specific behavior in the animation window.

- Left mouse button
  When clicked over a motion, selects the motion. Then left or right movement on the mouse will slide the motion along the time line. When clicked over the beginning or ending of a motion, selects the
beginning time or ending time, respectively, of the motion. You may then drag that time point about on the time line to set it.

- Middle mouse button
  Snaps the current time to the point under the mouse. This is equivalent to doing a goto time command. The display is updated to that frame.

- Right mouse button
  Makes the motion under the mouse the current motion, and pops up the animation menu of commands (which will all work implicitly on this motion).

8.1.1 Time
The numbers along the bottom of the animation grid are the time line. By default, the units of time are in seconds.

When the animation window first appears, it has a width of 3 seconds. This can be changed with the arrows below the time line. The horizontal arrows allow you to scroll through time keeping the width of the window constant. The vertical arrows allow you to expand or shrink the width of the window, in time units.

TUTORIAL 8.2: SCROLLING TIME IN THE ANIMATION WINDOW
Pick up where you left off with the previous tutorial. You should have a human figure, and you should have already read in the demo animation file step.motions.env.

1. Click the mouse on the down arrow in the animation window.

   ![Animation Window Zoomed Out](image)

   Figure 8.2: The animation window zoomed out.

   This zooms out on the animation. The spacing of the grid should become smaller. It is smaller because you have zoomed out: a larger interval of time is now filling up the same space in the window.

2. Click the mouse on the up arrow in the animation window.

   This zooms in on the animation. The spacing of the grid should become larger. It is larger because you have zoomed in: a smaller interval of time is now filling up the same space in the window.

3. Click on the left arrow in the animation window.

   This scrolls the animation to the left, showing you the motions active at earlier times. This should be obvious because the time line scrolls, too. The window will not scroll past time 0. Unless you have zoomed in on the example animation, you may not be able to scroll because the entire animation may fit in the window.
CHAPTER 8. THE JACK MOTION SYSTEM

Figure 8.3: The animation window zoomed in.

Figure 8.4: Scrolling the animation window left.

Figure 8.5: Scrolling the animation window right.
4. Click on the right arrow in the animation window
   This scrolls the animation to the right, showing you the motions active at later times. The window will not scroll past the last time in the animation. Unless you have zoomed in on the example animation, you may not be able to scroll because the entire animation may fit in the window.

5. Scroll the window so that the red time line is visible

6. Press down the mouse button while the cursor is over the time line.
   The time line should turn blue. You are now moving the time line. As you move the mouse left and right, the current time changes.

7. Move the mouse left and right.
   At each step, the figure moves to its posture at that time. If you move the time line to the left or right side of the animation window, the window will automatically scroll so that the time always appears within the bounds of the window.
   When you release the button, the figure stays in that position.

8.2 Motion Parameters

8.2.1 Time Intervals
Each motion is active over a specific interval in time, delimited by a starting time and an ending time. Each motion creation command prompts you to enter a value for each of these parameters. You can enter it in seconds from the keyboard, or you can press the right mouse button in the animation window. When you press the right mouse button down, the value in the status window will display the current value. When you release the right button, the value remains at its current value. To accept the value, hit ENTER or double click the left mouse button, just like anytime you accept a value from the edit buffer.

8.2.2 Weight Functions
The yellow line drawn with each motion in the animation window illustrates the motion's weight function. Each motion describes movement of a part of the body through a kinematic constraint. The constraint is only active when the current time is between the motion's starting time and ending time. It is entirely possible to have two motions which affect the same part of the body be active at the same time. The posture which the figure assumes is a weighted average of the postures described by the individual motions. The weights of each constraint are described through the motions' weight functions. The weight functions can be one of several types:

- constant The weight does not change over the life of the constraint.
- increase The weight starts out at 0 and increases to its maximum at the end time.
- decrease The weight starts out at its maximum and decreases to 0 at the end time.
- ease in/ease out The weight starts at 0, increases to its maximum halfway through the life of the motion, and then decrease to 0 again at the end time.

The shape of the yellow line in the animation window illustrates the weight function. The units of the weight are not important. You should think of this line as an icon describing the weight function.

8.2.3 Velocity Functions
The green line drawn with each motion in the animation window illustrates the velocity of the movement which the motion describes. A motion describes movement through a kinematic constraint. Each motion describes a single movement for a single part of the figure. The starting point for the motion comes from the current posture of the figure when the motion beings. The ending position of the motion is defined as a parameter of the motion and is specified when the motion is created. The speed of the end effector along the path between the starting and ending positions is controlled through the velocity function.
constant  Constant velocity, the life of the constraint.

increase  The velocity starts out slow and increases over the life of the motion.

decrease  The velocity starts out fast and decreases over the life of the motion.

ease in/ease out  The velocity starts slow, increases to its maximum halfway through the life of the motion, and then decrease to 0 again at the end time.

The shape of the green line in the animation window illustrates the velocity function. The scale of the velocity is not important. You should think of this line as an icon describing the velocity.

8.3 The Motion of Human Figures

The commands on the human ⇒ motion menu allow you to create motions which describe movement of the parts of the human body. These motions may be combined to generate complex animation sequences. Taken individually, each motion is quite simple and uninteresting. The interplay between the motions must be considered when describing a complex movement. These motions are also subject to the behavioral constraints described in Chapter 6, except as noted below.

Each one of these commands operates on a human figure. If there is only one human figure present, these commands automatically know to use that figure. However, if there is more than one human figure, then each command will begin by prompting you to select the figure. Each of these commands will then prompt you to enter the starting time and ending time of the motion. This governs where the motion lies in time. You may enter exact values or you may enter the default values and then reposition the motion in the animation window using the mouse.

A motion is a movement of a part of the body from one place to another. The movement is specified in terms of the final position, and the parameters of how to get there. The initial position of the motion, however, is defined implicitly in terms of where the part of the body is when the motion starts. For example, a sequence of movements for the feet are defined with one motion for each foot fall. Each motion serves to move the foot from its current position, wherever that may be, when the motion starts, to the final position for that motion.

8.3.1 Controlling Behaviors Over Time

The behavior commands described in Chapter 6 control the effect of the human movement commands. The also affect the motions as well, but their effect is permanent. In other words, the effect of the behavior commands does not change over the course of an animation.

However, the "timed" behavior commands on the human behavior allow you to specify controls over specific intervals of time. These commands, create timed balance control, create timed torso control, create time hand control, and create time head control allows you to enter a specific interval of time just like the other motion commands. The behavior takes effect at the starting time and ends with the ending time. At the ending time, the behavior parameter reverts to the value it had before the motion started.
8.3. The Motion of Human Figures

8.3.2 The Center of Mass

A movement of the center of mass can be created with the create center of mass motion command. This controls the balance point of the figure. There are two ways to position the center of mass. Choose from the pop-up menu one of the two options:

- **type:**
  - between feet
  - location

The first option positions the balance point relative to the feet. To do this, it prompts you to move the center of mass into the desired location relative to the feet, and it then records the location in terms of the feet locations.

You may alternatively cause the center of mass to move to a specific point in space by choosing the location option. If you select this option, Jack will prompt you to move the center of mass to its desired location using the same technique as with the move center of mass command on the human manipulation menu.

After choosing the positioning type and entering the appropriate parameters, the command will then prompt you to enter several other parameters, including the weight function and velocity. The weight of the motion is the maximum weight of the constraint which controls the motion, subject to the weight function.

8.3.3 The Pelvis

The lower torso region of the body is controlled in two ways: through the center of mass and through the pelvis. The center of mass describes the location of the body. The pelvis constraint describes the orientation of the hips. You can cause the hips to rotate over time with the command create pelvis motion.

The create pelvis motion command will prompt you to rotate the pelvis into the final position, using the same technique as the rotate pelvis command. It will then prompt you to enter the velocity and weight functions, and the overall weight.

8.3.4 The Torso

The movement of the torso of a figure may be specified with the create torso motion. This command will then prompt you to bend the torso into the desired posture, using the same technique as the move torso command. Like the move torso command, it will also prompt you to select the torso parameters.

The create torso motion command will prompt you to enter a velocity function, but it will not ask for a weight or a weight function because this command does not use a constraint to do the positioning. Because of this, it is not allowable to have overlapping torso motions.

After the termination of a torso motion, the vertical torso behavior is turned off. The behavior of the torso can be changed at a specific moment with the create timed torso control command. This command prompts you to enter a starting time and ending time, and then to specify the type of control, just like the set torso control command. When the motion's ending time is reached, the behavior reverts to its previous value. Therefore, you should make the interval of the motion as long as you require the behavior to remain at the given value.

8.3.5 The Feet

You can control the feet of the figure through the pair of commands create foot motion and create heel motion. These two commands can be used in conjunction to cause the figure to take steps. The feet are controlled through constraints on the heels and on the toes. The toe constraints control the position and orientation of the toes. The heel constraint controls only the height of the heel from the floor. The position of the heel, and the entire foot, comes from the toes. The commands then prompt you to specify the right or left foot. You can choose this value from the pop-up menu.

The create foot motion command then prompts you to specify the ending position for the foot by moving it there using the technique of the move foot command. After specifying the new location for the foot, the
command will prompt you to enter a height. The motion causes the foot to move from its initial position to its final position through an arc of a certain elevation. A height of 0 implies that the foot moves in straight-line path. If both the initial and final positions are on the floor, then this means the foot will slide along the floor. A height of 10cm means the toes will reach a maximum height from the floor of 10cm halfway through the motion.

The effect of the create foot motion command depends upon how the figure is supported. Interactively, the move foot command automatically sets the support of the figure to the foot you are moving, and the create foot motion command does the same. However, this does not happen during the generation of the movement sequence. The behavior of the feet depends very much on the support of the figure, although effect is quite subtle and difficult to define. A foot motion can move either the supported or non-supported foot, but it is much better at moving the non-supported one.

The general rule of thumb for figure support during a movement sequence is the opposite of that for interactive manipulation: during a movement sequence, it is best to have the support through the foot on which the figure has most of its weight. This will ensure that this foot remains firmly planted.

The behavior of the feet can be changed at a specific moment with the create timed foot control command. This command prompts you to enter a starting time and ending time, and then to specify the type of control, just like the set foot control command. When the motion's ending time is reached, the behavior reverts to its previous value. Therefore, you should make the interval of the motion as long as you require the behavior to remain at the given value.

8.3.6 The Arms

The arms may be controlled through the command create arm motion. This command allows you to move the arms to a point in space or to a reference point such as a site. The arm motion may involve only the joints of the arm or it may involve bending from the waist as well. The command then prompts you to specify the right or left arm. You can choose this value from the pop-up menu. It will then prompt you to choose whether you want the arm movement to be confined to the arm or to include a bending of the torso. Arm movements involving the torso should not be combined with torso movements generated with the create torso motion command. Both of these control the torso in conflicting ways.

The command will then prompt you to move the hand to the new position in space, using the same technique as the move arm command. After you have done so, it will ask you if this position is relative to a segment. This allows you to move the hand to a global coordinate location or to a location relative to another object. If the location is relative, the hand will move to that object even if the object is moving as the hand moves during the movement generation.

You can also specify a hand behavior over time with the create timed hand control command. This allows you to temporarily attach the hand to certain objects over certain intervals of time. This command prompts you to enter a starting time and ending time, and then to specify the type of control, just like the set torso control command. When the motion's ending time is reached, the support reverts to its previous value. Therefore, you should make the interval of the motion as long as you require the behavior to remain at the given value.

Objects can be attached to the hands over an interval of time with the create timed attachment command.

8.4 Other Types of Motion

The motion menu on the utility menu has commands for creating other kinds of motions. These are described below.
8.4. OTHER TYPES OF MOTION

8.4.1 Joint Motions

The command create joint motion creates a motion which describes the movement of a single joint. It prompts you to select the joint and enter the usual motion parameters, and it then prompts you to adjust the joint. The motion will then rotate the joint to that position over its range of time.

A joint motion stores only one position of the joint: the final position into which you adjust the joint in creating the motion. Whatever angle the joint is in when the motion’s starting time is reached, it begins rotating from there to its final position. You should view a joint motion as an instruction, delivered at a particular point in time (the motion’s starting time), to rotate the joint to a certain angle by a certain time (the motion’s ending time).

You may create several successive joint motions to instruct the joint to move in a more complex pattern. This command is designed to be used with non-human figures, although it can be used with humans. It may be particularly useful in adjusting knees or elbows. Currently, joint motions controlling the same joint may not overlap in time.

The following motions all are based on the above joint motion, but are more convenient to use when animating many joints together.

create figure joint motion This command creates a motion that will move all the joints of a figure simultaneously. If you need to have all the joints moving, but don’t want an individual joint motion for each joint, then this is the motion for you. There is a small trick with this command, as well as the ones that follow. The command will prompt you for a figure to animate, after which it will not prompt you to move any of the joint angles in the figure. It will grab all the current joint angles of the figure, and use these as the goals. So you should posture the figure in its goal posture before executing this command. This is different than most other motion commands, which allow you to manipulate the designated parameter to its goal position inside the motion creation command.

The reason for this is that Jack has no command that will allow you to manipulate all the DOFs of a figure simultaneously.

When interpolation starts, the joint angles of the figure will be stored at the beginning of this motion, and interpolated together to the final goals, just as if an individual joint motion were being executed on each joint of the figure.

create joint chain motion This command operates very similarly to the previous command, except that it will animate all the joints in a chain between two arbitrary segments in a figure.

create finger motion This command will animate all the joints in the fingers of the hand. It’s as if you entered joint chain motions for each of the five fingers of the hands. Again, you must execute commands to
properly position the fingers in the goal posture before executing this command. See the grasping commands for doing this, in Section 6.6.7.

8.4.2 Figure Motions

You can describe the motion of the location of a figure with the create figure motion command. This command is similar to create joint motion except that it controls the location of a figure. It prompts you to pick a figure and enter the usual parameters, and then move the figure to its desired location.

Just like joint motions, a figure motion stores only one location of the figure: the final location into which you move it when creating the motion. Wherever the figure is in when the motion’s starting time is reached, it begins moving and rotating from there to its final location. You should view a figure motion as an instruction, delivered at a particular point in time (the motion’s starting time), to move the figure to a certain place by a certain time (the motion’s ending time).

You may create several successive figure motions to instruct the figure to move in a more complex pattern. This command is designed to be used with non-human figures, and it may cause strange behavior if used on humans. Jack and its creators cannot be held liable for death or injury resulting from its misuse.

Currently, figure motions controlling the same figure may not overlap in time.

8.4.3 Path Motions

Another type of motion is a path motion. The command create path motion prompts you to input a path, as well as a traversal direction, either forward or reverse. This will make the point site of the path move either forward or backwards over the duration of the motion. See Section 17.7 for more information on creating paths. A moving site is a useful reference for other types of motions, particularly the timed behaviors of the human figure.

You can also make a figure follow a pre-defined path with the command create figure path motion. This command prompts for a figure, as well as the other parameters of the previous command, create path motion. As the point site of the path moves along it’s interpolated path, the figure will be moved to that site (it’s root site location will be the same as the point site on the path). This is useful for moving a figure along arbitrary curved paths.

8.4.4 Camera Motion

You can describe the motions of the camera of a window with the create camera motion command. This command is similar to create figure motion in that it controls the location of a figure, in this case, the camera figure associated with a particular window. It interpolates the motion in a way appropriate for cameras. The prompts you to pick a figure and enter the usual parameters, and then change the view in the window to the desired view.

Just like figure motions, a camera motion stores only view: the final view which you set when creating the motion. Whatever the view when the motion’s starting time is reached, the camera begins moving and rotating from there to its final location. You should think of a view motion as an instruction, delivered at a particular point in time (the motion’s starting time), to move the camera to a certain place by a certain time (the motion’s ending time).

You may create several successive view motions to instruct the camera to move in a more complex pattern. Currently, view motions controlling the same window may not overlap in time.

8.4.5 Timed Events

Timed Attachments The create timed attachment allows you to attach a figure to another for over a specific interval of time during an animation. This command prompts you to enter a starting time and ending time, and then to specify the attachment, just like the attach figure command. When the motion’s starting time is reached, the attachment is made. When the motion’s ending time is reached, the attachment is cancelled. Therefore, you should make the interval of the motion as long as you require the behavior to remain at the given value. If you want the attachment to remain active over the entire length of the animation, consider attaching it once and for all with attach figure.
8.5. MOTION GROUPS

Timed Constraints Timed constraints are created with the `create timed constraint` command. The constraint will be activated on the first frame of the motion, and will remain active until the last frame of the motion, after which it will be turned off. See Chapter 10 for more information on creating constraints.

JCL Motions The command `create jcl motion` allows you to have `Jack` execute an arbitrary JCL command during motion execution. The command prompts you for 3 JCL commands, a `preaction`, `apply`, and `postaction` JCL command (any of which can be empty). The `preaction` JCL will be executed on the first frame of the motion. The `apply` JCL command will be executed on every frame of the motion (inclusive), and finally, the `postaction` JCL will be executed on the last frame of the motion. The JCL command strings must obey all JCL syntax rules.

Lisp Motions A Lisp motion can be created with the Lisp commands described in Section F. We are currently working on some good examples of this. This is an advanced topic that will be covered in an upcoming appendix.

8.5 Motion Groups

A motion group is a construct or facility for grouping related motions together. By default, whenever you create a new motion in `Jack` it belongs to the default motion group. Each motion group is represented in the animation window by a special motion, with the name of the group, and a color to identify it and its member motions. Motion groups are created with the `create motion group`. When a group motion is manipulated in the animation window, its member motions are changed also. You can move a group motion left or right in the animation window, and its member motions will move correspondingly. You may also stretch or shrink a group motion just as any other motion, and it will re-scale the starting and ending time of its member motions appropriately. You add motions to a group (other than default) by executing the command `set motion group`, described below. Once a group is created, it can be written to a file with the `write motion group definition` command. You can think of motion group files as analogous to figure files. When they are read, they instantiate a group of motions.

8.6 Time

The animation system is based on a notion of time. The actual movement of an animation sequence, as a set of discrete frames, is generated by simulating time. `Jack` has two modes related to animation: it is either advancing time or time is standing still.

The simplest way to simulate time is the `go` command. This command sets the time to 0 and begins simulating and generating frames. As it does, you are free to do other things and execute other commands, although interfering with the animation may cause problems. It is best to just let it proceed. When time advances past the last motion, it automatically stops.

When time is activated with the `go` command, interpolated frames will be stored and saved. The command `go without save` will do the same thing as `go`, but interpolated frames will not be saved. This is useful in certain special cases (as when `Jack` is being driven through a command port or lisp port. This is an advanced topic).

When time stops, you may jump to a specific time with the `goto time` command. This prompts you to enter an explicit time value. You can start advancing time with the `start time` command. You can stop it with `stop time`. Starting and stopping time is tricky, though. If you stop time with the `stop time` command and then start it again with `start time`, you may get slightly different results than if you had not stopped it, even if you didn’t change anything in between. This is because `Jack` passes control of the figures from the motions back to the passive behavioral constraints when time is stopped.

8.6.1 The Starting Frame

When you execute the `go` command, it automatically jumps to time 0 and starts simulation time 0. The position at time 0 is defined by the posture of the figures when you create the very first motion. This initial posture cannot be changed by of the motion commands, but it can be changed with the command `set frame 0`. This command takes the current posture of the figures and defines them as the postures at time 0. The
8.6.2 Moving Time in the Animation Window

Instead of jumping to different times with the *goto* time command, you can drag the current time around with the mouse in the animation window. Press the middle mouse down on the red time line and it will turn light blue. You may then drag it forwards and backwards. Note that this only works for reviewing a motion which has already been generated. If you drag the current time into the “future,” i.e. a region of time which has not been simulated with *go* or *start time*, it has no effect.

When you leave the current time indicator at a particular time, the figure assumes the posture it had at that time, including its support and its behavioral control. Note: Before changing a motion, it is a good idea to position the current time towards the end of the motion so that as you change the motion the same elements of control will be active as when the motion is in effect.

8.7 Changing Motions

You can change motions with the commands on the *change motion* menu. This menu pops up when you press the right mouse button in the animation window. To do this, point at a motion in the animation window and press down the right mouse button. The *change motion* menu will appear:

```
<table>
<thead>
<tr>
<th>change motion</th>
</tr>
</thead>
<tbody>
<tr>
<td>change motion</td>
</tr>
<tr>
<td>set motion group</td>
</tr>
<tr>
<td>delete motion</td>
</tr>
<tr>
<td>turn motion off</td>
</tr>
<tr>
<td>turn motion on</td>
</tr>
</tbody>
</table>
```

The motion to be changed or deleted is the one which is underneath the mouse. If there’s not one, the menu won’t appear. When you change a motion, you enter its parameters again as if you are creating it, except for the starting and end times. This time, however, the current values are used as the default values. Just hit return for any values you don’t want to change.

The command *set motion group* will move this motion out of its current group, and into a new one which you specify.

Note: As mentioned above, it is a good idea to position the current time inside a motion before changing it. This ensures that the behavioral control of the figure matches that which occurs with the motion.

If the motion you are changing is in fact a group motion (as described above), then the following menu will appear (instead of the *change motion* menu):

```
<table>
<thead>
<tr>
<th>motion group</th>
</tr>
</thead>
<tbody>
<tr>
<td>collapse motion group</td>
</tr>
<tr>
<td>expand motion group</td>
</tr>
<tr>
<td>turn motion group off</td>
</tr>
<tr>
<td>turn motion group on</td>
</tr>
<tr>
<td>delete motion group</td>
</tr>
<tr>
<td>collapse all motion groups</td>
</tr>
<tr>
<td>expand all motion groups</td>
</tr>
</tbody>
</table>
```

The command *collapse motion group* will force the animation window to draw only this group motion, but not the member motions. This is useful for cutting down the visual clutter in the animation window if you have many motions defined in this group. The *turn motion group off* command will turn off all the motions in the group. Conversely, *turn motion group on* will turn them all back on. The command *delete motion group* will delete the group and all its member motions.
8.8 Saving Motions

The motion menu on the utility menu has commands for writing motions to files.

The starting frame for a motion sequence is stored in an environment file. This is important because the initial posture of a figure has a great influence on its behavior under the influence of the motions. The write motions command will prompt you, first, to write the environment to a file, followed by a second file which stores the actual motions.

Motion sequences are stored in .env files. The write motions command prompts you for an .env file to store the frame 0 positions for the entire environment. It then will then prompt you for a second .env file (with .motions appended to it), for writing the actual motions to.

To read the motion sequence back in, just read the .env files with the read file command (read the frame 0 file first, then the motion file).

8.9 Playing, Saving, and Recording Frames

After you have generated a movement sequence with the go command, you can play back the frames of the animation with the play frames command. This plays the frames as fast as possible. You should only do this after the motion sequence is completely generated. Don’t do this while time is being simulated.

You can step through the frames one by one with the command step through frames command. This will prompt you to hit a key to advance forward or backward in time, one frame at a time.

The command write frames writes a file containing the frame data of the animation (actually, this is channel data, see below). This file can be read with the read frames file. This contains only the joint displacements and figure locations for each frame. It does not contain the definitions of the motions which generated the sequence. This is stored in the .env file generated by the write motions command. If you write the frames of an animation to a file, then exit Jack, run it again later and read in the frame file, you can play the frames back but the information about the motion will not be retained.

The command write frames to env files will write out each frame to a peabody environment file. It will prompt you to enter a base name to use as the file name. Each environment file will be formed by the base name followed by the frame number, with .env as the suffix.

You can record the movement sequence to the Abekas video disk with the command record frames to vdisk. This command will prompt you to enter the range of frames and their destination location on the video disk. It will then record them one by one. This takes approximately 7-8 seconds per frame.

You may also record the frames to a sequence of image files with the command record frames to rgb files. You can then use the various image format translators that come with Jack to translate the images into any format you wish (you can use the iristorle utility to convert the images to RLE format).
8.10 Channels

*Jack* stores interpolated (in-between) frame data in an object known as a *channel*. Each channel holds the interpolated data for an individual (or set) of parameters, for each frame in the animation. For example, each figure in the environment will have a channel associated with it. The figure channel will hold the root site and the global transformation matrix for each frame in the animation. If the animation is 60 frames long, each figure channel will have 60 units. The data in *all* the channels at a particular time represents the frame data.

The naming conventions for channels are simply `channeltype.name` where `channeltype` is one of `figure`, `joint`, `site`, `light`, and `name` is the peabody name of the object (e.g. the channel for the human's left elbow joint would be named `joint.human.left.elbow`). The channels are created automatically every time you issue the `go` command. Most of the time, you will never need to deal with channels, but occasionally you may want to do some special things with them, so there are several commands for dealing with channels, on the utility ⇒ channel menu:

```
main ⇒ utility ⇒ channel

- enable channel
- enable all channels
- disable channel
- disable all channels
- draw channels on
- draw channels off
```

Each channel is automatically enabled (meaning that interpolated data is stored in the channel, and the channel updates the environment on playback). The command `disable channel` can be used to disable a channel, so that it won’t be recorded over during a `go` and won’t playback during play frames. For example, if you had an animation of several figures flying around, you could disable the channel associated with one of the figures, and that figure would then not move. You can use the `enable channel` to enable a channel which has been disabled.

The command `draw channels on` will draw the interpolated channel values in a peabody window. Joint channels are not drawn, but figure channels and constraint goal channels are. This is useful if you wish to see the trace of a motion. The `draw channels off` will turn them back off (they are not drawn by default). The commands `read frames` and `write frames` actually read and write the contents of the channels.

8.11 (Near) Real-Time Animation Preview

One of the major difficulties that animators have had while working with *Jack* has been determining the correct timing for events. Quite often, it is not until the animation has been rendered and dumped to tape that unrealistic timing problems can be perceived. Since the rendering/transfer process is time consuming, a preview of the animation can save a great deal of time.

IRIX 4.0.5F includes a set of real-time movie facilities, namely `moviemaker` and `movieplayer`. *Jack* now uses these programs to create and view previews of animation sequences. After an animation has been created, and a set of frames are available (after a `go` command), the user can execute the `create preview movie` command. *Jack* will prompt for the range of frames to be included, the name of the movie file, and the resolution of the movie (this defaults to 180 x 120). The system will then resize the window in which the command was executed, and run through the animation, creating a temporary .rgb file for each frame. `Moviemaker` is then used to create a movie from these images, and `movieplayer` is invoked to view the movie. On an Indigo XS24Z this process takes between roughly 0.5 and 1 second per frame; the rate is approximately double this on a Crimson VGXT.

Note: When `movieplayer` is running, you should iconize the *Jack* window, so that *Jack* doesn’t consume all your CPU cycles. The movie playback will then be as fast as possible (otherwise `movieplayer` will start skipping frames to try and maintain 30 fps, and the motion may appear jerky).
8.12 Recording Notes

The two techniques listed below can greatly improve your Jack animations, when you are recording to video using the standard Jack hardware shading facilities.

8.12.1 Field recording

Jack can perform field recording to the Abekas A60 digital video disk. You can execute the command `set record mode` to set the `type` of recording, where `type` is either `frame` for 30 frames/second, or `field` for 60 fields/second recording. If you choose `field`, then Jack will use two consecutive frames to build one frame on the video disk. Note that it will then use 60 frames to make 1 second of animation, so you need to make your motions twice as long. This can easily be done by simply multiplying the start time and duration fields of all your motions.

For fast moving objects, and especially moving cameras, the motion will appear much smoother.

8.12.2 Full Scene Anti-Aliasing:

Jack can utilize the accumulation buffer present on SGI systems to perform full-scene anti-aliasing. This is achieved by drawing several (2 to 23) images, each slightly different, then blending the results together to remove aliasing artifacts. The results are very pleasing. To try it out, shade a window, then execute the command `render windows`. The window will be drawn several times (in single-buffer mode, so you'll notice flashing), then the final image will be displayed, with the aliasing artifacts significantly reduced. On machines with hardware accumulation buffers (VGXT and above), this is very fast. On machines with software accumulation buffer (certain VGX, Elans and below), you'll notice it's significantly slower. The command `set render parameters` tells Jack how many `samples` to take to build the anti-aliased image. Values for `samples` are: 2, 3, 4, 8, 15, or 23 samples.

There also is command `set window size nts`, which will set your window size to 720 x 485, the size used to record to the Abekas.
Chapter 9

Reading Files and Creating Objects

This chapter describes how to read files in Jack. It describes the commands for reading files as well as the rules for where Jack looks for the files that you refer to as a user. These rules can be somewhat tricky if you need a particular arrangement of your data files. Most cases involve very simple rules. If you attempt to read a file in Jack and get error messages about files not being able to be found, study these rules carefully to determine what Jack is doing.

9.1 Reading Files

The create menu has commands for reading files, which is the primary means of creating objects in Jack.

All of the objects in Jack are represented by peabody. The primary mechanism for reading files in Jack is the read file command. This command infers the type of file from its suffix and reads it accordingly. The types of files which Jack recognizes by suffix include:

- **.env** Peabody environment files. These files may contain any legal peabody syntax. It may contain a complete "environment," or it may contain a description of only a small portion of the environment, such as joint angles.

- **.fig** Peabody figure files. These files contain a template definition for a peabody figure. You will be prompted for a name for the figure.

- **.pss** Psurf files. When Jack reads a psurf file, it creates a figure with a single segment and associates the psurf with it. The name of the figure is taken from the name of the psurf, once it is made unique.

- **.bps** Binary psurf files. These are pre-processed for more efficient handling. Reading them is essentially like text psurf files.

- **.jcl** Jack Command Language files. These files contain Jack commands and arguments.
Any of these types of files may also be entered on the command line when you run Jack, and they will be read automatically.

The command read peabody string prompts you to enter a text string of peabody syntax. It will then parse the string and print the results on the screen. This does not necessarily create an object.

### 9.2 Binary Psurfs

Since psurf files are frequently fairly large, they can sometimes take a long time to read into Jack. To speed this process up, you can generate binary psurf files which contain a pre-processed form of the data which is much easier and faster to read into Jack. These files are not text files, so you can't look at them or edit them. They are meant to go hand in hand with their text psurfs. Binary psurf files have the suffix `.bps`.

Another advantage of using binary psurfs is that reduces the amount of memory which Jack requires. Jack read binary psurfs with a “lazy evaluation” method, which means that it will only read information from the file when it needs it. Most operations in Jack don't require very much information about the geometry, and Jack can get by with only a little information with each psurf. Operations which require more information may have to open the file and read more information from it. The message

```
computing psurf info for 'wksta.pss'
```

means that Jack is reading more information from the binary psurf file.

#### 9.2.1 Generating Binary Psurfs

You can generate binary psurf files from text files outside Jack with the accompanying program `bps`, which takes psurf files as command line arguments and generates corresponding binary psurfs.

For example,

```
% bps wksta.pss
```

will generate a binary psurf version of `wksta.pss` and store it in the file `wksta.bps`.

You can also generate binary psurfs with Jack using the commands on the `write` menu. The commands `write psurf` and `write global psurf` prompt you to enter the name of an output psurf file. If you enter a filename with a `.bps` suffix, Jack will write a binary psurf file. These commands are described in Section 14.2.

#### 9.2.2 Compressed Files

Jack can read compressed files, compressed with Unix utility `compress`. These files have the suffix `.Z`. Whenever Jack looks for a file, it looks to see if the file is compressed. If so, it uses the compressed version. It is not necessary to refer to the file with its `.Z` suffix, although it is legal to do so.

Any Jack file can be compressed, except for archives. Jack cannot read compressed archive files, and the files stored inside the archive may not be compressed either.

### 9.3 Keeping Track of Jack Files

Jack uses peabody and peabody data files. In many cases, these files are specific to a user's individual project, such as environment files describing a scene. In some cases, though, it is convenient for different users to share some of the same files. In this case, there is a single copy of a file and each user is allowed to read it. This may happen when several people are working on a related project, but it also happens with Jack's standard psurf and peabody files, such as the standard human figure definition, and the standard primitive objects like the cube and cylinder.
9.3. KEEPING TRACK OF JACK FILES

9.3.1 The Installation File

The need to share files and to keep files in a centrally located place requires that users give Jack special some instructions about where to look for particular files. This is done through Jack installation file. Each Jack user has a file called "./.jack5.install", in his home directory. This file lists installed files, which are the special files which Jack users will need to access. If the file is a directory, then each of the files in that directory is installed.

The format of the installation file is quite simple: it lists the complete path name of each file. When Jack starts, it reads the user's installation file and stores the information internally. From then on, when a reference is made to a file which Jack cannot otherwise find, it will look to see if the file is among the installed files.

Whenever you refer to a file, Jack always looks for the file first in the current directory, that is, the directory from which you ran the Jack program. If it is not there, then Jack looks to see if the file is installed. If, however, the file is located in the current directory and it is also installed and the installed version is somewhere else, then Jack will still read the version from the current directory, but it will issue a warning message informing you that a conflict exists, in case you did not realize that two files with the same name exist.

There are two main reasons for installing files. The first is for convenience in referring to them when reading them in. You can always read in a file by giving its complete path name, directory included, but this is too tedious. It is more convenient to refer to files by their names and have the directory determined automatically. This is the job of the installation file.

The second reason is so that when Jack is reading a peabody file and it encounters a reference to psurf or a figure file, it will be able to locate the proper file based only on its name.

9.3.1.1 Example

For example, suppose your initialization has a line in it

```c
/usr2/users/jacques/stuff/ant.fig
```

When you refer to the file ant.fig, either by reading it into Jack directly or by reading a file which makes a reference to it, Jack will look for it in the directory /usr2/users/jacques/stuff.

9.3.2 Installing Files Interactively

Most of the organizational work in installing files must be done outside of Jack. If you have object models which you have generated or acquired from somewhere else and you want to make them easily accessible in Jack, you must first decide on a convenient place to keep them, a directory in your own account or possibly in someone else's account. Once the files are located in that directory, you must "install" them, and you can do this either by editing your "./.jack5.install" file explicitly or by executing the install file command in Jack. This command simply prompts you for the name of the file and adds an entry for it in your installation file.

9.3.3 The Default "./.jack5.install" File

The first time you run Jack Version 5, it will automatically create a "./.jack5.install" file for you. This mechanism examines the shell variable JACKLIB. It expects that the value of this variable will be the name of the file jack.install file distributed on the Jack distribution tape. This file is a sample .jack5.install file which points to data files also distributed on the tape. Jack reads jack.install file and constructs your "./.jack5.install" file based on the entries in this file by prepending the directory in which the jack.install file lies.

Once your "./.jack5.install" is created, Jack does not again refer to the JACKLIB variable.
9.3.4 How Jack Reads Peabody Files

Peabody files usually make references to other files, such as figure files and psurf files. When Jack is reading a peabody file and it encounters a reference to another file, it first looks for the new file in the current directory. If it cannot find it there, it then looks for the file in the directory of the current file, that is, the one being read. If it can't find it there, it looks to see if the file is installed.

The policy of looking for files in the directory of the current file allows psurf files to be grouped together with the figure files and environment files which use them. Usually, object models are stored in the form of figure files, and they refer to several psurf files. Only the figure file should be installed. If the psurfs are grouped together with it, they will be found automatically.

9.3.5 How Jack Finds Psurf Files

Psurf files store geometric primitives, and there are several options for how Jack can access them. Typically, an application will have many psurf files, and each file may take a while to read into Jack. These options make it more convenient to organize psurfs and faster to read them.

When Jack encounters a reference to a psurf file, it automatically looks for a corresponding .bps file. It is not necessary to refer to the psurf with the .bps suffix. Jack expects that the .bps file is a pre-processed version of the .pss file. However, if a .bps file exists but it is older than its .pss file, Jack assumes that you have modified the psurf file and have not yet generated another .bps file, so it issues a warning and reads the .pss file.

Psurf files may also be stored in archives. Archives are created and maintained with the Unix utility ar. They provide a way of storing several files in one. Archives have the suffix .a. It is sometimes convenient to group the psurf files for a particular figure together into an archive. This is the case with the human figure model. Psurf archives may be associated with individual figures in the peabody language with the archive statement. When a figure has an archive, Jack will look in the archive for its psurf files. Jack uses the same mechanism to locate the archive file.

For example, assume Jack is reading a file called /usr2/users/jacques/truck/truck.fig, and Jack's current directory is /usr2/users/jacques. Assume that the truck figure has an archive truck.a. If truck.fig refers to a psurf file tire.pss, Jack will look for the file in this order:

1. /usr2/users/jacques/tire.bps
2. /usr2/users/jacques/tire.pps
3. tire.bps in /usr2/users/jacques/truck.a
4. tire.pps in /usr2/users/jacques/truck.a
5. /usr2/users/jacques/truck/tire.bps
6. /usr2/users/jacques/truck/tire.pps
7. tire.bps in /usr2/users/jacques/truck/a
8. tire.pps in /usr2/users/jacques/truck/truck.a
9. is tire.bps installed?
10. is tire.pps installed?
11. is truck.a installed? if so does it contain tire.bps? tire.pps?

The first four entries are the current directory. The next four are from the directory of the file being read.

9.4 The Uses of Peabody and Psurf Files

In order to keep files organized more efficiently, consider how files are used by Jack to describe objects and what each file represents. There is considerable leeway in how the files represent the objects.
9.5. HOW TO ORGANIZE YOUR DATA FILES

9.4.1 Uses of Psurf Files

Psurfs are the basic geometric primitives of peabody objects. Each peabody segment has a single psurf associated with it, although it is possible to have a segment which has no psurf. Some psurf files are complete “things”, like a chair or a table. Peabody figures made out of such a psurf will have a single segment and thus have no movable components. You can create such a figure simply by reading in the psurf file.

Some psurf files are designed to be used only as one component of a more complex figure. This is the case with the psurfs for a human figure definition. In this case, you will rarely ever have an occasion to refer to the individual psurf file explicitly: the peabody file which describes the figure will handle that.

Some psurf files are simply examples, such as the cube, pyramid, and cylinder. These are basic building blocks which you can use as “starters” for designing more complex objects. However, don’t confuse Jack with a computer-aided design system which allows you to easily construct complex figures by making references to these simple objects. If you make any alterations in the primitives, you will need to make your own copy.

9.4.2 Uses of Figure Files

Figure files are “templates” for peabody figures. They contain a definition of a figure, with its segments and joints, but they are not an instantiation of the figure, so they usually don’t contain joint angles and surface attribute information, because these are properties of how the figure is used in a scene. A reference to a figure file constitutes an instantiation of the figure. This happens in an environment file or when you read a figure file explicitly into Jack. When a figure file is referenced, the associated figure must be given a name.

There are several reasons to organized the definition of a figure into a figure file. The first is when the figure will be used and shared among different Jack users. The figure file makes a convenient “handle” for referencing the definition of the figure. If you need to create several instantiations of a single figure definition, then you can place the definition of the figure in a figure file and an environment file may reference it several times, giving a different name to each.

Figure files refer to psurf files which describe the geometry of the segments. It is common to group the files associated with a figure file into an archive.

9.4.3 Uses of Environment Files

Environment files can either describe entire scenes or portions of scenes. In the first case, the environment file will refer to other files, possibly figure files and psurf files. The figure file references are instantiations of the figures they define. An environment file will make a reference to a psurf file only in the course of defining a segment as a part of a figure.

Environment files need not be complete environments. An example of this is a file which contains only joint angles and describes the “posture” of a figure or figures.

9.5 How To Organize Your Data Files

The brute-force way of organizing your data files is to put all important peabody and psurf files in a single directory. This is sometimes unsatisfactory for several reasons. First, as you accumulate object models, the directory will grow and grow and you will likely lose track of which files are important and which are old and unused. Another problem is that if you ever need to copy an object model elsewhere, you must determine which peabody and psurf files it uses and copy only them. Another more severe problem is that it is not possible to have different psurfs with the same name, as is sometimes convenient.

A better way of organizing your data files is to create a different directory for each object model. Each directory should hold a few relevant figure files or environment files and its only their related psurf files. This isolates the object from other objects. Normally, there is a single file which gives a “handle” to the object, such as a figure file. This file name should be put in your .jack5. install file. You should not install all the psurf files. The psurf files will be located automatically. If you install them, they may be used incorrectly for other objects.
9.6 Getting Information About Files

When Jack reads files, it automatically prints messages on the screen giving the complete path of the files it is reading. This is especially helpful in the case of psurf and figure files. It helps you ensure that Jack is reading the proper files.

Once objects have been read in, you can get information about what files the psurfs and figures came from with the command `show object files`, on the info menu.

The command `show installed files` simply lists the files you currently have installed. This involves a little more than just listing you `~/.jack5.install` file, since it lists each file in any directories referenced in the installation file.

9.7 Creating Sites

The `create site` command prompts you for a segment on which to create the site, and then it prompts you to enter a name for the site. It then creates the site and allows you to position it relative to the coordinate frame of its segment. This transformation is the `location` field of the site.

9.8 Creating Light Sources

Peabody represents light sources as segments in the environment. In the peabody language, segments and lights are interchangeable, except that lights have luminance properties. This allows articulated figures to have individual "light source" segments.

Typically, though, it is convenient to have a light source be a single-segmented figure by itself. The `create light` command creates a figure of this type. It is a figure with a single segment. The psurf associated with the light segment is read from the psurf `light.pss`, which looks like a little flashlight. The shape of this psurf does not affect the lighting. Depending upon the parameters of the light, whether it is local or infinite, the orientation of the light may determine how it emits light. The name given to the figure and the segment is `lightsource`.

See Section 12 on for more details about lights and lighting parameters.

9.9 A Note About Surface Attributes

When Jack creates objects for which it has no surface attribute information, it creates surface attributes on its own. It assigns random colors to these attributes so that your objects are displayed in different colors. The colors it chooses are constant in saturation and value, with a random hue. You may override this behavior by supplying your own set of colors for Jack to use. See Section 12.2 for more information.
Chapter 10

Constraints

This chapter describes constraints. Jack allows you to position figures using inverse kinematics. This facility uses an iterative optimization algorithm to compute a set of joint angles which satisfy the constraint by placing the figure in a desired posture. We sometimes use the term reach because it is easy to visualize in terms of a reaching human arm. Jack allows you to define multiple constraints of various types.

Jack Version 5 has relies on constraints much more than previous versions to model human figures, although it actually requires you to know less about them. This chapter describes the details of constraints and their parameters. This information will help you understand how a human figure moves and what the human manipulation commands actually do.

It is difficult to experiment with the basics of constraints using the human figure as an example, however, because the human figure is ordinarily under the influence of several constraints doing different things. For this reason, this chapter presents the information about constraints using a different model, a chain.

10.1 Introduction

The term constraint applies to a desired geometric relationship. It also refers abstractly to all of the information which collectively defines the relationship. Constraints are solved by the inverse kinematics algorithm, which means that the appropriate joints of the articulated figures are positioned so that the desired relationships are satisfied.

Constraints are geometric connections between objects, similar in some ways to joints, although the two are intrinsically different in two important ways. The first comes arises when a figure is over constrained, meaning that all of its constraint relationships cannot be completely satisfied. The types of relationships modeled with joints, in human figure models and robot models, should never be violated, even just a little. Joints as we know them are not desired relationships: they are absolute relationships. Constraints are more flexible.

The second distinction between joints and constraints is that in practice the desired relationships commonly expressed by the two fall into different classifications. Joints are typically rotational and/or translational. The transformation across a joint consists of a set of rotations or translations along predefined axes. Constraints can be much more general and vague, such has point-to-plane, direction, etc.

For convenience, all of the commands for dealing with constraints are grouped together on the constraint menu, which is sub-menu of the edit menu.
10.2 Constraints and Motion

The purpose of a constraint is principally to describe position of a figure so that it satisfies some geometric condition. The constraint expresses the desired relationship. The constraint evaluation procedure is the process by which joint angles of the figure are set so the proper relationships are achieved.

The inverse kinematics algorithm is iterative, and each iteration generates a new set of joint angles which describe a new posture for the figure. Taken collectively, these positions can be viewed as motion: as the constraint is solved, the figure moves from its current posture into a posture which satisfies the geometric relationships. It is important, however, to keep this idea motion in the proper context. This is not motion in the sense of animation or simulation, but is an illustration of the execution of the numerical algorithm. Because of the problems with local minima and redundancies, the inverse kinematics algorithm is not good at generating motion sequences.

We often refer to the influence of constraints, and speak of figures moving under the influence of constraints. The constraints themselves are not the primary cause of motion, but they allow the figure's to reach to motion initiated by other means. If we measure the desired relationships described by the constraints with potential energy, then the process of evaluating the constraints is the search for equilibrium. The animation or simulation of articulated figures consists of the explicit movement of certain elements of the geometric environment. The process of evaluating the constraints generates the desired motions as a side effect.

10.3 The Definition of a Constraint and its Parameters

Constraints have the following parameters:

- **end effector** The end effector is the reference point on the figure which is constrained. The positional component of the end effector must be a single point. Currently, the end effector may be a site or a node. The significant part of the end effector is one dimensional: *Jack* does not yet have the ability to define multi-dimensional end effectors, such as lines or planes.

- **goal** The goal is a geometric location which describes the desired position for the end effector. Currently, the goal may be a site, a node, a face, or a general homogeneous transform.
10.3. THE DEFINITION OF A CONSTRAINT AND ITS PARAMETERS

The goal itself is a geometric entity: how the goal defines the desired position of the end effector is defined by the objective type. The positional component of the goal may be more than a single point, such as an unbounded line or plane.

**Objective type** The objective type describes the type of geometric relationship between the end effector and the goal, i.e., position, orientation, direction, etc. This describes the *distance* or *potential energy* between the end effector and the goal.

The inverse kinematics algorithm is based on the minimization of the value described by the objective function. The objective type comes in two parts: the position component and the orientation component. The two are evaluated separately and summed with the help of a position/orientation weight which describes their relative importance.

For the position component, the objective type may be *point-to-point*, *point-to-line*, or *point-to-plane*. Note that everything is point-to... because the end effector must be a point.

The orientation objective may be full 3D orientation, direction, or the hybrid types *aim* or *view*. With the aim type, the orientation of a reference vector on the end effector is directed to be aimed towards the goal point. The view is similar except with the additional restriction against twist along the aiming direction (this is useful for camera positioning where the camera should not twist but should maintain its horizontal axis perpendicular to the world vertical axis). The position of the goal then does not describe the desired position of the end effector but rather is used to describe its orientation. Therefore it is not allowable to have a position component when the orientation component is aim or view.

The goal is a geometric entity: a site, a node, a face, or a global transform in space. The objective type specifies what element of the goal is important to the objective function. With the objective types point-to-line and point-to-plane, the location of the reference point defining the line or plane is taken from the point of the goal, whether it is a node, site, etc. The direction of the line and normal of the plane are additional parameters. Therefore, it is possible to match different objective functions with different types of goals.

**Set of joints** The set of joints define which joints apply to the constraint. These joints are the variables of the constraint.

The goal, end effector, and objective type collectively describe a desired geometric relationship, but the set of joints associated with the constraint define which part of the geometric environment is allowed to "move" to satisfy the relationship. Internally, the joints are the variables to the inverse kinematics. There are two reasons for explicitly defining this set of joints. The first is efficiency: it is best to limit the number of variables which the inverse kinematics algorithm controls. The second and more important reason is control: giving explicit directions about which joints move to satisfy a positioning task provides much more local control over the behavior of the figure.

The set of joints is specified by the designation of a single "starting joint." The sequence of joints between the end effector and the starting joint defines the joint set. The set of joints of a constraint must be completely contained within a single figure.

*Tutorial 10.1: Creating a Constraint*

1. Read the file chain.fig

2. Read in a box from the file smallcube.pss

   These objects serve as a good example to illustrate the use of constraints. Position the box beside the chain, just below the top end.

3. Execute the create constraint command

   This command is on the constraint menu which is a sub-menu of the object menu.
The first parameter \textit{Jack} will prompt you for is the type of goal:

\begin{verbatim}
goal type: site
\end{verbatim}

4. Select the goal type \textit{site}

You can select the type of goal from a pop-up menu by pressing the right mouse button.
10.3. THE DEFINITION OF A CONSTRAINT AND ITS PARAMETERS

The default type is site. Select this type by hitting ENTER. You may also enter this type by clicking the left mouse button.

5. **Pick the site in the corner of the box as the goal**
   There is a site there call base.

6. **Select the end effector type site**
   When Jack prompts you to enter the end effector type, you may select it from the pop-up menu by pressing the right mouse button. The default type is site. Enter this value by clicking the left mouse button or by hitting ENTER.

<table>
<thead>
<tr>
<th>goal type</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
</tr>
<tr>
<td>node</td>
</tr>
<tr>
<td>face</td>
</tr>
<tr>
<td>closest node</td>
</tr>
<tr>
<td>hold</td>
</tr>
<tr>
<td>transform</td>
</tr>
<tr>
<td>relative transform</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>end effector type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>site</td>
</tr>
<tr>
<td>node</td>
</tr>
<tr>
<td>closest node</td>
</tr>
</tbody>
</table>

7. **Pick the site in the top end of the chain as the end effector**
   The constraint expresses a desired relationship between the goal and end effector. The goal is the box; the end effector is the end of the chain.

8. **Select the orientation relationship type none**
   The orientation relationship describes the desired relationship between the orientation of the goal and the orientation of the end effector. In this example, we want the orientation to be ignored, so enter the type none.

<table>
<thead>
<tr>
<th>orientation relationship type:</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
</tr>
<tr>
<td>align_frame</td>
</tr>
<tr>
<td>align_direction</td>
</tr>
<tr>
<td>planar_direction</td>
</tr>
<tr>
<td>aim</td>
</tr>
<tr>
<td>view</td>
</tr>
</tbody>
</table>

Do this by pressing the right mouse button and selecting the none entry from the pop-up menu.

9. **Select the position relationship type point-to-point**
   The position relationship type describes how the position of the goal affects the position of the end effector. The choices are given in the pop-up menu, which you get by pressing the right button.

<table>
<thead>
<tr>
<th>position relationship type</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
</tr>
<tr>
<td>point_to_point</td>
</tr>
<tr>
<td>point_to_line</td>
</tr>
<tr>
<td>point_to_plane</td>
</tr>
<tr>
<td>point_to_face</td>
</tr>
<tr>
<td>limit.spring</td>
</tr>
<tr>
<td>rest_angle</td>
</tr>
</tbody>
</table>
The default value is point-to-point. This type causes the constraint to position the end effector at the point of the goal.

10. Enter no to the question “use current position offset?”

11. Pick the bottom joint of the chain
   The starting joint defines which joints in the figure apply to the constraint. The constraint satisfaction algorithm will use these joints to position the end effector at the goal.
   Selecting the bottom joint means all joints in the chain will move to satisfy the constraint.

12. Enter a weight of 1.0
   The constraint weight of a goal specifies its relative importance when there are several goals.
   The default value of 1.0 is suitable for most circumstances.

13. Watch the constraint be solved!
   After entering the weight, the specification of the constraint is complete and Jack begins to solve it. The end of the chain should move gradually towards the box and stay there. The constraint is illustrated with a white line drawn between the end effector and the goal. You can think figuratively of this line as a rubber band, pulling the end of the chain towards the box. When the chain reaches the box, the white line becomes a dot.

![Figure 10.2: The end of the chain is constrained to the cube.](image)

**TUTORIAL 10.2: MANIPULATING CONSTRAINED FIGURES**
This example explores moving figures which have constraints. Continue with the constraint you just created between the chain and the small box.

1. Move the box figure
   Constraints remains active after they have been created and solved. To see this, execute the move figure command and select the box. As you move the box, the end of the chain will stay on it. If you move the box outside of the range of the chain, the white line will reappear, showing where the chain is reaching, but the chain will not come apart.
   The constraint describes a desired position for the end effector, but it does not say anything about the joints in the joint chain, except that they must stay within their limits. This can cause problems with redundancies, in which the elbow frequently gets stuck in the wrong position. The way out of this is to either continue moving the object and attempt to “wrap” the chain around, or move the elbow directly. We will discuss this later.
2. Move the chain figure
Leave the box in a convenient place in front of the chain, and then move the chain figure with the move figure command. Now as you move the figure, the chain will move around to stay on the box.

3. Adjust one of the joints in the middle of the chain
The joint is in the joint chain of the constraint. The constraint satisfaction algorithm is interactively determining joint angles for all joints in the chain. If you adjust one of the joints, Jack temporarily takes the joint out of the joint chain so that the inverse kinematics algorithm doesn't consider it as a variable.
This also causes a strange artifact: the position of the joint moves out from under the rotation wheel! This is actually quite natural since the entire chain is moving, but it means that the actual location of the joint moves away from the rotation wheel as you rotate.

![Figure 10.3: Adjusting a joint that is also under the influence of a constraint.](image)

**Tutorial 10.3: Creating Another Constraint**
Jack allows you to create several constraints and have them all active at once. In this example creates another constraint. You should continue with the constraint between the end of the chain and box which you created in the previous examples.

1. Read another box from the file smallcube.pss
Position the box at the middle of the chain.

2. Create a constraint between the new box and a site in the middle of the chain
Use the same parameters as before, but use the a different site as the goal. The chain is now constrained to the two boxes in different places.

10.4 How Jack Solves Constraints
Jack uses an iterative numerical procedure to evaluate the constraints. It helps to understand the behavior of the constraints to understand a little about the underlying algorithm.
The algorithm is a minimization process which computes the total "potential energy" of the constraints. Each constraint has an energy, which is the distance between the end effector and goal. The distance need
not be purely euclidean distance, since constraints can measure orientation as well. The total potential energy of the system is the sum of the potentials of each constraint, using the goal weights as weighting factors.

The central element of the algorithm is the ability to determine the gradient of the total potential energy, or how the potential changes as things move in space. It uses the gradient to compute the jacobian, which relates the change in each joint angle to the change in the total potential energy.

The algorithm begins by computing the jacobian of the initial configuration. This determines an increment in all the joint angles which will reduce the total potential energy. This increment is applied to each of the joint angles, and this gives a new posture to the figures. The new posture does not necessarily satisfy the constraints, but it is closer than before. The algorithm then repeats itself until the potential energy is reduced below some threshold, or until it determines that it cannot be reduced further. Because of the way the gradient governs the direction of the end effector, it is often called a gradient descent approach.

There are two problems with this approach: redundancies and local minima. Each requires some special care and understanding.

10.4.1 "Why is the Elbow Over There?"

It is frequently the case that a constraint does not uniquely determine the angles for its joints. Instead, there is some leeway in where the segments in the interior of the joint chain go. A good example of this is a reach with the human arm. When the finger tips are placed at the proper location, the elbow may be any of many locations. The constraint satisfaction algorithm is not particularly good at choosing among them. In fact, the posture which algorithm generates is simply the one it arrives at in the gradient descent.

The solution to this problem may be as simple a interactively manipulating the figure to get it “kinked” into the proper position. Another approach is to adjust the internal joints of the joint chain to get them positioned appropriately. It may also be useful to create additional constraints which describe the posture more rigorously.

10.4.2 "Why Isn’t That Constraint Satisfied?"

The other problem with the algorithm is that it can get caught in local minima. Because of the gradient descent approach, the algorithm will only bend joints in a direction which strictly decreases the potential energy. It cannot consider “backing up” to reach a solution.
This situation can arise when joints such as the elbows and knees become “locked” in place. An example of this is when the arm is fully extended outward and then drawn back in towards the body. When the goal comes back inwards, the fingers follow it and the wrist curls, but depending upon exactly where the goal lies, the elbow may not bend because doing so would temporarily increase the potential energy. Although the goal is reachable, the algorithm needs some assistance in knowing which joints to bend.

The problem can be alleviated usually by manually tweaking the joint angles in the joint chain, or by dragging the goal around interactively.

### 10.4.3 Constraint Evaluation Parameters

*Jack* employs two techniques for improving the interactivity with constraints. The first is employed by the `move figure command`, and it involves limiting the amount of time spent in the inverse kinematics algorithm at each interactive iteration. As you move figures interactively, *Jack* applies the positioning algorithm at each iteration. When there are several constraints, this can be quite time consuming and the interaction rate can degrade considerably. Therefore, *Jack* uses what it calls a *constraint time limit*, which is the length of time it spends in the algorithm at each iteration. If this amount of time expires, then *Jack* accepts the intermediate position, which is not complete, but is better than before. The default time limit is 0.1 seconds.

The second technique which *Jack* uses helps to alleviate the problem local minima by taking advantage of the fact that the algorithm takes its initial “guess” of a solution from the current configuration. This means that it is very good at solving constraints in which the end effectors are very close to their goals. In these cases, the end effectors move very smoothly. *Jack* places a limit on the distance between the end effector and the goal. As you manipulate figures interactively, the end effectors become separated from their goals. Instead of using the actual goals, *Jack* uses *intermediate* goals placed in the direction of the end effectors but only a fixed distance away. This causes the end effector to move more smoothly towards the real goal when the goal is far away. The default *step factor* is 10 centimeters.

---

**TUTORIAL 10.4: CHANGING THE CONSTRAINT STEP FACTOR**

This example is easiest to understand with a single constraint. Set up an example with the chain and the box with a single constraint.

1. **Execute the command** `set constraint step factor`
2. **Enter a value of 5.0**
   The default value is 10cm, which is the distance which the goal is allowed to move in one iteration.
3. **Move the box**
   As you move the box around, you should notice that the chain follows the box more slowly, but its motion is smoother.
4. **Set the step factor to 0.0**
   This effectively disables the step factor calculation and uses the complete goal at each iteration. If you move the box around again, you should notice that the interaction is more sluggish and the chain may flap around uncontrollably.

---

**TUTORIAL 10.5: CHANGING THE CONSTRAINT TIME LIMIT**

This example requires that you have several constraints. You can use the ones you created in the previous examples. When there are several constraints, the performance tends to be bounded by the constraint time limit, so changing the limit will have a noticeable difference.

1. **Execute the command** `set constraint time limit`
2. **Enter a value of 0.20**
   The default value is 0.10, which is the number of seconds to spend per iteration. A value of 0.2 will double the amount of time spent.
3. **Move the box**
   You should notice that the interaction is slower: *Jack* will seem more sluggish and the window will be redrawn much more slowly. The constraints, however, should be solved more completely at each iteration.

4. **Execute set constraint time limit and enter a value of 0**
   This will effectively disable the time limit and the constraints will be solved completely at each iteration.

### 10.5 Controlling Constraints

The most important aspect of constraints is how they are evaluated, since it is the evaluation process which positions the figures so that the desired relationships are met. Normally, *Jack* evaluates constraints continuously so that as soon as you create a constraint, it begins to be evaluated. When you move a figure interactively, *Jack* also evaluates the constraints.

It is sometimes convenient, however, to turn disable the evaluation of constraints. This can be done with the commands **turn constraints off** and **turn constraints on**. When constraints are turned off, they still exist and are still drawn, but they aren't evaluated. They can be evaluated either by turning them back on or by executing the command **evaluate constraints**.

The command **evaluate constraints** evaluates constraints completely, ignoring the step factor and time limit.

#### ♠ Tutorial 10.6: Turning Constraints On and Off

1. **Execute the command** turn constraints off
2. Create several constraints
3. **Execute the command** evaluate constraints
   - The intermediate steps which the figures go through illustrate the execution of the inverse kinematics algorithm. This algorithm is monotonically convergent, so that at each iteration the end effectors are closer to their goals, but it tends to oscillate so the end effectors generally follow zig-zag path.
4. **Move one of the figures**
   - The while lines and fans will tell you where the end effectors are supposed to go, but they won’t go there as long as the constraints are off.
5. **Execute the command** turn constraints on
   - The constraint evaluation process will begin again with each end effector generally following a straight line path towards its goal.

### 10.6 Changing Constraints

You can delete constraints with the command **delete constraint**. You can delete all constraints with the command **delete all constraints**. Another command **delete all non-rooting constraints** is useful to delete all constraints except for rooting constraints. This is because rooting constraints typically serve as connections between objects which you may want to treat differently from other constraints.

If you want to change just some of the properties of a constraint, you can do so with the command **change constraint**. This command is similar to **create constraint**, except that it operates on a constraint which already exists, and it uses the current values as the defaults, helping you to alter just part of the constraint.

These commands prompt you to pick a constraint. You can pick a constraint by pointing at either the goal, the end effector, or the line between them.

You can get information about constraints with the **describe constraints** command.

#### ♠ Tutorial 10.7: Deleting a Constraint
10.7 Exploring Relationship Types

This set of examples demonstrates the behavior of the different objective types. To do this, use the chain and the box from the psurf file smallcube.pss. For each example, create a constraint between the end of the chain and the box. You should do each example by deleting the previous constraint, resetting the chain figure and box to its default position, and then creating a new constraint. Although it is not necessary in general to do this, these examples describe behavior which you may not be able to reproduce if the figures begin in some awkward position.

The simplest way to proceed is to create a chain figure model and a box and then write out an environment file with the command write environment. Then proceed with each example, creating a constraint and investigating its behavior. Then delete the constraint and re-read the environment file. This will cause the figures to be reset to the position described in the file. Be sure to write out the environment file before creating the constraint, because constraints are written to environment files, too.

The position and orientation components of a constraint may be mixed and matched as you please. These examples demonstrate the behavior of each type in turn. For the position components, it is easiest to visualize them without orientation. The orientation types can be specified without any position component, but the result is rather strange since the end effector is free to flap around in space. The frame and direction types are easiest to visualize with a point-to-point position component.

1. Execute the command delete constraint
   This command prompts you to pick a constraint. You can pick a constraint with the mouse by pointing at either its goal or end effector, or the white line between the two.

2. Press the mouse while the cursor is over either the end effector or the goal
   The constraint goes away.

† Tutorial 10.8: Changing a Constraint

In order to change a constraint, you must have one to operate on. In this example, use the constraint which you created in the previous examples between the chain and the box.

1. Execute the command change constraint

2. Pick the constraint between the chain and the box
   The process of changing a constraint is parallel to creating the constraint from scratch, although the default parameters are taken from the current properties of the constraint.
   For the parameters which you wish to leave as is, just enter the current value.

3. Enter all the default values, up to the starting joint
   For the goal and end effector, don’t pick sites: just hit ESCAPE. This will signify to leave the value as is.

4. Pick the bottom joint as the starting joint
   This will mean that all joints in the chain will apply to the goal.

† Tutorial 10.9: Point-to-Line Constraint

Reposition the chain and box to their original location.

1. Enter the orientation relationship type of none
   You may enter an orientation component with a point-to-line goal, but for the purpose of this example, do not do so now.

2. Enter a position relationship type of point-to-line
   Jack will prompt you to enter a direction vector for the line. The direction vector is local to the coordinate frame of the goal transformation. The line is assumed to pass through the origin of the goal.
3. Enter a direction vector of \((0, 1, 0)\)
   This is the \(y\) axis of the box. If the box has not been rotated, then this extends upward.

4. Move the Box
   Translate the box side to side and the chain will remain along the line of its \(y\) axis. If you translate the box upward, the chain doesn't move: it is only constrained to lie along the given line. If you rotate the box, the line will rotate, too, and the chain should remain on it.

\[\text{Tutorial 10.10: Point-to-Plane Constraint}\]

1. Enter the orientation relationship type of none
   You may enter an orientation component with a point-to-plane goal, but for the purpose of this example, do not do so here. The orientation and position are applied independently.

2. Enter a position relationship type of point-to-plane
   Jack will prompt you to enter a normal vector for the plane. The normal vector is local to the coordinate frame of the goal transformation. The plane is assumed to pass through the origin of the goal.

3. Enter a normal vector of \((0, 1, 0)\)
   This is the \(y\) axis of the box. If the box has not been rotated, then this extends upward.

4. Move the Box
   Translate the box side to side and the chain will remain in the plane defined by the origin of the goal and normal to the \(y\) axis of the box. If you translate the box side to side, the chain doesn't move: it is only constrained to lie along the given line. If you rotate the box, the plane will rotate, too, and the chain should remain in it.

\[\text{Tutorial 10.11: Align-Frame Orientation Constraint}\]

Create a constraint between the chain and the box.

1. Create a constraint with orientation relationship type frame
   Jack will then prompt ask you whether you want to use the current offset:
   
   | use current offset? | yes |

2. Enter the default answer yes
   An orientation constraint will attempt to align the orientation of the end effector with the orientation of the goal. This means that the \(x\), \(y\), and \(z\) axes of the end effector will be aligned with the \(x\), \(y\), and \(z\) axes of the goal. However, frequently that is not really the orientation that you want to the end effector to have. You can specify an offset between the end effector and the goal which is a fixed orientational relationship between them. This transformation is optional. If you answer yes, then Jack will use the current relationship between the end effector and goal as the desired orientational relationship. If you enter no, then Jack will not use an offset and will align the two frames as is.

3. Set the position relationship type to point-to-point
   This is allowed to be any of the position types, but for this example, use point-to-point.

4. Enter a position/orientation weight of 0.5
   The position/orientation weight specifies the relative importance of position over orientation. 0.0 implies all position; 1.0 implies all orientation. The default value of 0.5 is sufficient for most cases.
5. Enter the remaining default parameters

An orientation goal is drawn with with blue "fans" between the current orientation and
the desired orientation. When the orientation is met, the fans disappear. The fans are the
rotational analog of the rubber bands. They help to illustrate which way the constraint is
attempting to move the end effector.

6. Move the box

As you move the box, the position of the end of the chain will follow the box because of
the point-to-point position component. The orientation of the end of the chain should remain
fixed relative to the box as well. Rotate the box and the segments in the chain will rotate
along with it.

![Figure 10.5: Moving the cube.](image)

<table>
<thead>
<tr>
<th>TUTORIAL 10.12: ALIGN-DIRECTION CONSTRAINT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reposition the chain and box to their original location.</td>
</tr>
</tbody>
</table>

1. Create a constraint with orientation relationship type of direction

Jack will prompt you to enter an end effector vector and a goal vector. These vectors are
local to the coordinate frame of the end effector and goal, respectively. The constraint will
attempt to align these two vectors in space.

In this example, the box is aligned with the world coordinate frame so the x axis extends
to the side, y is up, and z is out. The end of the chain has the y axis extending upwards,
the x axis to the right, and the z axis extending forwards.

2. Enter the end effector vector (0, 0, 1)

This is the z axis of the end of the chain.

Jack will then prompt you to enter the goal vector.

3. Enter the goal vector (0, 0, 1)

This is the z axis of the box.

4. Set the position relationship type to point to point

This is allowed to be any of the position types, but for this example, use point-to-point.

5. Enter a position/orientation weight of 0.5

The chain will move so that it stays on the box but so that it also points in the same
direction as the z axis of the box.
6. Move the box
   If you translate the box, the chain will follow, keeping the same orientation. If you rotate
   the box, then the chain will rotate as well.

† TUTORIAL 10.13: AIM CONSTRAINT
Reposition the chain and box to their original location.

1. Create a constraint with orientation relationship type of aim
2. Enter a position component of none
   With the aim constraint, the relative position of the goal and end effector define the aiming
   direction, so the position of the goal influences the orientation of the end effector, unlike the
   other types of orientation. The orientation of the goal is ignored.
   If you select no position component, then the end effector is free to move in space in any
   way so that it can achieve the proper orientation. Adding a position component means that
   it will be drawn towards the goal as well.
   Jack will prompt you to enter an end effector vector. This vector is local to the coordinate
   frame of the end effector transform. The constraint will attempt to position the end effector
   so that this vector is aimed at the goal.
3. Enter the end effector vector (0, 1, 0)
   This is the y axis of the end of the chain.
4. Move the box
   As you move the box, the end of the chain will remain aimed at the box.

The view constraint is almost identical to the aim constraint except for an additional condition on horizontal twist. This is useful for orienting objects such as eyes and cameras when you want to keep them from twisting in space.

10.8 Hold Goals
Sometimes, it is convenient to create constraints where the goal is not associated with any peabody objects. This is possible by selecting the goal type hold, which signifies that the end effector should remain at its current location. The position and orientation of the goal is taken from the current location of the end effector when you create the constraint.

† TUTORIAL 10.14: CONSTRAINTS WITH HOLD GOALS
Use the chain figure.

1. Create a constraint and enter a goal type of hold
   This signifies that there is no peabody object at the location of the goal.
   Jack will not prompt you to pick a goal.
2. Select the site in the end of the chain as the end effector
3. Enter an orientation relationship type of frame
4. Enter a position relationship type of point-to-point
5. Select the bottom joint as the start joint
6. Complete the constraint
   Enter a weight of 1.0 and an position/orientation weight of 0.5.
7. Move the chain
   The end of the chain should remain in the same place. If you move the figure up, you will
   see the white line illustrating where the end of the chain is supposed to be. The orientation
   fans show its desired orientation.
   Move the figure backwards and upwards and the end of the chain should remain at the same
   place in space.
10.9 Face Goals

Jack allows you to have constraints with goals which are multi-dimensional, i.e. not just a single point in space. When you select a face goal type, the end effector is directed towards any point in the face. With a face goal, you may have a position relationship of any type: point-to-point, point-to-line, or point-to-plane. The orientation relationships use the orientation of the segment to which the face belongs.

With a point-to-point position relationship, the potential energy of the constraint is the shortest distance between the end effector and the face. The relationship is a piecewise function. If the point on the face closest to the end effector is a vertex of the face, then it acts like a point-to-point type. If the closest point is along an edge of the face, then it acts like a point-to-line type. If the closest point is in the interior of the face, then it acts like a point-to-plane type.

With the point-to-line relationship with a face goal, the line is allowed to pass through anywhere in the face. The direction of the line is a parameter which you must enter. The default value which Jack supplies you is the normal to the face.

With the point-to-plane relationship, the plane passes through the center of the face. The normal to the plane is a parameter which you must enter. The default value is the surface normal of the face. With this value, the plane is the plane of the face.

♠ Tutorial 10.15: Constraints with Face Goals

1. Create a constraint with a goal type of face
   Jack will prompt you to pick a face. Pick one of the faces of the box.
2. Enter the orientation relationship type of none
   You may enter an orientation component with a face goal, but for the purpose of this example, do not do so here.
3. Enter the position relationship type of point-to-point
   In this case, it is technically not point-to-point, since the constraint attempts to place the end effector anywhere within or along the boundary of the face.

10.10 Interactive Reaching

Sometimes it is convenient to move a branch of a figure without the overhead of defining a constraint to another object and moving the object. The interactive reach command allows you to drag an end effector as if there were a constraint between it and and point you manipulate. It actually creates a temporary hold constraint and then allows you to interactively move the holding position. When you stop moving, the constraint goes away.

In previous versions of Jack, the interactive reach command provided a good way of moving the arms of a human figure. In Jack Version 5, the move arm command is designed to do this, and you should not use interactive reach. However, the move arm command is really just a special version of interactive reach.

♠ Tutorial 10.16: Dragging an End Effector

1. Execute the command interactive reach
   The parameters of the interactive reach are similar to an ordinary ordinary constraint, except that there is no goal: the goal will be a global transform which you will manipulate interactively. Also, the relationship type is automatically set to point-to-point.
2. Pick the site in the end of the chain as the end effector
3. Pick the bottom joint as the start joint
   This completes the specification of the temporary constraint. The chain will appear highlighted in yellow. This is to illustrate the portion of the figure which is being moved by the interactive reach.
CHAPTER 10. CONSTRAINTS

Jack will prompt you with a message similar to the move figure message:

```
Move end effector '(site)chain.link6.distal' ...
```

You are now interactively manipulating the goal of the constraint using the same movement mechanism as the move figure command. You can translate and rotate the goal just like an ordinary figure.

As you move the goal, the usual constraint line appears between the end effector and the goal position.

4. Hit ESCAPE to stop moving

Sometimes it is convenient to have orientation in the interactive process. The interactive reach command creates a temporary constraint with a point-to-point position relationship and a frame orientation relationship, but with a position/orientation weight of 1.0, meaning all position. This causes the orientation of the goal to be ignored. You can interactively adjust the position/orientation weight by hitting the o and p keys. These increase and decrease the orientation weight, respectively.

<table>
<thead>
<tr>
<th>Tutorial 10.17: Reaching with Orientation</th>
</tr>
</thead>
</table>

1. Execute the interactive reach command
   Try rotating the goal and you will see that it has no effect on the chain.

2. Hit the o key
   A message will appear in the upper left hand corner of the screen saying

   90% position
   10% orientation

   Hit the o key several more times. With each o, the orientation percentage increases. Increase it to 0.5. The p key increases the weight in favor of position.

3. Rotate the goal around the z axis
   The chain should rotate as well, moving the interior of the chain along with it. You may need to move the end of the chain down a little in order for it to reach. Don’t rotate them too far or they won’t go. This motion is bounded by the joint limits. The fans show you the desired orientation.

4. Translate the goal
   The end of the chain should move as before, but now it should keep the same relative orientation as well.

A common sequence of operation is to use the interactive reach to position an end effector near a particular object and then create a constraint to keep it there. You can do this automatically as you terminate the interactive reach command by hitting the letter C. This will stop the interactive moving procedure and prompt you to enter a goal.

<table>
<thead>
<tr>
<th>Tutorial 10.18: Creating Constraints with Interactive Reach</th>
</tr>
</thead>
</table>

1. Execute the interactive reach command
   Move the chain close to the box.

2. Hit the C key
   Jack will prompt you to enter a goal type and goal just as if you were creating a constraint.

3. Enter a goal type and a goal
   The constraint will be created using the same parameters as existed for the interactive reach. It is not possible to use this mechanism to create constraints with other types besides point-to-point and frame orientation.
10.11 Rooting Constraints

Older versions of Jack relied on the notion of a rooting constraint to attach figures to one another. Jack 5\footnote{\textsc{Tutorial} 10.19: Rooting Constraints} has a much better mechanism of doing this, one which doesn't use rooting constraints. This mechanism is provided through the attach figure command, described in Sections 6.4.2.1 and 6.6.4. However, rooting constraints still have a useful purpose, though, because they allow an object to be attached to two different objects simultaneously.

The constraints we have discussed so far have their behavior limited to the joints defined in the chain between the end effector and the starting joint. Only the portion of the figure along this chain can be affected by the constraint. These constraints affect the posture of the figure. It is also possible to create constraints which control the location of a figure. This allows a figure to be connected to another figure. Therefore, these constraints are called rooting constraints.

The specification of a rooting constraint is just like an ordinary constraint except that you do not specify a starting joint. No joints are affected by the constraint. The constraint variable in this case is the figure location.

Just as with ordinary constraints, a rooting constraint has an end effector, a goal, and an objective function. The objective function describes the desired geometric relationship between the end effector and the goal. The inverse kinematics algorithm determines a location for the the figure which minimize the objective function by placing the end effector at the goal. This causes the entire figure to move, although the joints remain fixed.

You can think of rooting constraints as another way describing the location of a figure, but it can do much more. It is possible to have several rooting constraints for each figure. The resulting location of the figure is determined by minimizing the potential energy of each of the constraints.

This situation complicates the process of moving a figure interactively in Jack when the figure has a rooting constraint. Technically, the process of moving a figure in Jack specifies a location for the figure's root, although if the figure has a rooting constraint, there is a conflict between the position specified by the moving facility and the position dictated by the constraint. Therefore, the move figure command first looks to see if the selected figure has a rooting constraint. If it does, it creates a temporary constraint between the figure's root and the transform being manipulated by the moving process. The position which the figure assumes is a weighted average of the two constraints, determined by the inverse kinematics algorithm.

<table>
<thead>
<tr>
<th>Tutorial 10.19: Rooting Constraints</th>
</tr>
</thead>
</table>

In this example, create a rooting constraint between the box from the file smallcube.pss and the chain from the file chain.fig. The constraint will connect the box to the end of the chain so that when the chain moves, the box follows it. Note that this is the opposite behavior of the previous examples.

1. Execute the command create rooting constraint
2. Pick the site in the end of the chain as goal
3. Pick the site in the corner of the box as the end effector
   The end effector is on the box because we want to control the location of the box with the chain, not vice versa.
4. Enter an orientation relationship of none
5. Enter a position relationship of point-to-point
6. Enter a weight of 1.0
   Note that no joints are involved. This completes the specification of the constraint, and the box should snap to the end of the chain
7. Adjust a joint of the chain.
   The box should follow the end of the chain, but keeping the same global orientation. Since the constraint has no orientation component, the orientation of the box is unaffected, so it doesn't rotate. If we had chosen a frame orientation component, the box would have rotated with the end of the chain.
TUTORIAL 10.20: MOVING FIGURES WITH ROOTING CONSTRAINTS

This example involves interactively moving a figure which has a rooting constraint. Use the constraint between the box and the chain which you created in the previous example.

1. Move the box

When you execute the move figure command and select the box, you will get a message saying that the box is constrained. Jack will automatically create a temporary additional rooting constraint and the move figure command will allow you to move its goal.

As you translate the box, you will notice that you do not have complete control over it. That is because there is a conflicting description of where the box should be: the box’s rooting constraint tells it to go to the box, but the moving facility tells it to follow the mouse. The box’s location is an average of the two.
Chapter 11

Changing Objects

The commands on the object menu allow you to interactively change properties of objects.

<table>
<thead>
<tr>
<th>object</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete environment</td>
</tr>
<tr>
<td>delete figure</td>
</tr>
<tr>
<td>delete joint</td>
</tr>
<tr>
<td>delete segment</td>
</tr>
<tr>
<td>reroot figure</td>
</tr>
<tr>
<td>attach figure</td>
</tr>
<tr>
<td>unattach figure</td>
</tr>
<tr>
<td>figure center of mass</td>
</tr>
<tr>
<td>scale environment</td>
</tr>
<tr>
<td>scale environment globally</td>
</tr>
<tr>
<td>scale figure</td>
</tr>
<tr>
<td>scale figure globally</td>
</tr>
<tr>
<td>create joint between segments</td>
</tr>
<tr>
<td>create joint between sites</td>
</tr>
<tr>
<td>merge segments</td>
</tr>
<tr>
<td>flatten hierarchy</td>
</tr>
<tr>
<td>split segment</td>
</tr>
<tr>
<td>set joint type</td>
</tr>
<tr>
<td>set joint limits</td>
</tr>
<tr>
<td>freeze joint</td>
</tr>
<tr>
<td>thaw joint</td>
</tr>
<tr>
<td>rename figure</td>
</tr>
<tr>
<td>rename joint</td>
</tr>
<tr>
<td>rename segment</td>
</tr>
<tr>
<td>rename site</td>
</tr>
<tr>
<td>rename attribute</td>
</tr>
</tbody>
</table>

11.1 Re-rooting Figures

The reroot figure command changes the root site of a figure. Jack represents figures in a hierarchy. The joints represent transformations between levels in the hierarchy and determine the internal posture of a figure. One site on each figure is designated as the figure's root and represents the root of the hierarchy. The location of a figure is defined as the global location of the figure's root site. This location is only meaningful when
associated with a particular reference point on the figure, which is the root site. jack allows you to root figures through any of their sites and to change the root whenever convenient.

The root site serves as the origin of translation and rotation when you move a figure. When you move a figure interactively, you are manipulating its location, i.e., the global location of its root site. The translation and rotation icons are centered at the root. Therefore, changing the figure root changes the center of rotation.

Since the joints in a figure constitute transformations from one level of the hierarchy to another, when a particular joint rotates, only the segments below it in the hierarchy will be affected. This means that the “behavior” of joints depend upon how the figure is rooted. Joints connect two segments: when a joint changes angle, only the segment on the opposite side of the joint from the figure root will move.

Jack represents the root of human figures in a special way. See Section 6.4.3. If you attempt to re-root a human figure in Jack with the reroot figure command, Jack will issue an error message and refuse to do it.

Twotorial 11.1: Re-Rooting a Figure

1. Read in the chain figure from the file chain.fig
   The default root for the figure is the bottom site. When you rotate the figure, it rotates about its lowest point.

2. Execute the reroot figure command
   It will prompt you to pick a site. This site will become the new root for the figure to which it belongs.

3. Select a site in the middle of the figure
   The figure will not move and does not appear to change. However, the “location” of the figure has changed, because the location now describes the position of the new root site. Jack automatically initialized the figure’s location so that the figure as a whole remains in the same place.

4. Move the figure
   The center of rotation and translation should now be located in the center, at the site you selected.

5. Adjust the bottom joint of the chain
   The figure’s upper part remain fixed, while only the bottom link moves.

Figure 11.1: Adjusting the bottom joint after re-rooting the chain.
11.2 Scaling Figures

Scaling a figure changes its overall dimensions. You can scale figures equally in all directions, or you can scale a figure by different amounts in the $x$, $y$, and $z$ direction. Scaling a figure equally in all directions is quite straightforward. Scaling by dimension is more complex.

You may scale figures with the `scale figure` command. This prompts you to enter a vector giving the scale factor in each dimension. To scale the figure equally in all directions, you can either enter the same value for the $x$, $y$, and $z$ scales, or you can enter a single number in place of the vector. The scale factors are ratios applied to the current size of the figure, not overall dimensions. You can un-scale a figure by scaling by the inverse of the scale. Please don't scale by zero: you won't be able to recover!

When you scale a figure differently in each dimension with the `scale figure` command, Jack scales each segment in its own local coordinate system. The affect may not be what you expect because the individual segments may have orientations different from the global orientation of the figure.

The `scale figure` command modifies the definition of a figure. It applies a `scale` parameter to each `psurf` in the figure. Therefore, if the figure is defined in a figure file, when you save the environment, you must write the figure to a new figure file. When you scale a figure, Jack makes a note of the fact that the figure definition has changed, so that when you subsequently write an environment file with the `write environment` command, it will prompt you as to whether you want to write the figure to a new figure file. If you choose not to do so, you will loose the effect of the scaling operation.

TUTORIAL 11.2: SCALING A FIGURE

Read in the puma robot arm from the file `puma560.fig`.

1. Execute the `scale figure` command
2. Select the robot
3. Enter a scale value of $(2, 2, 2)$
   You can also enter a scalar value of 2 by erasing the vector and replacing it with a number.
   The figure will become twice as large.
4. Execute the `scale figure` command again
5. Enter a scale value of $(0.75, 1.5, 0.90)$
   This scale factor is applied over top of the previous scale factor, so that the result is that the figure is scaled by $(0.75, 1.5, 0.90)$ from its original shape.
   Note, however, the odd shape of figure. The scaling takes place along the local axes of each segment, so each segment does not necessarily scale in the same global dimension.
6. Execute the `write environment` command
   The command will prompt you to enter the name of an output file. It will then issue a message:
   ```
   figure 'puma560' modified: write to new figure file? (C if no): puma560.fig
   ```
   This is asking you for the name of a figure file to which to write the scaled figure definition.
   The default value is the current file name. If you select this by hitting ENTER, you will overwrite the existing definition if it came from the current directory.

It is also possible to scale a figure along the global axes, using the `scale figure` globally. With a scale value equal in all directions, this does basically the same thing as the `scale figure` command. However, if you scale differently in each dimension, it scales each segment along the global axis. Note that this means that the posture of the figure affects the scaling!

This command modifies the actual `psurfs` as well as the figure itself. Therefore, when you write an environment file, Jack will ask you to write a new figure file, and it will then prompt you to write new `psurfs`. If the `psurfs` came from an archive, it will automatically put the `psurfs` in an archive.

TUTORIAL 11.3: SCALING A FIGURE GLOBALLY

Use a fresh puma model for this example, not the scaled one from the previous example.
Figure 11.2: The robot after scaling commands.

1. Execute the scale figure globally command
2. Select the robot
3. Enter a scale value of $(0.75, 1.5, 0.9)$
   The figure will become taller and thinner.

Figure 11.3: The scaled robot again.

For fun, try adjusting some joints and doing the same thing again. The figure is warped in its current stance.

4. Execute the write environment command
   The command will prompt you to enter the name of an output file. It will then issue a message:

   "figure 'puma560' modified: write to new figure file? (C if no): puma560.fig"
11.3 Controlling Degrees of Freedom

The degrees of freedom of a joint may be changed with the set joint type command. This command prompts you to enter a degree of freedom expression in the peabody language. It sets the degree of freedom for the joints accordingly. See Section A.3.5.1 for the syntax of the degree of freedom expressions.

Joints may also be "frozen". A frozen joint may not be manipulated. More importantly, a joint which is frozen will not be changed by the inverse kinematics algorithm which solves the constraints. Joints may be frozen with freeze joint, and "unfrozen" with thaw joint.

11.4 Renaming Objects

Elements of the peabody environment may be renamed with the commands rename figure, rename segment, rename site, and rename joint.

11.5 Deleting Figures

You may delete figures with the delete figure command. If the figure has any constraints attached to it, they are deleted as well. You may delete all of the figures in the environment with the delete environment command. Technically, this doesn’t delete the camera figures, but it has to exist anyway.

11.6 Creating Articulated Figures

Jack has a set of commands for creating articulated figures from scratch. These commands allow you to perform operations on objects to create joints and merge segments together. These operations are particularly useful in articulating object models brought into Jack from other CAD systems. Typically, the database conversion process yields a model in Jack with no sensible moving parts, but these commands allow you to easily articulate such models.

11.6.1 Creating Joints

You can create articulated figures out of primitive objects with the commands create joint between sites and create joint between segments. These commands merge figures together by creating a joint between them. They are designed to merge figures together rather than create joints between existing segments in the same figure. Remember that peabody is not capable of representing closed loops of joints. This behavior can be simulated only through the use of constraints.

The command create joint between sites prompts you to pick two sites and it then creates a joint between the two. The figure of the second site will snap to position so that the second site lies over the first one.
The two figures will be merged into one. This command requires that you select sites which already exist. Therefore, you should use this command only if you have sites at the proper joint centers.

The command create joint between segments is an easier way of assembling articulated figures out of joints. This command automatically creates the sites necessary to properly locate the joint center. To use this command, first position the two segments with the proper relative location. The command will then prompt you to specify the joint center by moving a homogeneous transform there. When you terminate by hitting ESCAPE, Jack will automatically create sites on each of the segments, an a joint between them, such that the segments do not change location. The joint will connect the two from their current location.

فكّر في 11.4:إنشاء أشكال متصلة

1. قراءة وحبل وهرم

   هذه تأتي من الملفات cube.pss و pyramid.pss. أضف الهرم بحيث يكون على قمة الهرم.

2. تنفيذ 명령ة create joint between segments

   قم بإختيار الهرم كالجزء الأول والكعكة كالجزء الثاني. يعني أن الهرم سينتمي إلى الكعكة.

   Jack سيعمل بإحالة مركز الحزام. هذا يعني أن الجسمين لن يتحركاً أثناء إعداد الحزام. لهذا مهمته أن يضع الأشكال في موقع لهما قبل إنشاء الحزام.

3. إرسال مركز الحزام إلى وسط الجانب العلوي للكعكة

   اضغط على ESCAPE عند الانتهاء. Jack ستطلب منك إدخال نوع الحزام.

   هذا الإخبار يشير في自由도 الحزام، أي كيف يتأثر أو يتحرك. هذه القياسات تعتمد على آليات R وت. T

4. إدخال R(y)

   هذا سيفعل الحزام يدور حول محور الـ y. عند إدخال هذا النوع، الحزام يصبح مكرسًا.

5. حرك الحزام الجديد

   الكعكة يجب أن تدور حول قمة الكعكة.
11.6. CREATING ARTICULATED FIGURES

It is possible to create joints with no degrees of freedom. To do this, hit \textasciitilde C instead of entering the joint type. A joint with no degrees of freedom describes a fixed relationship between two segments.

You can also create joints between segments in the same figure, but this means that an existing joint will be deleted to prevent the formation of a closed loop. This is a good way of redefining the articulation of a figure that has been converted from an external CAD system. Such figures usually do not have joints placed in the proper location, but they sometimes do have segments collected into joints. Provided the segments are in a reasonable location relative to each other, you only need to use \texttt{create joint between segments} to properly define the articulation.

11.6.2 Merging Segments

The command \texttt{merge segments} combines two segments into one. One of the segments will be deleted and its psurf inserted into the other. The resulting static appearance of the figure will not change, although its articulation will be different.

This command will first prompt you to select a segment. The first segment you select is the one which will be deleted. After selecting this segment, the command will prompt you to select a second segment. The second segment is the one into which the first one will be inserted. The command will then prompt you to enter a new psurf file name. The default name will be that of the second segment. \textit{Jack} will write the combined psurf to this new file and then transfer the first segment and all its sites to the second.

The command \texttt{split segment} does exactly the opposite: it divides a segment into several segments by separating its psurf into disconnected pieces. A disconnected portion of a psurf is a set of nodes, edges, and faces which is complete unconnected to the rest of the psurf. If the psurf has only one connected component, then this command does nothing. If the psurf has several disconnected components, \textit{Jack} writes them to individual psurf files, and it creates corresponding segments within the original figure to go along with each one.

11.6.3 Deleting Joints and Segments

The commands \texttt{delete joint} and \texttt{delete segment} do exactly what they imply, except that in the process they split a figure in two.
11.7 Joint Motors

Joint motors are functions that allow the user to continuously "exercise" a degree of freedom for a particular joint. The joint will oscillate or rotate through its entire range of motion without user intervention. This is a useful tool while performing analyses, or for simply making an environment more interesting.

When a joint is motorized, it is automatically frozen. This makes it impossible for the user to interactively manipulate the joint. The joint can be used in a constraint, but since it is frozen it will not be changeable by the constraint solver.

There are several commands for the creation, deletion, and control of joint motors. These commands are on the motor menu:

- **motorize joint**: This command creates a joint motor. It will first prompt the user to select a joint, then a degree of freedom for that joint. Then the rate of motion (specified in simulation frames per cycle) and the type of motion must be given. There are two types of motions: constant, which will cause the d.o.f. to rotate in one direction only, or reciprocal, which will result in an oscillating motion between the upper and lower limits of the joint.

- **change motor**: This will prompt the user to pick a motorized joint, then prompt for a degree of freedom if the joint has more than one motor assigned to it. The motion type and rate can then be changed.

- **turn motor on**: This will turn the selected motor on. Motors are on when created, but can be turned off by the user.

- **turn motor off**: This turns the selected motor off. The motor function will still be present, but it will not change the joint angle. The joint is still frozen and cannot be manipulated.

- **turn all motors on**: Turns all motors in the environment on.

- **turn all motors off**: Turns all motors in the environment off.

- **delete motor**: Deletes the selected motor from the environment. If this is the last motor associated with the joint, it will be thawed. Additionally, if a joint is deleted, any motors associated with it will automatically be deleted.

- **delete all motors**: Deletes all motors in the environment.

11.8 Segment Physical Properties

You may change the mass or density of a segment in Jack. This is useful during strength analysis. The commands for doing so are on the edit \(\Rightarrow\) physical menu:

- **set segment mass**: Sets the mass of the segment.
- **set segment density**: Sets the density of the segment.
The command set segment mass allows you to set a segment's mass property. The density will be derived from the volume of the geometry and this mass. The command set segment density allows you to set a segment's density in grams per centimeter cubed. The mass will be derived from the volume of the geometry and this density. The density of water is 1 gm/cm³.
Chapter 12

Surface Attributes and Light Sources

*Jack* represents the colors and material properties of objects through the notion of *surface attributes*. A surface attribute represents a type of material. Each polygonal face of each *psurf* associated with the segments in the environment has a surface attribute associated with it. Rather than representing the complete set of material properties independently for each face, each face has an attribute reference, and many faces will generally refer to the same attribute. Typically, all faces of a single *psurf* refer to the same attribute. This is in effect saying that the faces of a *psurf* are all composed of the same material, which is generally the case.

*Jack* represents light sources as special types of peabody segments. The position of the segments in the environment define the location of the light sources. Light sources have additional properties as well, such as color and intensity.

*Jack* takes its lighting information from the light source objects in the environment if any exist. However, when you make an object or a window shaded and you have *not* defined any light sources, then *Jack* simulates a light source at the camera location. This type of shading works well when you just wish to have objects shaded, as opposed to incorporating light sources into your geometric environment or simulation.

### 12.1 Attribute Names

*Jack* creates surface default attributes for objects and chooses their colors from a rather random palette. However, *Jack* automatically gives the attributes names which describe their colors, such as *dark-green* or *pink*. This makes it more intuitive to associate attributes with peabody segments.

Because *Jack* expects that the names of attributes correspond to the attribute’s appearance, whenever you change a parameter of the attribute interactively, *Jack* will prompt you to give the attribute a new name. For example, if you create an object and *Jack* colors the object dark green, but you really wanted it to be pink, when you adjust the attribute’s color using the *set attribute color* command, *Jack* will prompt you to enter a new name, at which time you should enter *pink*.

### 12.2 The Jack Color Palette

*Jack* gives you some control over the palette it uses for new objects. Each user may keep a file in his home directory called ‘*~/.jack.colors*’ which lists color values and names. The format of the file has each line with a red, green, and blue value, normalized between 0 and 255, followed by a character string name. The name must be a single word, although it may contain underscores. *Jack* uses the attributes in the list in the order in which they appear. After all attributes have been exhausted, it reverts to the internal palette.

Here’s an example ‘*~/.jack.colors*’ file:

```
255 97 0 lightbrown
221 160 221 plum
134 26 26 darkfirebrick
47 79 79 darkslategray
0 255 127 springgreen
```
12.3 Picking Attributes Interactively

When Jack requires that you select a surface attribute for a particular operation, such as to change its parameter values, it allows you to pick it interactively by selecting a face which has this attribute associated with it. Jack also lets you enter the attribute name from the keyboard or pop-up menu, just as with picking other types of peabody objects.

12.4 The IRIS Lighting Model

The shading facility in Jack uses the lighting model of the Silicon Graphics IRIS Graphics Library. This lighting model performs gouraud shading of objects with ambient, diffuse, and specular material properties using the phong shading model. It models multiple colored light sources of varying intensities. Jack represents all the information necessary to control the various aspects of material properties in surface attributes and light sources. For the details of how the shading is actually performed, consult the IRIS Graphics Library User's Guide.

The basic lighting model equation is shown in Figure 12.1. Each color value C in the lighting model equation is a triplet of red, green, blue color values. The product of two colors is formed by multiplying each corresponding component.

\[
C_p = C_{me} + C_{sa}C_{ma} + \sum_l C_l \frac{1}{K_{sdaf} + K_{sdau}D_{lp}} \left( C_{md} [N_{pl} \cdot N_p] + C_{ms} \{N_l \cdot N_p\}^{E_{ms}} \right)
\]

- \(C_p\) the color of the surface point \(p\)
- \(C_{me}\) color of material emission
- \(C_{sa}\) scene ambient color
- \(C_{ma}\) ambient color of surface attribute
- \(C_{fa}\) ambient color of light source \(l\)
- \(K_{sdaf}\) fixed scene attenuation factor
- \(K_{sdau}\) variable scene attenuation factor
- \(D_{lp}\) distance from light source \(l\) to \(p\)
- \(C_{md}\) diffuse color of surface attribute
- \(N_{pl}\) direction from \(p\) to light \(l\)
- \(N_p\) surface normal at \(p\)
- \(C_{ms}\) specular color of surface attribute
- \(N_{b}\) bisector of \(N_{pl}\) and view vector
- \(E_{ms}\) glossiness of surface attribute

Figure 12.1: The IRIS Lighting Model
12.5. Attribute Parameters

The appearance of objects in the environment depends upon the parameters of their surface attributes, as well as the parameters of the light sources in the scene. There are four basic parameters of the surface attribute:

- **ambient**: The color of the surface when it is not illuminated by a light source.
- **diffuse**: The color of the object when it is illuminated by a white light.
- **specular**: The color of the specular highlights of the object.
- **glossiness**: An integer exponent describing the specular scattering of the surface. The lighting model simulates specular highlights using a cosine of the angle between the viewing direction and the reflected light ray, raised to the glossiness exponent. If this exponent is large (approximately, 50), the specular highlights are small and focused. If the exponent is small, the highlights are broad. If the exponent is 0, the object is not specular.

12.5.1 Ambient and Diffuse Parameters

The cosine law described in Figure 12.1 explains that the color of a surface depends upon the angle at which it is oriented relative to the light source. When it is oriented directly towards a light source, it is illuminated with diffuse light and the appearance of the surface comes from the diffuse parameter. When the surface is oriented away from a light source, no light shines on it and it is illuminated by ambient light. The appearance of the surface of the comes from the ambient parameter. Normally, the ambient and diffuse parameters are the same underlying colors but with different intensities. Therefore, the surface has one basic RGB color value, and the ambient and diffuse terms are scalar values which describe intensities of this color. An ambient value of 0.2 and a diffuse value of 0.8 are typical.

When Jack writes an attribute definition to a peabody file, it writes it as an rgb color and scalar ambient and diffuse terms. The rgb term is scaled to its brightest intensity and the ambient and diffuse terms are printed so as to reflect the attribute's actual appearance.

**Tutorial 12.1: Changing Attribute Colors**

For this example, read in the chain from the file chain.fig. This will demonstrate changing the color of the entire figure.

1. Execute the set attribute color command

   This command is on the attribute menu which is a sub-menu of the edit menu.
CHAPTER 12. SURFACE ATTRIBUTES AND LIGHT SOURCES

The commands set attribute ambient and set attribute diffuse allow you to give the corresponding parameters of the surface attribute using separate RGB values.

As is quite obvious from the lighting model equation in Figure 12.1, the behavior of the ambient and diffuse terms actually depends upon the color of the light source which is illuminating the surface. The behavior described above only occurs when the light is white, which is the default. If a light has a color other than white, it modulates the ambient and diffuse components of the surface color by its own color, on a red, green, and blue basis.

12.5.2 Specular Highlights

Specular highlights are the shiny parts of the surface. These occur when light reflects off of a surface towards the view. The color of the specular highlights is specified by the specular term, which is white by default.
The distribution of the highlight is determined by the glossiness parameter, which is 0 by default, which means there is no specular highlight. This value is an exponent of the specular cosine term in the lighting equation. A value of 30 is typical for a shiny surface. The commands set attribute specular and set attribute glossiness allows you to set these parameters.

**TUTORIAL 12.2: SETTING SPECULAR HIGHLIGHTS**

For this example, read a torus from the file `torus.pss`. This object is good for illustrating specular highlights because it has regions of different curvature.

1. **Execute the command** set attribute glossiness
   
   The command will prompt you to pick an attribute. Pick any face in the figure.

2. **Enter a glossiness value of 30**
   
   The appearance of the object will not change until you change the specular parameter.

3. **Execute the command** set attribute specular
   
   The command will prompt you to pick an attribute. Pick the same one. The command will then allow you to specify the specular component using an RGB meter window, just like ordinary colors.

4. **Enter a specular value of** (1,1,1)
   
   The figure should turn brighter in some places. You must move the view around to get a better feel for what the specular highlights really look like, because the highlights are highly dependent upon the viewing direction.

### 12.5.3 Making Objects Appear Smooth

Specular highlights appear more "realistic" on smooth surfaces. Most of the surfaces in *Jack* are flat because they are modeled as polyhedra. It is possible to make interpret the polygons of a psurf as a mesh-like approximation of a smooth surface. In this case, the faces are really just artifacts of the model and not a true geometric property. In this case, the normal to the surface should vary across the interior of the face. This is called *phong shading*. The IRIS performs *gouraud* shading by evaluating the lighting model at each vertex and interpolating the resulting color across the interior of the polygon.

Each face in the psurf has a flag defining whether it is **smooth** or **flat**. The details of the specification of the smoothness parameter in the psurf files are described in Section 12.5.3. If a face is smooth, it "shares"
its surface normal with neighboring faces. Jack computes the normal at each vertex as an average of the normals of each adjacent faces. If the polygons approximate a mesh, then the result is that the object will appear smooth.

The commands make face smooth and make face flat provide a means of changing this information interactively. The commands make segment smooth and make segment flat change the flags for every face in a psurf. This parameter is a geometric property of the psurf, so changing it interactively implies that you are altering the definition of the psurf. Therefore, if you want to save the changes, you must write out the psurf definition. The write environment command prompts you for this automatically.

**TUTORIAL 12.3: MAKING OBJECTS SMOOTH**

This example demonstrates the use of the smoothness parameter. This is best visualized with a simple example shape, such as the torus. The torus is a good example because it has many regions with different curvature.

1. Read in a torus from the file torus.pss
   The torus is approximated with a lot of little polygons.

2. Make the window shaded
   Notice that the torus is smooth, even though it is composed of little flat facets. Jack normally interprets objects as polyhedra, so each polygon is considered independent from its neighbors. However, the torus is smooth by default. The individual faces of the torus should not be detectable.

3. Execute the command make segment flat
   Select the torus. Now, the torus should appear as small flat facets. Notice how the specular highlights seem to be either "on" or "off" for each face. Each face is a constant color.

**TUTORIAL 12.4: MAKING OBJECTS PARTIALLY SMOOTH**

This example demonstrates the use of the smoothness parameter applied only to some of the faces of a psurf.

1. Read in a cylinder from the file cylinder.pss
   The cylinder is approximated with narrow polygons around its side and two large polygons for the top and bottom.
2. Make the window shaded
3. Set the glossiness factor for the cylinder to 30
4. Execute the command make segment smooth
   Select the cylinder. Now, the cylinder will appear mostly as a blob. This is because the
   surface normals around the edges are attempting to blend the top and side, something that
   cannot be done very well. With this arrangement of faces, we want the top and bottom to
   be completely flat.
5. Execute the command make face flat
6. Select the top face
   The top face should now appear a constant color and there should be a crease around the
   top edge.
7. Do the same thing to the bottom face
8. Write out an environment file with the write environment
   This will prompt you with a message:
   
   segment 'cylinder cylinder' psurf modified: write psurf to file? cylinder.pss

9. Enter the name cyl.pss
   Remember that you can use the arrow keys to move through the string to change it.
   This will write the "new" cylinder with the modified smoothness parameters to the file
   cyl.pss.
10. Exit Jack and print the file cyl.pss
    The last part of the file looks like:

    ```
    35 37 39 36 [smooth] [attribute 0];
    37 39 40 38 [smooth] [attribute 0];
    39 1 2 40 [smooth] [attribute 0];
    2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 [flat] [attribute 0];
    39 37 35 33 31 29 27 25 23 21 19 17 15 13 11 9 7 5 3 1 [flat] [attribute 0];
    ```

    Notice the smooth and flat parameters which indicate which faces have which property.
12.6 Associating Attributes with Surfaces

The commands discussed so far affect the color of attributes, but they do not control which segments have which attributes. This is possible with the other commands on the attribute menu.

12.6.1 Segments with Several Attributes

Normally, all faces in a psurf have the same attribute. It is also possible to have faces within a psurf refer to different attributes interactively.

12.6.1.1 Associating Surface Attributes With Faces

The important thing about attributes is how they are associated with the individual segments. The surface attributes actually belong to the psurf, not the segment, and the psurf provides a mechanism for associating a different surface attribute with each face. The syntax of the psurf file, described in Section A.4, allows an attribute index to be associated with each face. This is a small integer which refers to the elements of a set of attributes which will be associated with the psurf in the peabody grammar. In the peabody grammar, a list of attributes may be associated with each segment. The attribute indices of each face in the psurf refer to attributes in this list.

```plaintext
attribute brown {
    diffuse = (0.48,0.26,0.00);
}

attribute gray {
    diffuse = (0.95,0.84,0.62);
}

segment x {
    psurf = "thing.pssU;";
    attributes = (brown, gray);
    ...
}
```

The psurf file `thing.pss` may look like:

```
 0.0  0.0  0.0
 0.0 100.0  0.0
100.0 100.0  0.0
100.0  0.0  0.0
 0.0  0.0 100.0
 0.0 100.0 100.0
100.0 100.0 100.0
100.0  0.0 100.0
...
1 2 3 4 [attribute 0];
1 4 8 5 [attribute 0];
3 7 8 4 [attribute 0];
1 5 6 2 [attribute 1];
5 6 7 6 [attribute 1];
2 6 7 3 [attribute 1];
```

In this example, the psurf associated with the segment `thing` will have faces 1, 2, and 3 composed of the attribute `brown`, since they reference attribute 0 of the segment’s attribute list, and faces 4, 5, and 6 will be composed of the attribute `gray`, since they reference attribute 1 of the segment’s attribute list. Using this method, the same psurf may be assigned different surface attributes when it is used as the geometry of different segments.

In the absence of attribute information in a peabody file, `Jack` creates attributes to go along with each segment. When you read a figure and do not specify any attribute information, `Jack` will create as many attributes as it needs, but it will assign the same attribute list to each segment. Therefore, all segments in the figure will have the same set of attributes. When `Jack` creates a surface attribute, it assigns it a more or less random color, just to keep things exciting.
Although attributes are associated with the individual faces of the psurf, there are some very good reasons why it is a good idea to give all faces the same attribute. *Jack* draws the edges of each segment using the color of the first attribute associated with the psurf. What you see is edges, not faces, so it is difficult to see which faces have which attributes.

There is another more philosophical reason: the notion of a *surface attribute* is intended to connote a material of a certain type, and a single segment should usually consist of a single type of material. If an object is composed of two different types of material, then it is probably a *figure*, composed of two isolated segments. You do not have to think of joints as things which get frequently adjusted: a joint may serve just to link two things together. There are exceptions to this rule, of course, so the feature is supported, but you may find yourself becoming very confused if you try to create one huge segment with many parts with different attributes.

The command *give face new attribute* creates a new attribute and associates it with a single face in a psurf. The attribute index of the selected face is changed to reflect the new attribute. The index will generally be one greater than the maximum of the previously existing attribute indices. The command *give face old attribute* allows you to select a previously existing attribute and associate it with a single face in a segment. As with *give segment new attribute*, the attribute index of the selected face is changed. If the attribute already belongs to other faces in the psurf, the index is changed to be the index of the other faces with that attribute. If not, it is incremented as with *give face new attribute*.

The term *csurf* applies to a connected subcomponent of a psurf. The commands *give csurf new attribute* and *give csurf old attribute* set the attribute associations for all faces which are connected to a selected face. This provides a means of identifying the connected components of a psurf.

### 12.6.2 Setting Attributes for Entire Segments

The command *give segment new attribute* creates a new surface attribute and associates it with every face in a psurf. The attribute index of every face in the psurf is set to 0, specifying that there is a single attribute. The command *give segment old attribute* allows you to select a previously existing attribute and associate it with every face in a segment. As with *give segment new attribute*, every attribute index of the psurf is set to 0.

**♠ TUTORIAL 12.5: PREPARING FOR SUMMER**

Create a human figure for this example. Normally, the human figure has an elbow length shirt. We'll give him a tank top.

1. Execute the command *give segment old attribute*

   *Jack* prompt you to pick an attribute. Pick the *flesh* attribute by pointing at his lower arms or hands. *Jack* will then prompt you to pick another segment.

2. Pick the left upper arm.

   The upper arm will change to the flesh color.

3. Execute the command *give segment old attribute* again

   Now, since we want to repeat the process for the right arm, we must select a flesh-colored segment again. However, since it was the previous attribute we picked, we can simply hit the exclamation point key `!` and it will automatically reuse the previous one.

4. Pick the right upper arm

   The figure is now wear a tank top!

5. Now execute the command *give segment new attribute*

   Pick the right hand of the figure. The hand will change color, using a color from the random palette. If you don't like gloves of this color, you can change it with *set attribute color*.

### 12.7 Using Light Sources

The illumination of objects a shaded *Jack* window is controlled by light sources. Light sources are described in Section A.3.8, and they may be created interactively with the *create light source* command.
If no light sources have been created, Jack illuminates the window with a fake light source located at the center of projection, or the camera location. This type of illumination is quite nice for most applications.

If you have created any light sources, then Jack does not contribute same the illumination from the center of projection, and all of the illumination comes from the defined lights. This is good if you want to use special care in positioning the lights, but keep in mind that when there is a single light source which is not near the center of projection, there will usually be large portions of the visible surfaces which are oriented away from the light and thus illuminated with only ambient light, which is independent of the orientation of the surface. This makes the shading difficult to interpret.

12.7.1 Light Source Parameters

The illumination of objects in the scene depends upon the color of the light source. By default, lights are white, but their color may be changed with the set light color command. The color of the light models both the light's emitted spectrum and its intensity. To make the light dimmer, change the intensity of its color.

Lights also have an ambient light factor, which attenuates the ambient light property of the surface attributes. The color and intensity of the ambient light reflected from a surface is the product of the ambient light properties of the surface attribute and the light source illuminating it. The ambient light parameter of a light source may be set with the command set light ambient.

The IRIS illumination model represents lights which are either local to the scene or located at infinity. For a local light source, the light direction vector used in the light model evaluation is different for each point in the scene. For lights located at infinity, the same light vector may be used for the entire scene, which makes the light model much more efficient, but the lights do not cast rays in all directions. You can affect this parameter with the commands make light local and make light infinite.

Jack models light sources as peabody segments, and the global position and orientation of the light source segment determines the placement of the light. When a light is local, the position of the light defines the origin of the light, just like a light bulb. The orientation of the light is not significant. When a light is infinite, orientation of the light source defines its direction, but the location of the light is irrelevant. You may move the light around without effect. The light is effectively located at infinity along the direction of the light source.

The default light used in the absence of light source figures is an infinite light source directed along the line of sight.
12.8. REAL-TIME TEXTURE MAPPING

12.8 Real-Time Texture Mapping

Texture mapping in Jack no longer needs to be accomplished through ray tracing (see Section B.1). The interactive texture mapping facility allows one to texture map polygons in real time. By specifying a texture map (an image file), a texture map face, and a face (or list of faces) to map the texture to, textures can be visualized without the need of sending the environment through the ray tracer. One of the most powerful uses of texture mapping is to add visual detail to a scene without adding extra geometry (polygons) to render. For example, a complex control panel that a human is working on could be modeled as a psurf with two or three faces. Then, an image of the control panel could be texture mapped onto those faces, giving the illusion that you've modeled the whole thing.

Note: Texture mapping through Jack is only available on SGI machines with graphics systems that support it (i.e. All Indigo systems, Elan graphics, VGX, VGXT, and Reality Engine). If texture mapping is not supported on your machine, all the following commands will notify you of that fact.

All texture mapping commands are on the texture menu, a sub-menu of the edit menu.

The texture map face can be specified in one of two ways. First, the command make texture plane can be used. This function determines the current line of sight (from the global camera coordinates), and creates a new square polygon face in this line of sight. All the faces in the environment behind this plane will then be mapped with the texture map face, this new polygon then can be selected as the appropriate face. The other way to specify the texture map face is to choose an existing polygon. For instance, if a texture map is to be completely mapped to a single face on a figure one could choose the texture map face to also be the mapping face. (Note: The texture map face chosen, however, must have only FOUR vertices.) the advantage of creating a new texture map plane is that this new plane can be manipulated before specifying it as the actual texture map face. Scaling, translating, and rotating are all allowed for correct positioning of the plane over the correct faces in the environment.

A texture can be mapped to a figure or segment in two different ways. The texture can either be mapped to all the visible faces of the figure or segment behind the texture plane, or it can be mapped to all the faces of the specific figure or segment behind the texture plane. To choose only visible face mapping choose auto zmap texture to figure or auto zmap texture to segments. (NOTE: The visible face calculation is performed for each segment and is window size dependent. If the figure selected has many segments (i.e. human) the Jack window size should be decreased considerably until calculation is complete.) In order to map to all faces under the texture plane use either the command auto map texture to figure or auto map texture to segments. (NOTE: these two commands assign the given texture map to all the faces of the figure/segment without...
checking to see if they are within the texture map face coordinates). If mapping to segments is chosen Jack will ask you to click on all the segments to map, until ESC is pressed.

The face (or list of faces) to map a texture to can also be specified. The command map texture to faces allows the user to specifically choose which faces the texture map will be mapped to. Each face receives the appropriate section of the texture map, which is determined by the parts of the texture map face that cover the mapping face. The user, similar to segment mapping, can choose as many faces as desired, signaling the end of desired faces by pressing the ESC key.

Textures can also be turned on or off for individual attributes, segments, figures, or the entire environment through the appropriate commands on the edit ⇒ texture menu. This allows textures to stay active in the environment while deactivating particular textures for particular scenes.

In order to have multiple textures on a single figure, each face or segment which is to have a different texture must have its own individual attribute. This can be accomplished by the give face new attribute or give segment new attribute commands.

When asked to input the texture file name from any of the above mapping commands, the full name (including path) of the file must be given. If the map is compressed the .Z must also be included with the file name. A texture can when be specified as single or repeated. This means that the faces will only receive one copy of the texture map, regardless of size, or the faces will receive repeated texture maps if the mapping face is larger than the texture face.

If the texture mapping plane does not encompass an entire face, Jack will try to pull the last averaged pixels all the way over the entire face. This can be changed with the pad texture borders on options. This option leaves the border as blank, whereby the rest of the mapping face not covered by the texture plane will remain as it. This command must be executed before the mapping of the texture to be padded (i.e. before the texture is read in).

An existing figure, segment, or face can receive a new texture by the set attribute texture command. This command prompts the user for the new texture map name, and the face to change the attribute of. All faces with this attribute will then be assigned the new texture map.

When a texture is specified as repeated or the texture is tessellated the texture should be rendered using subdivision to prevent distortion. This can be accomplished by the command texture subdivide on. This command can be used for any texture that seems to have distortion (Note: This function slows rendering due to the computations of subdividing).

The default for all the above toggling commands is OFF.

**TUTORIAL 12.6: SIMPLE TEXTURE MAPPING**

1. Read in the file cube.pss
2. Execute map texture to faces
3. Input the texture file name mona.rle.Z
   
   *Note: you may have to include a path preceding the file name. It will be in the directory $JACK/jacklib5/textures/* See below for a set of textures included with Jack.
4. Choose single texture
5. Click on one face of the cube as the texture face
6. Click on the same face (for mapping face)
7. Hit ESC
8. Execute make everything shaded
9. Turn the cube's texture on with segment texture on
10. Select the cube as the segment

**TUTORIAL 12.7: USING A TEXTURE PLANE**

1. Read in the file cube.pss
2. Choose **make texture plane**
   Manipulate the plane using move figure or move face until the plane is covering more than one of the faces of the cube.

3. Choose **map texture to faces**
   Follow the above instructions, except choose all the faces that the plane covers as mapping faces.
   The texture should now be mapped to the selected faces, with the last pixels averaged and extrapolated out to the edges of the faces not entirely covered by the texture plane.

---

### Tutorial 12.8: Mapping a Whole Segment

1. Read in file `cube.pss`  
2. Execute **make texture plane**
   Manipulate the plane so that it covers multiple faces on the cube.

3. Execute **auto zmap texture to segment**
   Input the texture map name, choose single texture map, select the texture plane, and select the cube as the segment.

4. Execute **pad borders on**
5. Execute **make everything shaded**
6. Turn **the cube textures on**
   The cube should now look the same as it did at the end of the last tutorial.

---

### Tutorial 12.9: Change Textures

1. Execute **set attribute texture**
   Input the new texture map name.

2. Select one of the faces of the cube.
   The cube should now have a different texture, but on the same faces as the previous tutorial.

---

### Tutorial 12.10: Repeated Textures

1. Execute **make texture plane**

2. Repeat Step 1

3. Scale one of the planes using **scale figure**
   Input (0.1, 0.1, 0.1) as the scaling factors.

4. Execute **map texture to faces**
   Input `mona.rle.Z` and repeated, and choose the scaled plane as the texture face, and choose the other plane as the mapping face.

5. Make the larger plane shaded with **make segment shaded**
6. Turn the larger planes texture on with **segment texture on**
   Select the larger plane as the segment.

7. Activate subdivision rendering with **texture subdivision on**
   Select the larger plane. One hundred copies of the texture `mona.rle.Z` should now be on the larger plane. This is what the repeated option looks like on a face. Very useful for rendering bricks or floors.
12.8.1 Reflection Mapping

There are two commands on the texture menu which allow you to do reflection mapping. The command reflection map segment prompts for an image file and a segment, and will apply that image as a reflection or environment map. The command reflection map figure does the same for an entire figure. Of course, you must execute one of the * textures on commands described above to see the results.

12.8.2 Image Formats and Textures

Jack has an extensible mechanism for handling different image file formats. Internally, Jack uses the Utah Raster Toolkit RLE format for reading and writing images (reading: for texture maps; writing: when doing things like write window image). Externally, Jack can read and write several different image formats, which are defined in the file $JACK/jacklibS/jack.imagefilters.dat. This file defines input and output filter commands to support each image format.

Currently supported formats are:

<table>
<thead>
<tr>
<th>Type</th>
<th>Extension</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utah Raster Toolkit</td>
<td>.rle</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Utah Raster Toolkit (Compressed)</td>
<td>.rle.Z</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SGI RGB Format</td>
<td>.rgb</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>SGI RGB Format (Compressed)</td>
<td>.rgb.Z</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>GIF</td>
<td>.gif</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>GIF (Compressed)</td>
<td>.gif.Z</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>PostScript</td>
<td>.ps</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>PostScript (Compressed)</td>
<td>.ps.Z</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 12.6: Supported Image File Formats

For each format supported, there are four lines in the filter file which define the entry:

1. Name of the format.
2. The extension of files in this format.
3. The input command. The input command must take the input filename as an argument (%s represents the filename) and produce a URT rle file on its standard output.
4. The output command. The output command will be fed a URT rle file on its standard input and produce an image file of the correct type (%s represents the filename). If either input or output is not available, this is signaled by placing the special command not available.

An example entry for compressed GIF files looks like:

```
GIF (Compressed)
.gif.Z
zcat %s | giftorle
rletogif | compress | cat > %s
```

There are a set of sample textures in the $JACK/jacklibS/textures/ directory. The next few figures show iconic versions of some of these textures. An excellent tool for browsing the texture images is the xv, available from many anonymous ftp sites.¹

¹xv was also written at the University of Pennsylvania, by John Bradley at the GRASP Laboratory. The images for this manual were converted to PostScript using this program.
Figure 12.7: A Few Reflection Maps
CHAPTER 12. SURFACE ATTRIBUTES AND LIGHT SOURCES

Figure 12.8: A Few Plants and Trees

bush1.rgb  bush2.rgb  flowers.rgb  flowers2.rgb  maple.rgb  maple1.rgb

maple1s.rgb  tree1.rgb  tree12.rgb  tree12m.rgb  tree2.rgb  tree3.rgb

tree4.rgb  tree5c.rgb  tree5s.rgb  trees15.rgb  vase2.rgb
Figure 12.9: A Few Clouds and Fractals
Figure 12.10: A Few Simple Textures

- balmoral.rgb
- brick.rgb
- bumps.rgb
- camouflauge1.rgb
- camouflauge2.rgb
- concentric_cos.rgb
- cosine.rgb
- globe.rgb
- grass1.rgb
- guarana.rgb
- kandplate.rgb
- kitchencabinets.rgb
- leopard.rgb
- luna.rgb
- panel.rgb
- paper.rgb
- pine.rgb
- redboarpet.rgb
- redmar.rgb
- shield.rgb
- smudgenap.rgb
- smushbrick.rgb
- snow.rgb
- stone.rgb
- wood.rgb
- woodfloor.rgb
- wstar.rgb
Chapter 13

Manipulating Geometric Shapes

*Jack* has several features for describing and manipulating geometric objects. These include primitive operations on psurfs which allow their geometry to be altered, as well as a precise geometric representation through *cad* objects.

13.1 Modifying Psurf Geometry

13.1.1 Moving Nodes, Edges, and Faces

The commands move node, move edge, and move face on the move menu allow you to move nodes, edges, and faces of psurfs. Each command prompts you to pick the item you want to move and then move it, using the same movement technique as move figure.

This operation modifies the psurf, so it must be written to a new file to save the changes. The write environment will detect that psurfs have been modified and automatically prompt you to write them out.

Recall that as you are moving nodes you may use the snapping mechanism. This provides a good way of aligning the geometry of two objects. For example, if you have two segments which are connected by a joint and you want the shape of the two segments to meet snugly, move each node of one of the shapes and snap it to a corresponding node of the other.

13.1.2 Splitting Edges and Faces

The commands split edge and split face allow you to add elements to a psurf. The split edge command splits an edge into two by creating a new node. The original location for the node is at the midpoint of the edge, but the command then allows you to move it. The faces on the left and right side of the edge get a new vertex referencing the new node.

The command split face prompts you to split a face by creating a new edge between two of its vertices. The command begins by prompting you to pick a face. It then asks you to pick two of the vertices in the face. The nodes you pick must belong to the face. *Jack* will then split the face into two faces by creating an edge between the two nodes.

The command divide face at vertices automatically splits a face into several faces by creating a new node in the center of the face and then splitting the face by connecting the new node to each of the vertices of the original face. The command divide face at edges automatically splits a face into several faces by creating a new node in the center of the face and then splitting each of the edges of the face with a new node in the middle. It then splits the original face by connecting the center node to each of the midpoint nodes. The result is that rectangular faces are divided into four smaller rectangles.

The command triangulate face divides a face into triangles by splitting it between vertices. The pattern of the splitting is arbitrary. If you are concerned about how the face is split, you should split it explicitly with split face.
13.1.3 Creating Nodes and Faces

The command create new psurf creates a psurf with a single node. It creates a new figure with a single segment. From this basic unit, you can build up a more complex shape by creating more nodes and faces.

The command create node creates a new node on an existing segment. It allows you to position the node in space. The node will not be connected to anything else in the psurf. Because of this, this command automatically turns on the display of the nodes of the psurf. The nodes are drawn as small cross-hairs.

The command create face allows you to create a face by selecting a set of nodes. The command prompts you to pick nodes, one after the other. After you have selected the nodes for the new face, terminate the picking by hitting ESCAPE. The new face will then appear.

The command make face into solid operates on a single face not connected to anything else. It takes the face and duplicates it and then connects each of the corresponding edges of the two faces with a quadrilateral. It then prompts you to move the new face. This operation works only on a single face which has no neighboring faces. This command is useful for making prismatic objects. Note that this is similar to the extrude face command, but the extrude face command detaches the face from neighboring faces, and it does not "close off" the other end of the extrusion with a face.

13.1.4 Extruding Nodes, Edges, and Faces

The command extrude node extrudes a node from its surrounding faces. It does so by splitting the faces to which the node belongs, so as to isolate the node from its original faces. The node will then form the vertex of a set of triangles which connect it to the original faces. The command prompts you for a distance, which is the distance away from the node at which the adjacent faces will be split. The command then allows you to move the node to a new location.

The command extrude edge extrudes an edge from its surrounding faces. It does so by splitting the faces on either side of the edge and at the endpoints. The edge will be connect to the original neighboring faces with a pair of quadrilaterals, and to each original face touching the edge only at the endpoint by triangles. The command prompts you for a distance, which is the distance away from the edge at which the adjacent faces will be split. The command then allows you to move the edge to a new location.

The command extrude face takes a face and extrudes it from its neighbors. It creates new quadrilateral faces which connect the face to its original neighbors. The command prompts you for a distance, which is the distance away from the face at which the adjacent faces will be split. The command then allows you to move the face to a new location.
Figure 13.2: Extruding an edge from a cube.

Figure 13.3: Extruding a face from a cube.
13.1.5 Deleting Parts of Psurfs

Parts of psurfs may be deleted with the delete commands on the build menu. The delete node command deletes a node and the faces which contain it. The delete vertex command removes a vertex from a face but leaves the node in tact. It will prompt you to pick the node and the face from which it should be deleted. The delete edge command combines two faces by removing the edge between them. The delete face command removes a face but leaves behind the nodes which it contained. The replace node replaces one node with another. It then deletes the node which was replaced. The faces which originally contained the first node then reference the second.

13.1.6 Fixing the Orientation of Psurf Faces

Jack expects that the faces of psurfs have a consistent orientation. The orientation is determined by the right hand rule: a counterclockwise traversal by the right-hand rule should yield an outward-pointing normal. This orientation is required to be consistent because only the “normal” side of a face will be shaded correctly in Jack. The reverse side of a face is always shaded with only ambient light.

There are several commands in Jack to assist you in correcting the orientation of faces which may not be consistent. The command fix segment orientation attempts to correct the orientation of each of the faces, but this may not be possible, depending upon the shape of the object. This command is usually successful at correcting most of the faces, however.

The command reverse face orientation reverses the orientation of a single face. The command reverse segment orientation reverses the orientation of each face in a psurf.

13.2 Free-Form Deformations

Jack can deform psurf geometry using free-form deformations. A deformation is described through a lattice of points which surround an object. As the lattice points move, the object inside the lattice warps according to a blending function between the lattice points. The commands for manipulating deformations are on the deformation menu, which is a sub-menu of the edit menu.

<table>
<thead>
<tr>
<th>deformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>create segment deformation</td>
</tr>
<tr>
<td>delete segment deformation</td>
</tr>
<tr>
<td>move deformation lattice node</td>
</tr>
<tr>
<td>move deformation lattice edge</td>
</tr>
<tr>
<td>move deformation lattice face</td>
</tr>
<tr>
<td>set deformation mode</td>
</tr>
<tr>
<td>apply deformation</td>
</tr>
</tbody>
</table>

The command create segment deformation creates a deformation lattice. The lattice has a dimension along each axis. The dimension is the number of planes in the lattice along each axis. The default value which Jack provides for the lattice dimension for an object approximates the shape of the object, providing more planes along the longer dimensions. You may create deformations of any dimension, but it is best to use dimensions less than 10.

♣ Tutorial 13.1: Free-Form Deformations

Read in the chair from the file chair.pss.

1. Execute the command create segment deformation.
   Pick the chair. Jack will then prompt you to center the dimensions of the lattice.

2. Enter \((3,3,3)\)
   The lattice will appear around the object.
   Notice that there are three planes in the lattice along each axis.
13.2. FREE-FORM DEFORMATIONS

3. Execute the command move deformation lattice node.

   Select the upper corner of the lattice, next to the upper back of the chair. When you select
   the lattice node, you begin moving it.

4. Translate the lattice node upwards

   Notice how the chair deforms.

   Note that only the nodes of the psurfs are deformed. Although the psurf gets warped, the
   edges remain straight.

   The commands move deformation lattice edge and move deformation lattice face let you move edges and faces
   of the lattice. Notice that you move interior edges and faces as well as exterior ones.
13.2.1 Controlling Deformations

The deformation operation is very computationally expensive, and deforming objects which have lots of nodes and are painfully slow. It can also be slow to manipulate a simple object if the deformation lattice has a high dimension. You can control how the deformation is evaluated with the command `set deformation mode`. This command will allow you to specify when the deformation is evaluated.

[Box Tutorial 13.2: Controlling Free-Form Deformations]

For this example, read in the file `sphere.pss`. This psurf has many nodes.

1. Execute the command `create segment deformation`
2. Enter a lattice size of (2, 2, 2)
3. Execute the command `move deformation lattice node`
   It will be very slow. Stop when you've had enough.
4. Execute the command `set deformation mode`
   Select the sphere segment. `Jack` will then prompt you to enter a mode:

   ![Mode Selection Dialog]

   type:
   - interactive
   - semi-interactive
   - on demand
   - off

5. Enter the semi-interactive option
6. Execute the command `move deformation lattice node` again.
   This time, you can move the lattice node freely, although the object does not deform as you move. However, whenever you release the mouse, the deformation is evaluated again. This is semi-interactive.

   ![Deformation Demonstration]

   Figure 13.6: The deformation is not updated until you enter the command `evaluate deformation`.

7. Execute the command `set deformation mode`
   This means that the deformation will be evaluated only when you specifically request it, with the `evaluate deformation` command.
8. Execute the command move deformation lattice node again.
   This time, the deformation is not evaluated even when you release the mouse buttons.
   Move several of the lattice nodes around to displace them from their original location before
   proceeding.

9. Execute the command evaluate deformation
   The object will deform to match the lattice.

   The off option to set deformation mode turns the display of the deformation off, but does not delete it. The
   delete segment deformation command deletes the deformation.

13.3 CAD Objects

The commands on the cad menu allow you to create parameterized objects of several types. The objects
include spheres, cylinders, hollow cylinders, ellipsoids, toroids, truncated generalized cones, and gears. These
objects are represented internally by an exact, parameterized boundary equation. They are drawn in Jack
much like ordinary psurfs, using line segments and facets. However, since they are parameterized, their
parameters may be changed to change their appearance. It is also possible to generate ordinary psurfs out
of them.

The following types of objects are available, along with their parameters.

cylinder
   1. radius
   2. height
   3. origin
   4. resolution along radial axis

sphere
   1. radius
   2. resolution in xy-plane
   3. resolution in xz-plane
   4. origin

hollow cylinders
   1. radius of outer cylinder
   2. radius of inner cylinder
   3. height of inner and outer cylinder
   4. resolution in radius of hollow cylinder
   5. origin

ellipsoid
   1. origin
   2. major axis (a vector)
   3. minor axis (a vector)
   4. orientation

toroid
   1. radius
   2. radius of cross-section
   3. orientation of toroid. i.e. the tilt of toroid from the y-axis
   4. resolution in radius
   5. resolution in radius
   6. origin
CHAPTER 13. MANIPULATING GEOMETRIC SHAPES

truncated generalized cone
1. orientation and shape of base
2. orientation and shape of top
3. height vector
4. origin

gear
1. inner radius
2. outer radius
3. height
4. origin
5. number of teeth

13.3.1 Creating and Manipulating CAD Objects

CAD objects are created with the command `create cad object` on the `cad` menu. This command will prompt you to select a type of object:

```
sphere

cylinder

hollow cylinder

toroid

truncated general cone

gear

ellipsoid
```

The command will then prompt you to enter the appropriate parameters of the object, depending upon the type.

You can change the parameters of a CAD object with the command `change cad object`. This command will prompt you to pick an object and it will then allow you to re-enter its parameters. The default values provided are the current values of the object's parameters.

The command `save cad object` generates a psurf file of a CAD object.
Chapter 14

Writing Objects to Files

*Jack* is a system for manipulating and analyzing objects which are described by the peabody language. The commands on the *write* menu generate these types of files.

<table>
<thead>
<tr>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>write environment</td>
</tr>
<tr>
<td>write figure definition</td>
</tr>
<tr>
<td>write psurf</td>
</tr>
<tr>
<td>write psurf globally</td>
</tr>
<tr>
<td>write figure psurfs</td>
</tr>
<tr>
<td>write environment to psurf</td>
</tr>
<tr>
<td>write positions</td>
</tr>
<tr>
<td>write figure position</td>
</tr>
<tr>
<td>write global positions</td>
</tr>
<tr>
<td>write frames global positions</td>
</tr>
<tr>
<td>save window setup</td>
</tr>
<tr>
<td>write JCL log</td>
</tr>
</tbody>
</table>

These files are the mechanism through which you can store your work in *Jack*.

These commands prompt you to enter names files to which information will be written. Before *Jack* opens the file, it first checks to see if a file by that name already exists. If such as file does exist, *Jack* will prompt you as to whether you want to overwrite the file. The prompt will be in the form of the name of the file in question. If you do in fact want to overwrite the file, just enter that string by hitting ENTER. If you want to write to a different file, change the name to the desired file name. If this file exists, *Jack* will repeat the process until you have entered an file name which doesn't exist or you have chosen to overwrite a previous file. If you choose to overwrite, *Jack* then renames the existing file by appending a tilde "~" to the name.

### 14.1 Writing Environment Files

The *write environment* command writes the entire environment to a file, which should have the suffix `.env`. This command will prompt you for the name of a file. It then writes the environment to the file according to the following rules.

- The first part of the file lists all surface attributes used in the environment.
- Following the surface attributes are the figure definitions.
- If the figure was read from a figure file, or if its definition has been written to a figure file, it is written as a reference to that file. Otherwise, the complete definition is written.
• Figure definitions consist of definitions of the segments followed by definitions of the joints. The final element of the definition is the current figure root.
• Following each figure definition is the current set of joint angles for the figure and the references to the surface attributes for each segment. The surface attributes themselves are defined at the beginning of the file.
• Following each figure definition is the current figure root and the location field. This figure root may differ from the one defined in the figure file if the figure has been re-rooted since the figure file was generated.
• Following all figure definitions are the constraints. The write environment command will prompt you as to whether you want to write the constraints.

14.1.1 Writing Positions

As described in Section 9.5, peabody environment files may contain a complete definition of the geometric environment, or they may contain just settings for certain parameters. In particular, it is sometimes useful to store information about the posture of figures, in terms of joint angles and figure locations. This is a way of saving a state in Jack without bothering with the complete figure definitions. These files are still peabody environment files with the .env suffix, but they don’t include the complete definition of the environment.

The write positions command writes out the joint angles and figure locations for all figures in the environment. Reading the file back in over the same environment will cause the figures to assume the locations and postures they had when the file was written. The write figure posture writes the joint angles for a particular figure.

♠ Tutorial 14.1: Writing Position Environment Files

This example explores saving positions for figures. Read in a human figure model.

1. Position the human figure in an interesting posture
   Use any of the manipulation commands.

2. Execute the command write positions
   The command will prompt you to enter a file name. Enter the name pos1.env. The file must have a .env suffix.
   Jack will then ask you whether you want to save the camera position. Answer either yes or no, with the right-mouse pop-up menu. If you save the camera position, then the current view will be restored when you read the file back in. Make your decision based on whether the current view is important to you.

3. Change the figure’s position and location
   Again, use any of the manipulation commands.

4. Read the file pos1.env
   The figure should assume its previous position. If you saved the camera position as well, then the view should be restored to its previous setting. If you did not save the view, then the view will remain as is.

5. Exit Jack and print the file pos1.env

The write figure definition command prompts you to pick a figure and then writes the definition of a figure to a peabody figure file, which should have the suffix .fig. The command also prompts you for a the name of the file.

These commands write information about the peabody environment, but they do not automatically write information about the psurfs which describe the geometry of the segments. If you have interactively modified the geometry of any of the segments, Jack will prompt you as to whether you want to write these psurfs. Since the peabody files contain only the names of psurf files to define the segment geometry, if you modify the psurfs, this information must be written back to the psurf files.
There are two commands for writing psurfs to files. The `write psurf` writes the psurf associated with a segment to a file, as it is. The `write global psurf` writes the psurf in world coordinates. This provides a convenient way of transforming the coordinates of a psurf.

Each of these commands prompts you to enter a file name. The file should have a suffix of either `.pss` or `.bps`. If the file has a suffix of `.bps`, `Jack` will write a binary psurf.
Chapter 15

Windows and Display Parameters

*Jack* runs under the default IRIX window manager *4Dwm*. The window manager provides control over the shape, location, and ordering of the windows on the screen. The IRIS graphics library (GL) controls drawing the objects in the graphics window. *Jack* can either draw objects in wireframe or it can shade them. The shading takes into account the affect of light sources.

*Jack* always has at least two windows: the message window across the bottom, and the primary graphics window which displays the peabody environment. Normally, this window covers most of the screen. You may create additional graphic windows as well. Each window is a view into the same geometric environment. The window manager allows you to move and reshape these windows as you like, possibly to expose text windows underneath.

15.1 Window Display Parameters

The primary type of *Jack* window displays the peabody environment. These are sometimes called “peabody” windows for this reason, to differentiate them from other types of windows such as the meter windows used for adjusting colors. The way in which the objects are displayed in the window depends upon the properties of both the window and the objects. Objects may be displayed in wireframe, or they may be shaded using the IRIS lighting model. The method of display may be set individually for each object in the environment.

The window menu has commands for manipulating the *Jack* windows.
Peabody windows come in three types: ordinary, wireframe, and shaded. Ordinary windows display objects based on the objects' individual parameters. Some objects may be shaded, others may be wireframe. Wireframe window display everything in wireframe, regardless of the individual objects' parameters. Likewise, shaded windows shade everything in the environment using the IRIS lighting model.

The type of a Jack window may be changed with the commands make window ordinary, make window wireframe, and make window shaded. The default Jack window is ordinary.

**Tutorial 15.1: Making Windows Shaded**

For this example, make sure that you have read in some objects.

1. Execute the command make window shaded

   The blue ground plane should change to a filled-in checkerboard, and the objects should appear shaded.

### 15.2 Object Display Parameters

*Jack* gives you control over many aspects of how it draws objects in the windows, using the commands in the display menu.
The display mode of each segment may be controlled with the commands `make segment shaded` and `make segment wireframe`. These parameters affect the display of the objects in ordinary *Jack* windows. Recall that in *wireframe* windows, all objects are drawn in wireframe regardless of the objects' individual parameters.
and likewise for shaded windows. The commands make figure wireframe and make figure shaded perform the same operation on all segments in a figure. The commands make everything shaded and make everything wireframe operate on all segments in the environment.

 Tutorial 15.2: Setting Object Display Parameters

This example involves changing the display parameters of objects and windows. Read in a human figure model and at least one other object, such as a box.

1. Execute the command make everything shaded
   The human figure and all other objects in the window should appear shaded. The ground plane will remain wireframe.

2. Execute the command make everything wireframe
   This changes everything back to its default state: wireframe.

3. Execute the command make segment shaded

4. Pick the head of the human figure
   The head should appear shaded but everything else will remain wireframe.

5. Execute the command make figure shaded

6. Pick the human figure
   Now the entire figure will appear shaded. This operation just sets the shading parameter for each segment in the figure.

The mode of display of objects depends upon both the parameters of the object and the window they are drawn in. As described above, ordinary windows display objects based on their own display parameters. Shaded windows display objects shaded. Wireframe windows display all objects wireframe. By default, windows are ordinary.

 Tutorial 15.3: Setting Window Display Parameters

For this example, read in a human figure model and a box. Start out with the window in ordinary mode.

1. Make the human figure shaded
   Do this with the command make figure shaded. Leave the box in wireframe.
15.2. OBJECT DISPLAY PARAMETERS

2. Execute the command `make window shaded`

   Everything should become shaded, including the ground plane. The box has a wireframe
   display parameter, but since the window is shaded, it appears shaded.

3. Execute the command `make window wireframe`

   Everything should become wireframe, including the ground plane. The human figure has a
   shaded display parameter, but since the window is wireframe, it appears in wireframe.

15.2.1 Transparent Objects

Objects can be made transparent with the command `make segment transparent`. The IRIS GT hardware is
able of drawing transparent objects well, but the non-GT graphics hardware, such as with the 50/G
and Personal IRIS G series machines, cannot. On these machines, the transparency is simulated with a
"screendoor" approach which is very primitive.

 Tutorial 15.4: Making Objects Transparent

This works well only on IRIS-GT machines. Make sure you have a human figure model and a box.

1. Execute the command `make segment wireframe`

   Pick the box. The box will appear shaded. Move it up in front of the human figure and you
   should be able to see through it.

15.2.2 Displaying Parts of Psurfs

The geometry of each segment is represented as a psurf. Psurfs are graphs of nodes and edges, either of
which may be displayed. Commands for changing the display properties of psurfs are on the psurf display
menu, which is sub-menu of the display menu.
CHAPTER 15. WINDOWS AND DISPLAY PARAMETERS

| psurf display
| turn nodes on
| turn nodes off
| turn edges on
| turn edges off
| turn face enumeration on
| turn face enumeration off
| turn node enumeration on
| turn node enumeration off
| turn face normals on
| turn face normals off
| toggle psurf bounding boxes
| toggle psurf nodes only

By default, only the edges of psurfs are drawn in wireframe, and only the faces are drawn when shaded. The nodes should be implicit from where the edges come together. If you are operating on a psurf and want to see the nodes explicitly, you may turn the display of the nodes on with the turn nodes on command. When the nodes are on, they are drawn as red cross-hairs.

The commands face enumeration on and node enumeration on cause Jack to display the numeric index of each face and node, respectively, as a part of the image. This is useful as a way of drawing diagrams of psurfs when modifying them by hand.

15.2.3 Displaying Sites

In addition to the appearance of the psurf, you may also enable or disable other parameters relating to the segment as a part of the geometric environment. Many operations require picking individual sites on certain segments. By default, the sites are not displayed, so this can be difficult if you do not know where they are. To see where they are, you may turn them on, with the turn sites on command. Sites are displayed as little red axes labeled with $x$, $y$, and $z$. It is usually rather annoying to have lots of sites displayed, so only turn them on when you need to see exactly where they are. This is not always necessary. For example, there is a site in the tip of the hand on the human body. You don’t need to see it to know where it is.

The world segment has sites associated with it, to which everything else is rooted. It is sometimes beneficial to turn these sites on. This segment has no associated psurf, but you can pick it interactively by clicking the mouse at the coordinate origin.

15.2.4 Turning Objects Off

You can completely disable the display of segments with the command turn segment off. This will completely disable its display so it will be invisible. It will still exist: turning it off is quite different from deleting it!

The command turn segment on turns the display back on. Note, however, that when a segment is turned off, you cannot see it to pick it. You must enter its name from the keyboard.

There are corresponding commands for operating on entire figures. These commands simply apply the same operation to each segment within the selected figure.

♠ Tutorial 15.5: Turning Segments Off

Use a human figure model for this example.

1. Execute the command turn figure off
   Select the human figure. It will disappear, although it still exists.

2. Execute the command turn segment on

3. Pick the head
   To do this, you must enter the value from the keyboard. Hit "K," and when Jack prompts you for the segment name, enter human.bottom.head.
15.3 Shading Windows

Shaded windows can take a long time to draw, so sometimes it is not convenient to maintain a window in shaded mode, even though it is helpful to see the object shaded. For this reason, the command shade windows which draws the windows shaded just once and then pauses. This is useful to execute every so often to get a better visualization of the arrangement of the objects.

Â TUTORIAL 15.6: SHADING WINDOWS

For this example, make sure you have some objects in the scene. Make sure your window is an ordinary one, also.

1. Execute the command shade windows

   The window should appear shaded, and Jack will prompt you to hit any key to continue:

   Hit any key to continue...

15.4 Generating an Image file of a Window

Sometimes it is convenient to save what is on the screen in an image file. This can be done with the command write window image, which generates an image file. The command will prompt you to enter the name of the output filename. The filename must have one of the supported suffixes as listed in Section 12.8.2.

Â TUTORIAL 15.7: WRITING AN IMAGE

Be sure you delete the image file when you’re done: it takes a lot of space.

1. Execute the command write window image.

   Jack will prompt you to enter the name of an output image file.

2. Enter the file name jack.rle

   The file name may have any suffix as defined in Figure 12.6, but this time we’ll just use the suffix .rle. When you enter the name, Jack will begin writing the image to the file. It may take some time.

3. Exit Jack and Display the File

   Use the display program:

   % display jack.rle

   It looks just like a Jack window, doesn’t it?

4. Delete the file jack.rle when you’re done

   It is very large.

15.5 Creating New Windows

The create ordinary window command creates a new ordinary Jack window. The create wireframe window command creates a new Jack window and makes it wireframe. The create shaded window command creates a new Jack window and makes it shaded. The new window provides a different view of the same peabody environment.
The delete window command deletes the current *Jack* window. If you close a window using the *4Sight* window manager pop-up menu, you kill *Jack*.

**Tutorial 15.8: Creating New Windows**

1. **Execute the command create ordinary window**
   
   The window manager will prompt you to open the window. You will probably want to resize the existing window so that the two windows do not overlap too much. Use the window manager pop-menu to do this, the one which you get by pressing the mouse in the window's title bar.

2. **Execute the move figure command**
   
   During executing this command, you may transfer the mouse from one window to the other. The two windows are just different views of the same thing.

3. **Execute the change view command**
   
   You begin changing the view in the "current" window, i.e. the one in which the mouse is when you executed the command.
   
   You can't transfer the operation to other windows like with moving figures. Pressing the mouse in another window has no affect.

### 15.6 2D Windows

Normally, *Jack* draws the objects in the windows with a 3D perspective view. It is also possible to have 2D windows which display orthographic projections. Commands for creating 2D windows are on the *ortho* menu, which is a sub-menu of the window menu. The *four panel screen* command arranges the screen with four windows: an orthographic projection window for each axis, and fourth perspective view.

You can move objects in 3D windows just like in 3D windows, except that you can't translate or rotate along axes parallel to the window view. When you move figures in 2D windows, you must also select the axes to correspond to the type of window.

**Tutorial 15.9: The Four Panel Screen**

Read in a human figure model for this example.
1. **Execute the command** four panel screen
   The screen will rearrange itself so that there is an $x$ view window, a $y$ view window, a $z$ view window, and a normal 3D perspective view window.

2. **Move the human figure**
   Execute the move figure command and pick the human figure. You can point at it in any of the windows.

3. **Move the mouse to the perspective window**
   Moving the figure here is just like with a regular window.

4. **Move the mouse to the $x$ view window**
   You are looking down the $x$ axis. Press the middle button to move the object up and down in $y$. Press the right button to move the object side to side in $z$. Pressing the left button has no effect because you can't translate along $x$ in this window.

5. **Move the mouse to the $y$ view window**
   You are looking down the $y$ axis. Press the left button to move the object up and down in $x$. Press the right button to move the object side to side in $z$. Pressing the left button has no effect because you can't translate along $y$ in this window.

6. **Move the mouse to the $z$ view window**
   You are looking down the $z$ axis. Press the left button to move the object side to side in $x$. Press the middle button to move the object up and down in $y$. Pressing the left button has no effect because you can't translate along $z$ in this window.

### 15.7 Windows and Cameras

*Jack* represents the view in each window by the global position and orientation of a reference site on a figure. *Jack* automatically creates a "camera" figure when it creates a window. You normally don't see the camera because it is turned off. You can turn it on with the command turn camera on. Displaying the camera gives a visual illustration of how the view changing mechanism works.

*♦ Tutorial 15.10: Displaying the Camera*

1. Create two windows
   Reshape them so they are side by side.

2. **Execute the command** turn camera on
   This enables the display of the camera in the current window, i.e. the one where the mouse is when you execute the command. You may not immediately be able to see it. You definitely won't be able to see it in that window, because you are looking out the front of it.

3. **Change the view in the other window**
   Move the mouse to the other window and change the view. Zoom way out, and you should be able to see the other window's camera.

4. **Change the view in the other window**
   As you change the view, you can see the camera move in the other window. This illustrates how the mouse controls the global position of the camera.

### 15.8 Changing Window Parameters

#### 15.8.1 Frozen Windows

When there are several complex objects in a scene, the amount of time which *Jack* consumes drawing the windows can become significant. This becomes especially important when manipulating an object when...
there are more than one window. To avoid this, you may freeze windows. A frozen window will not be dynamically redrawn when you are manipulating a figure in another window. Its image will remain the same until then end of the command. At the end of each Jack command, all windows are redrawn, frozen or not. Normally, windows are not frozen, so that any changes in the environment are reflected immediately in all windows. This may be reset with the command thaw window.

15.8.2 Setting Window Location

The command set window location allows you to explicitly enter the left, right, bottom, and top screen coordinates for the location of a window. This is convenient when you need to place a window in a precise location on the screen. The parameters are in screen coordinates, ranging horizontally from 0 to 1279, and vertically from 0 to 1023.

15.9 Changing the Appearance of the Background

By default, Jack windows have a ground plane as a background, displayed as a wireframe grid with two gradations of lines, the major axis and the minor axis, along with the coordinate axis. This plane serves as an aid in judging the placement of objects in the environment. Without it, objects just float in space with no sense of orientation.

You have control over the appearance of this background plane using the commands in the background menu.
You may control the display of the background with the commands `turn background off` and `turn background on`. This turns off both the grid and the stars. The display of the grid and stars may be set individually with the commands `turn stars off`, `turn stars on`, `turn grid off`, and `turn grid on`. The display mode of the background grid may be controlled with the commands `make background shaded` and `make background wireframe`. You may also control the display of the coordinate axis projections with the commands to enable and disable the x, y, and z projections. All projections may be controlled simultaneously with the commands `turn projections off` and `turn projections on`.

The spacing of the major grid and minor grid are in terms of the screen scale. By default, the major grid size is the same as the screen scale, which is 100, and the minor grid size is 25. You may change these with the major grid size and minor grid size commands.
Chapter 16

Customizing Jack

This chapter describes how to change some of Jack's basic parameters and how to set it up so that Jack always uses these parameter settings whenever you run it.

Jack allows you to control many of the parameters which affect its behavior and appearance. These allow you to change the basic colors Jack uses to draw the windows, to specify different sets of units for printing data, to change the general scale of the objects it models, and to extend the way you can execute commands. After experimenting with these commands, you may arrive at a configuration which you especially like, but it is tedious to enter these commands every time you run Jack. Jack lets you group these commands into an initialization file which it reads whenever it starts up. This file, ~/.jack.jcl, may contain any Jack commands in the JCL syntax. The most common, however, are the ones described here.

16.1 The ~/.jack.jcl Initialization File

When you run Jack, it looks for a file in your home directory called ~/.jack.jcl. If one exists, it reads JCL commands from it. These commands can technically do anything that can be done in JCL, but it is best to keep these commands limited to ones which set parameters and bind commands to keystrokes.

When you execute Jack Version 4.7 or greater for the first time, it will automatically create an initialization file for you which contains a standard set of key bindings. You can add whatever other information you like after that. You can also redefine the standard bindings.

16.1.1 Figuring Out the JCL Syntax

The JCL syntax for Jack commands can be awkward and is not well documented. The best way to go about collecting commands to place in your ~/.jack.jcl initialization file is to execute the relevant commands interactively and then write a JCL log file with the write JCL log command, on the write menu. You can then edit this file to extract the proper commands and place them in your ~/.jack.jcl file.

16.2 Binding Commands to Keystrokes

Jack allows you to bind commands to keystrokes to avoid having to go through the menus to execute commands. Jack comes with a default set of bindings:
These bindings are included in the default initialization file which Jack creates for you. You can change them if you like.

You can bind commands to keystrokes with the command bind command to key. This command prompts to enter a command, which you pick from the pop-up menus as if you were going to execute it. It then prompts you to enter a key. You can enter most any key on the keyboard. The function keys are especially nice. It is generally better to bind commands to control characters rather than actual characters.

♣ TUTORIAL 16.1: BINDING COMMANDS TO KEYS

1. **Execute the command** bind command to key
   Jack will prompt you to enter a command from the pop-up menus, just as if you were going to execute it.

2. **Pick the command** read file
   Jack will then prompt you to enter a key.

3. **Hit the F5 key, the #5 function key**
   Now, whenever you hit F5, Jack will execute the read file command.

4. **Hit the F5 key**
   This you should do when Jack is waiting for you to enter a command. Jack will execute the read file command.

5. **Execute the command** write JCL log
   This command is on the write menu.

6. **Enter the file name** log.jcl

7. **Exit Jack and print out the file log.jcl**
   There should be an entry in it like:
   
   ```
   bind.command.to.key("read.file","F5");
   ```
   
   This is the JCL syntax for the key binding command. If you place this in your "/.jack.jcl" file, then every time you run Jack, you can execute the read file command by hitting F5.

The JCL syntax for the bind command to key command gives the key in double quotes. Control characters are given with a preceding circumflex, such as "^W". The function keys along the top of the keyboard are signified by their number preceded by the letter F.

16.3 Setting Colors

By default, Jack uses mostly shades of blue to draw the screen. If you don't like blue, you can change the colors. You can tailor the colors to your own liking and then save them in a file which can be read in automatically when you run Jack, so that things look the way you like them to.
You may adjust colors using the commands in the color menu, which is a sub-menu of the display menu. Select one of these menu items to change the color. These commands will first prompt you to open a meter window. You should make this window relatively small and place it out of the way of the primary Jack window.

To change the color, press down a mouse button while the mouse cursor is over one of the meters. The value in the meter will then snap to the level indicated by the mouse position inside the meter. While the mouse button is down, vertical movement of the mouse changes the value in the meter. Release the button when the desired value is achieved. You may then repeat the process on any of the three meters. When you achieve a satisfactory color, hit the ESCAPE key. If you want to abort the process and revert to the previous color, hit -C.

The axis lines in the background grid are drawn using the axis color for the coordinate axes, major grid color for the grid lines spaced every unit distance apart, and minor grid color for the smaller in-between lines. The background of the window is drawn using background, which is usually black. By default, the axis lines are white, the major grid lines are light blue and the minor grid lines are dark blue. The remaining colors are self explanatory.

The major and minor highlight colors are the colors Jack uses to highlight objects when you pick them interactively.

16.4 Setting Jack Parameters

The commands on the parameter menu provide a means of controlling some of Jack global parameters.
16.4.1 The Scale of Jack Windows

*Jack* makes the initial assumption that all objects are sized proportionally to the human body, in centimeters, and it sets up the view and ground plane so that a human body, which is approximately 200cm tall appears comfortably on the screen. However, the scale of the display may be changed to display objects of any size. *Jack* maintains a scene *scale*, which is a ball-park number of centimeters which will appear comfortably on the screen. By default, the scene scale is 100, which means that the human figure will fill most of the screen. This value is the same for every window.

You can change the scene scale interactively using the `set scene scale` command, which prompts you for the scale in centimeters. The default value supplied by the command is the current window scale. This command re-scales the view in each window, adjusting the ground plane as well.

You can instruct *Jack* to set the scene scale automatically with the `-s` command line option. This option takes a floating point scale value, which follows the `-s` on the command line, with or without a space in between.

*Jack* can display various features of the geometric objects in the environments. One of the most beneficial aspects to display is the site coordinate systems, which appear as little labeled coordinate systems, labeled as $x$, $y$, and $z$. The size of these auxiliary icons is chosen quite arbitrarily so that they look appropriate on the screen, but that can be changed. The auxiliary items of a window, the sites, nodes, and rotation wheel, are drawn according to the *auxiliary scale*. This scaling factor is applied to the scene scale. By default, the auxiliary scale is $\frac{1}{4}$, so that the site frames appear one quarter the size of a "unit" object. This scale can be changed with the `set scene aux scale` command.

16.4.2 Setting Units

When *Jack* prints distance and angle values associated with peabody objects, it always prints the units along with the numerical value so there is no confusion about which units are being used. By default, distances are printed in centimeters, angles in degrees, and masses in grams. The "units" commands on the parameter menu allow you to change the default units. Each of these commands prompts the user for a new unit. Legal units for distances are: mm, cm, m, in, ft, yd, mi. Legal units for angles are rad for radians and deg for degree. Legal types for masses are: g, kg, lb.

The "precision" commands may also be used to set the precision with which the various values are formatted. The precision is an integer number of decimal places with which to format the value. By default,
the values are printed with two decimal places.

The rotation part of a homogeneous transformation may be printed either in terms of the \textit{xyz} operator or the \textit{quat} operator. The \texttt{set rotation type} command provides control over this. The default type is \textit{xyz}.

### 16.4.3 Shading Parameters

The IRIS illumination model has several parameters which control the effect of the shading but are not properties of the attributes or light sources. These parameters give you control over how the lighting calculations are performed, and how much time it takes to perform them.

The scene has an ambient light parameter which specifies how much light is present independent of any light sources. Ambient light is reflected equally in all directions, independent of the orientation of the surface. The command \texttt{set scene ambient} allows you to adjust this parameter, which is an RGB color value.

The light emitted from the light sources may also be attenuated so that objects farther away from the lights appear darker than objects which are close. The attenuation is defined by two parameters, $K_{sdf}$ and $K_{sdav}$, the fixed scene distance attenuation factor, and the variable scene distance attenuation factor, respectively. They modulate the intensity of a light source according to the following formula:

$$
M_{distance} = \frac{1.0}{K_{sdf} + K_{sdav} D_p}
$$

where $D_p$ is the distance from the light to the point being shaded, and $M_{distance}$ is the factor by which the illumination is attenuated. If $K_{sdav}$ is zero, then the attenuation calculations are not performed.

The lighting model also allows the viewer to be local or at an infinite distance. When the viewer is local, the view vector is different for each point in the scene. When the view is located at infinity, the same view vector may be applied to all points in the scene, making the lighting calculation much more efficient, although the specular highlights will not be computed as correctly. You may set this parameter with the \texttt{make viewer local} and \texttt{make viewer infinite}.

For more details about the parameters of surface attributes and light sources, refer to Section 12. For more information about the illumination model, refer to the \textit{IRIS Graphics Library Users' Guide}. 

Chapter 17

Utilities

17.1 Collision Detection

There is a formal collision queue in Jack. It is a list of segment pairs (called a CollisionPair object). Each CollisionPair (CP) object contains pointers to two segments, which will be checked for collision, plus various pieces of information used by the different collision detection (CD) routines. Only the segment pairs on the collision queue (CQ) will be checked for collision. This is different from the earlier collision detection system, which checked all segments in the environment against all segments for collision. There are several new JCL commands for managing the CQ. The commands are located on the utility ⇒ collision menu:

<table>
<thead>
<tr>
<th>Collision</th>
<th>CQ check environment</th>
<th>CQ check figure</th>
<th>CQ check figure pair</th>
<th>CQ check segment</th>
<th>CQ check segment pair</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CQ clear queue</td>
<td>CQ clear figure</td>
<td>CQ clear figure pair</td>
<td>CQ clear segment</td>
<td>CQ clear segment pair</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CQ clear intra figure</td>
<td>CQ clear current figure</td>
<td>CQ clear figure pair</td>
<td>CQ clear segment</td>
</tr>
<tr>
<td></td>
<td>set collision mode</td>
<td>set collision display</td>
<td>set collision detect type</td>
<td>set collision threshold</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CQ update collisions</td>
<td>CQ report collisions</td>
<td>CQ flash queued segments</td>
<td>CQ flash colliding segments</td>
<td></td>
</tr>
</tbody>
</table>

17.1.1 Command Interface to Collision Detection

This section provides a brief description of the interactive and JCL commands which you can use to control the CQ and the collision detection routines. In the following, CQ means Jack's global collision queue, and CP means a CollisionPair object.

For purpose of illustration below, let's assume that you have a simple environment that has $\sum$ total segments, and each figure has $\Gamma$ segments.
17.1.1 Commands for Adding Collision Checks

These commands will add segment pairs to the collision queue. Using collision detection is an expensive operation, in that the more segment pairs on the queue, the longer it takes to compute the collisions. Using a mix of the following commands will let you set up the CQ in the most efficient manner (i.e., with the fewest pairs possible).

CQ check environment  This command will add a check for every segment pair in the environment to the CQ. This will add \((\Sigma \times \Sigma) - \Sigma\) segment pairs to the CQ. For example, if your environment has 100 segments, this would generate almost 9,900 collision pairs. Only use if you have small environments!

CQ check figure  This command prompts for a figure and will add a check between every segment in the figure and the environment. It won't add any intra-figure checks. This will add \((\Sigma \times \Gamma)\) segment pairs to the CQ.

CQ check figure pair  This command prompts for two figures, \(\text{figure}_1\) and \(\text{figure}_2\). It will add a check for every segment pair between \(\text{figure}_1\) and \(\text{figure}_2\), but no intra-figure checks (i.e., only inter-figure checks). This will add \((\Gamma \times \Gamma)\) segment pairs to the CQ.

CQ check segment  Prompts for a segment and then adds a check between this segment and every other segment in the environment. This will add \((\Sigma \times 1)\) segment pairs to the CQ.

CQ check segment pair  Prompts for two segments, \(\text{segment}_1\) and \(\text{segment}_2\), then adds a check between the two segments. This is the lowest level command for creating collision pairs; This will add only 1 segment pair to the CQ.

17.1.1.2 Commands for Removing Collision Checks

These commands will remove collision checks between sets of segments. These commands should be used to reduce the size of the collision queue, so that collision computations can be quicker (and focused on only those potential collisions of importance).

CQ clear queue  Clears the collision queue by removing every check (CP).

CQ clear figure  Prompts for figure and removes every check involving any segment of figure.

CQ clear intra figure  Prompts for figure and removes all checks which involve a pair of segments from figure (i.e., all intra-figure collision checks will be removed).

CQ clear current figure  Prompts for figure and removes all checks which involve segment pairs from figure and are currently in collision (i.e., all intra-figure collision checks which are currently in collision will be removed). You are signaling the CQ that the current collisions are OK.

Collision detection with human figures, or for any articulated figures, involves allowing some collisions to occur and be ignored, such as the upper and lower arm segments which overlap at the elbow. This will take the figure in its current posture and catalog all collisions as acceptable. This may be done several times. This command replaces the old command initialize figure for collision.

CQ clear figure pair  Prompts for two figures, \(\text{figure}_1\) and \(\text{figure}_2\), then removes all checks which involve segments of the two figures.

CQ clear segment  Prompts for segment, then removes all checks involving segment.

CQ clear segment pair  Prompts for \(\text{segment}_1\) and \(\text{segment}_2\), then removes the check involving these two segments, if it exists.
17.1. COLLISION DETECTION

17.1.1.3 Commands for Controlling Collision Checks

These commands affect the way Jack checks for collisions, and how they are reported to you.

**set collision mode** Prompts for *type* and sets Jack's collision detection mode, where *type* is one of:

- **on demand**: only updates CQ via CQ check collisions.
- **continuous**: updates CQ every interactive step.
- **prevention**: updates every interactive step, plus backs up one frame when a collision occurs.

Collision prevention is very different from collision avoidance, and the current system uses no intelligence about how to accomplish a movement without collisions.

It merely waits for them to happen and then backs the figure up to its previous non-colliding posture. This also means that the figures are left in a posture in which the figures do not generally touch, but are some distance apart, depending upon how far the figure moved in the last time step.  

**set collision display** Prompts for *type*; It tells whether collisions are highlighted or not. If *type* is off will not highlight, and `highlight` will. Textured segments may disappear during highlighting.

**set collision detect type** Prompts for method; It determines which algorithm Jack uses to detect a segment-segment collision. Unfortunately, all the methods assume a convex polyhedra, so if your psurfs are not convex, the algorithms work on the convex hull. The choices for *method* are:

- **bounding box only**: only use the global axis aligned bounding box of the segment's psurf.
- **bounding sphere only**: only use the bounding sphere of the segment's psurf.
- **Gilbert & Johnson**: first checks bounding box, then uses the named algorithm. This algorithm also can compute the distance between the segments, negative means colliding. This tends to be the faster algorithm.  
- **Moore & Wilhelms**: This tends to be slower (much) because the implementation isn't the most efficient, but it should be accurate.  
- **The Unknown Method**: This is a routine in Jack which uses something that looks a little like Moore & Wilhelms method, but a little more optimized. There were no comments in the code...

**set collision threshold** Prompts for *number*, then sets the distance at which a collision is reported. Only works when using Gilbert & Johnson. Defaults to 0.

17.1.1.4 Commands for Reporting Collisions

These commands are for reporting the status of any collisions on the collision queue.

**CQ update collisions** Forces an update of the CQ. Use when CD mode is on demand. You may want to bind this command to a key, say "U", then you can quickly update the collisions by pressing "U".

**CQ report collisions** This command prints a verbose listing to the terminal of current CQ status. It lists all collision pairs currently being checked, and if they are in collision.

**CQ flash queued segments** Flash all the segments that are involved in at least one CP. This is a quick visual way to confirm which segments are being checked for collisions.

**CQ flash colliding segments** Flash all segments that are in collision. A quick way to confirm which segments are currently in collision.

---

1. This option is currently under construction, and is unavailable at this time.
17.2 Tracing Sites and Segments

Jack lets you trace sites and segments. Traces are graphical trails that objects may leave behind as they move. The commands for creating and controlling traces are on the trace menu.

The trace site command causes a site to generate a trace. The trace will show where the site moves in space. The trace is drawn as a series of line segments connecting each of the site's previous global positions. Whenever the site moves, due to either the movement of a joint or a figure, a new section will be added to the trace.

The trace segment command cause a segment to generate a trace. The trace will show where the segment moves in space. The trace is drawn as a series of images of the segment, one at each of the segment's previous global positions. Whenever the segment moves, due to either the movement of a joint or a figure, a new section will be added to the trace.

You can clear a trace with the clear trace command. This erases the currently drawn path, but the trace remains active, so when the site moves again, the trace will continue. The command untrace site stops a trace but does not erase it. The command delete trace deletes it altogether, clearing its path as well. These commands each prompt you to pick a trace by pointing at it with the mouse.

The color of a trace may be set with the command set trace color. It is not possible to have a single trace with two different colors.

17.3 Ghost Figures

Ghost figures are images of figures in specific postures. Sometimes it is necessary to position a figure in a certain posture very close to a previous posture. A ghost can help out. The create ghost figure allows you to pick a figure and it then creates a image of the figure in its current posture. The image will look exactly like the figure, except that it will be all one color. However, the ghost figure is not movable or manipulatable. It serves only to mark a posture temporarily for future reference.

The command delete ghost figure deletes a ghost.

17.4 Recording Images to the Video Disk

Single frames can be recorded to the Abekas Digital Disk Recorder with the command write image to vdisk. This command allows you to send either a single window or the entire screen. This command is designed to be used in an automated fashion, usually in a JCL file.

The will prompt you to select the option:
The window option causes the current window to be recorded. The window is automatically reshaped to be 486 rows and 720 columns, the size of a frame on the video disk. The window sideways options causes the current window to be recorded just like the window option, but the window is turned sideways first. The reason for this is that this allows the window to be tucked out of the way on the IRIS screen so you can continue to work on the machine while the image records. Don't expect Jack to function very well with the window turned sideways: changing the view and moving figures interactively don't work.

The whole screen causes the whole screen to be recorded, although the image is cropped off just above the status window. This option allows you to record a multi-window scene in Jack. You should arrange the windows before issuing the command.

The command write environment files to vdisk reads a sequence of peabody environment files and displays them in turn, recording the image to the video disk. The command prompts you to enter the base name of the file and a starting and ending file number. It will read files whose names are formed by concatenating the base file name with the number padded with zeros to four digits. For example, a base name of /usr/users/norman/anim/frame and a frame range of 0 to 749 would read the files:

```
/usr/users/norman/anim/frame0000.env
```

through

```
/usr/users/norman/anim/frame0750.env
```

The command prompts you to select a recording option just like write window to vdisk. It also prompts you to enter a starting frame number for the video disk.

### 17.5 Disabling Graphics

The commands disable graphics and enable graphics turns off and on Jack drawing of the graphics windows. This is sometime useful when you need to perform operations which you can do without graphics, particularly involving a very large and complex environment. This is especially useful in JCL files, in which a long sequence of commands can be executed much more quickly without re-drawing the windows between each command.

### 17.6 Remote Commands and the Command Port

Jack can allow an external process to control and query Jack through a socket. The socket is called a command port. The socket is bidirectional, receiving JCL commands and sending a reply message back. Messages sent to Jack can be any valid Peabody string, but normally would be one or more JCL commands.

The format of the return messages from Jack is an ASCII string:

```
size return_code : message ;
```

This format was adopted to match the GDCLI format used in the DEPTH project, and is very basic (using Xlisp is much more powerful — see Section F). The size field is a 5 character string representing the length of the rest of the message (in bytes). For most JCL commands, the return_code will be 0, representing success , or 1 representing some kind of failure (these are GDCLI artifacts). Also, several JCL commands exist that are intended to be used exclusively by external programs communicating with Jack via the command port. They generally have the remote+ prefix, and return useful values in the message field. These commands do not appear on any Jack menus. The following list is an overview of these Jack commands.

Also, there is a small program, jacket.c, which is a skeleton program which sends commands to Jack and prints the return messages from Jack. This program is distributed with Jack in the $JACK/src/jacket directory.

---

This method of controlling Jack will probably not be developed much beyond where it is today. Other protocols will most likely be used.
17.6.1 Network and Remote commands for Command Ports

open command port Prompt for a host, and attempts to establish a connection to the remote client. The host parameter can either be self or an IP number or IP host name (e.g. `bambi.cis.upenn.edu`). If it’s self, then Jack will block waiting for a socket connection. If it’s an IP number, then Jack will attempt to connect to that host, where a connection should be waiting. There is no limit to the number of command ports opened.

close command port Prompts for a number, then will close the command port associated with file descriptor number.

[show-hide] port messages These commands toggle the display of the command port stream into the Jack message window. Useful for debugging your external program and insuring the messages are formatted correctly.

remote.query.figure.names() Queries the peabody environment for a list of all figures. The return message format is:

```
0: n, figure-name1, figure-name2, ..., figure-name_n;
```

remote.query.site.names(“segment”) Returns a list of site names on the segment segment. The message format is:

```
0: n, site-name1, site-name2, ..., site-name_n;
```

remote.query.distance(“site1”, “site2”) Returns the distance between two sites in the environment. Both site1 and site2 should be fully qualified peabody site names. The return message format is:

```
0: distance, n_x, n_y, n_z;
```

Where distance is the cartesian distance, and n are the components of the unit vector from site1 to site2.

remote.query.position(“name”) Returns the global (world coordinates) position of a peabody object name. Name is a fully qualified peabody name for a figure, segment, or site. The return message is the transform of the location:

```
0: T_x, T_y, T_z, R_x, R_y, R_z;
```

where T_i is translation and R_i is rotation along the global axis i. If name is a figure name, then the root site location is returned. If it’s a segment name, then the segments base site is returned. Otherwise, the site’s location is returned.

remote.pick.figure(“prompt-string”, figure) Prompts user to pick a figure, makes it current, then returns the figure name. Of course, to force an interactive pick, you should use $ in place of figure. After the command is executed, figure will be replaced with the name of the figure picked. Prompt-string will appear in the status window during the pick.

remote.pick.site(“prompt-string”, site) This is the same as above command, except figure is replaced by site.

17.6.2 Remote Commands for Collision Detection

remote.CQ.query.collison.status(“query”) This command handles two queries of the CQ. For each query, it returns a message according to various conditions. For query:

- `global`: Return message is [170: ;] if no collisions on the CQ, or [20: ;] if there are.
- `full`: Return message is [170: ;] if no collisions on the CQ, or if there are collisions, returns:

```
[20: n, segment_{11}, segment_{12}, 0, 0, 0, ..., segment_{n1}, segment_{n2}, 0, 0, 0;]
```

where n is the number of collisions occurring on the CQ, followed by each colliding pair of segments, and the location of the collision, which we can’t produce so it’s just 0,0,0.
17.7. **PATHS**

remote_CQ_query_collision("name1", "name2") This function checks for collisions between the segment(s) in the first argument, and the segment(s) in the second argument. If name is a figure name, it will use the entire list of that figure's segments. Note that the segment pairs that are checked should be on the CQ. The return message is [170: ;] if there is no collision, or [20: ;] if there is a collision between any of the specified segment pairs.

### 17.7 Paths

A path is a set of \( n \) sites on a segment, named \{ \( pnt_0, pnt_1, \ldots, pnt_n \) \}, with one additional site called point. The line connecting the sites is defined by a spline curve (that's why we sometimes call them spline paths). Paths are mostly useful for animation (see the command `create path motion`).

When a path is animated, the point site moves along the path. This allows a convenient way to reference the motion in other motions or behavior commands. For example, a timed arm control could reference the point site to have the arm follow the site as it moves along its path.

There are several commands for manipulating paths. The command `create path` will create a new figure, with one segment called `paths`, and then start building a path on that segment. It will prompt you to move each new site it creates, while continuously updating the curve between the sites. After you place each site, press ESC to continue. It will then prompt you for a yes/no reply to add more sites to the path. Entering no will complete the path.

Once a path is built, you can attach it to other figures by executing the `create joint between segments`, which will detach the `paths` segment from its mother figure, and place it on another figure (or you can leave it as an independent figure if you wish).

The command `delete path` will delete a path. The commands `add path point` and `delete path point` may be used to add and remove, respectively, points from a path. The command `set path parameters` can be used to set the number of samples taken along the curve. This defaults to 31. You would want to increase this number if your path has more than, say, 10 points.

### 17.8 Prototype and Contributed Software - The contrib menu

There is a sub-menu off the main menu, contrib. Under this sub-menu are placed prototype commands (prototypes of new features we are working on) and also contributions that students/researchers have made to Jack.

They are documented in the following sections. If a contribution generates a lot of interest from the Jack user community, we will then fully test it and incorporate it into the regular Jack release.

### 17.9 Fractals

The fractal editor \(^5\) for Jack allows quick generation and editing of landscape-like 3d fractals. The generation process is fast and allows landscapes with up \( 2^{18} \) data points to be designed and manipulated in real-time, though higher exponents are possible if you're willing to wait through the swapping.

The generation process uses a random number generator which is deterministic given the same seed so it is possible to regenerate a fractal with different parameters but with the same basic structure since the random values chosen are identical.

---

\(^5\) Welton Becket
Aside from the usual parameters for fractals like fractal dimension and standard deviation, power laws are allowed for data points above a dynamically alterable sea-level and below the same sea-level. The current fractal literature speaks only of power laws for above sea level and allowing one for below offers the possibility of generating scenes where mountains and valleys have evolved from very different erosive forces (rain and wind for mountains and glaciers and water drainage for valleys.) Many approaches to fractal landscapes cut points below sea level to zero, and this creates too harsh a discontinuity between mountains and level regions. With the dual-power-law approach smoother and more realistic transitions can be achieved.

17.9.1 Type of Fractals Implemented

Three-dimensional fractals (2d complex) are the fractals considered, where the complex data is interpreted as a height field. The method chosen for fractal generation is midpoint subdivision because it is very fast and the only method feasible for near real-time use. Although midpoint subdivision is susceptible to what Mandelbrot calls “creasing” and not as “pure” as spectral synthesis or white noise integration, it is fast and as Mandelbrot notes, some of the ill effects can be relieved by using non-gaussian distribution functions.

17.9.2 Fractal Generation and Post-processing Parameters

There are two sets of parameters used to edit fractals: statistical information for subdivision, and post-processing information used to shape the resulting fractal.

The parameters used in fractal generation are:

1. **Maxlevel** or the number of recursions. This controls the resolution. The fractal will have \(2^{\text{Maxlevel}}\) data points across each of the two real axis, and \(2^{2\text{Maxlevel}}\) data points altogether.

2. **H** where the fractal dimension \(D\) is given by \(D = 3 - H\).

3. **Standard Deviation** — Initial standard deviation for midpoint displacement. On each iteration it is multiplied by \(\frac{H}{2}\).

The post-processing parameters are:

1. **X and Z dimensions** — these define the dimension of the fractal in its local xz-plane. They do not involve regeneration of the fractal.

2. **Power Laws** — the fractal editor allows application of power laws to data points above and below the defined sea level (two values are allowed — one for above and one for below sea level.) A value of zero is treated as a special case, and clips the respective data points to sea level.

3. **Sea Level** — This is used with the above mentioned power laws.

17.9.3 The Interface

The interface to the fractal generator is through Jack. The **contrib fractal menu** contains the operation create fractal and select current fractal and the other commands for editing a fractal (all commands of the form fractal *). The command create fractal prompts for initial parameters, generates and displays the fractal, and defines the current fractal as the one generated. Select current fractal allows reselecting the fractal by typing in the name. The fractal * commands allows changing the existing parameters.

A possible approach to achieving a desired fractal can be:

1. start with a fractal of low resolution, perhaps a maxlevel of 5 or 6. Low resolution fractals draw and regenerate faster, and because the subdivision algorithm is breadth-first, regenerating the fractal with a higher resolution will give a fractal structured the same but with finer resolution (incrementing maxlevel by one yields the same fractal with the midpoint of every line displaced.) When the desired fractal is achieved the resolution can be increased to 7-9.
2. flip through different fractals by selecting fractal re-seed and then pressing '!' to repeat until a reasonable fractal is found. Re-seed constructs a new fractal by re-seeding the random number generator with a value based on the current system time.

3. modify the H value and standard deviation until the desired amount of roughness is found. (via the commands fractal H and fractal standard deviation).

4. perhaps apply powers to points above and below sea level. Powers very close to 1.0 but less than 1.0 soften the respective points, and powers from 0.0 to 0.6 tend to flatten the respective points increasingly towards 0.0. Leaving points above sea alone, and applying a deadening power to points below sea can create an ocean-like scene or a desert-with-mountains scene. Leaving below-sea points alone and applying a deadening power to above-sea points can create plateaus or cliff-like scenes (via the commands fractal above sea power law and fractal below sea power law).

5. alter the sea level to shift the amount of mountain/plateau or sea/planar area (via the commands fractal sea level and fractal max level).

17.9.4 Reading and Writing Fractals
The peabody language has been appended to read and write fractal data. Since the parameters for the fractal along with the seed completely define the fractal, this is all that is written. The line:

fractal = (name,seed0,seed1,seed2,H,additions,xdim,ydim,
          maxlevel,std.dev,sea_level,above_power,below_power)

in a segment pirates the enclosing segment and turns it into the specified fractal with the old segment's attributes and global position. It's not a good idea to construct a fractal from the fractal vector in a peabody file, but it is possible, and convenient perhaps only for maxlevel.

The ray-tracer has been augmented to accept these fractal definitions although at the moment it treats them as a huge set of faces and does not take advantage of the special spatial coherence of the fractal or the fact that midpoint-subdivided fractals are a function of height (so the ray-tracer doesn't render them as fast as it could.)

You can also save a generated fractal as a regular psurf, using the command write fractal as psurf.

17.10 Network Jack
You can connect 2 Jack processes, each running on a separate workstation, but both sharing the same environment. Each figure in the environment is controlled by one Jack process, where it is considered a "player", and it is echoed in the other Jack's environment, where it is considered a "ghost".

The implementation uses TCP/IP stream sockets to transmit messages back and forth.

The normal cycle of messages between a player and its ghost is:

• Player changes position or posture, realizes it must update it's ghost.
• Player sends a Figure Position message to it's ghost.
• Ghost receives the message, updates the root and the joint angles.
• Ghost sends a reply to Player telling it's OK to send more position updates. The Player will not send any more updates until this reply is received. This will keep the two Jack processes roughly in sync, and update messages won't build up at the ghost.

Here are a list of the new Jack commands for using two networked Jack processes. Note that only two Jacks are supported now, but more could be added in the future.
open jack port: This command opens a connection to another Jack process. For example, one Jack process is started on a machine called bambi and another on a machine called thumper. On bambi, you would execute the command open jack port and you will be prompted to enter a host name. You enter self, meaning that this process will wait for a connection from another Jack process. On thumper, you execute open jack port, giving bambi as the host name. Now the two Jack processes are connected by a bi-directional stream socket.

close jack port: This will close the socket and kill any players that are left in the environment. All ghost figures will now be under local control (they aren’t ghosts anymore).

send JCL command: This allows you to send a JCL command to the other Jack process. For example, if you executed make window shaded on the local Jack process, you could execute send JCL command, then hit ^P (for previous command), and that same command is echoed to the remote Jack process. I usually bind this command to a key, for example F9, then I can execute a Jack command locally, press F9 ^P and the same command is executed in the remote Jack process.

make figure player: This command makes a local figure a “player” which means that it becomes a “ghost” in the remote environment. Whenever the figure moves in the local environment a message will be sent to the ghost to update its position (and posture). In the remote environment, the ghost figure will not be manipulable.

unmake figure player: This command will stop a figure from being a player, meaning that it will no longer send position update messages to its ghost in the remote environment, and the ghost figure in the remote environment will be under control of the remote Jack process (i.e. it can be manipulated).

kill player list: Jack maintains a list of figures which are also players. This command will execute the command unmake figure player on each of those players.

show player list: Simply prints out a list in the message window of the current players.

send message: This command prompts you to enter a string, then echoes that string in the remote Jack’s message window (highlighted. of course). This provides a simple means to “chat” with the remote Jack operator.

refresh player list: If for some reason the remote Jack process dies and needs to be restarted, once you’ve reconnected (using open jack port), you may use this command to run through the local player list, rebuilding the ghosts in the remote Jack environment.

send ping: This command will send a message to the remote Jack process and wait for a reply. The round trip time will be shown. Gives you a measure of the message delay between the two Jack processes.

17.11 Flock of Birds Interface

The Flock of Birds (referred to hereafter as the FOB) is a 6 degree of freedom position and orientation measuring system developed by Ascension Technology Corporation. This system may be used as an input to control figures in the Jack environment. A figure’s position and orientation can be bound to Bird input directly. These bound figures can then be used as the goals of constraints to drive articulated figures.

The interface between Jack and the FOB is divided into two parts; a Bird daemon process that communicates with the sensors and answers update queries from client processes, and the Jack simulation function, which queries the daemon and updates the corresponding figure in the environment. Once your system is properly configured, the Bird daemon will be started automatically when needed, and terminate when the Jack process has exited normally.

---

6 The Flock of Birds (referred to hereafter as the FOB) is a 6 degree of freedom position and orientation measuring system developed by Ascension Technology Corporation. This system may be used as an input to control figures in the Jack environment. A figure’s position and orientation can be bound to Bird input directly. These bound figures can then be used as the goals of constraints to drive articulated figures.

The interface between Jack and the FOB is divided into two parts; a Bird daemon process that communicates with the sensors and answers update queries from client processes, and the Jack simulation function, which queries the daemon and updates the corresponding figure in the environment. Once your system is properly configured, the Bird daemon will be started automatically when needed, and terminate when the Jack process has exited normally.

---

6 Mike Hollick
17.11. FLOCK OF BIRDS INTERFACE

17.11.1 Hardware/Operating System Information

The flockd daemon supports both standard and long range transmitter versions of the FOB. Each Bird should be connected via a separate serial port to the host computer. ERC's do not need to have a serial connection to the host; commands will be passed to it via the Fast Bird Bus. The ERC should have the highest address on the FBB.

There are several steps that must be taken to allow the flockd to operate normally. It is highly recommended that these are performed by someone familiar with Unix system administration.

1. Change the permissions on the serial ports you will be using. These ports are represented by the files /dev/ttyd?', where '?' is the serial port number. So, if you were going to be using 4 serial ports, you could issue the following command as root: `chmod ugo+rwx /dev/ttyd[1-4].`

2. Turn off terminal checking on the serial ports you will be using. By default serial port one is checked for terminal logins. Do do this you must edit /etc/inittab. You must change the entry for on-board port number 1 as such:

   # on-board ports
   tp:23:respawn:/etc/getty tport co_9600 # textport
   t1:23:respawn:/etc/getty ttyd1 co_9600 # alt console
   t2:23:off:/etc/getty -N ttyd2 co_9600 # port 2
   t3:23:off:/etc/getty -N ttyd3 co_9600 # port 3
   t4:23:off:/etc/getty -N ttyd4 co_9600 # port 4

   should be changed to:

   # on-board ports
   tp:23:respawn:/etc/getty tport co_9600 # textport
   t1:23:off:/etc/getty ttyd1 co_9600 # alt console
   t2:23:off:/etc/getty -N ttyd2 co_9600 # port 2
   t3:23:off:/etc/getty -N ttyd3 co_9600 # port 3
   t4:23:off:/etc/getty -N ttyd4 co_9600 # port 4

   All that is being done is replacing the respawn flag for port 1 with off.
   You should then kill the getty job that is currently checking port one, or simply reboot the machine.

3. Place an entry in /usr/etc/inetd.conf for flockd. This will allow flockd to be started automatically. This line should be added to the end of the file:

   flockd stream tcp nowait root /jack/bin/flockd /jack/bin/flockd >& /tmp/flockd.log

   NOTE: /jack/bin/flockd should be replaced with the proper location of the flockd program. It is located in the same directory as the Jack executable (normally $JACK/bin).

4. Add an entry in the /etc/services file so inetd knows what port flockd will be using. Add the following line at the end of this file:

   flockd 1212/tcp

5. Restart inetd to enable the above changes. Type /etc/killall -HUP inetd as root.

6. Create the flockd configuration file. To do this, simply use the flockd configure command and answer the questions. (Serial line multiplexing is currently not implemented, so the "multiplexing" question should be answered "no"). This will produce the flockd.config file. Put this file in the Jack bin directory.

7. Set the permissions on the kill_flockd program. This file should be setuid root, to allow any user to kill a runaway flockd.
17.11.2 Using the FOB in Jack

Before you can attach a figure to sensor input, the flockd must be started. Use the FOB initializeflock command to do this. When asked for a hostname, enter the name of the machine on your network that you are using to control the FOB. If the FOB is on your workstation, use “localhost” as the hostname. Once this is done you can use the FOB attach figure to bird command to initialize the sensor control. It should be noted that the FOB unattach figure from bird command must be used before deleting an attached figure!

17.11.3 Problems

If you have trouble connecting to the bird daemon, it may be the case that a previous daemon did not exit normally. This is usually caused when the Jack process exited abnormally. To kill the old daemon, execute the kill_flockd command on the FOB server machine.

A message log will be kept in the file /tmp/flockd.log. If you experience any problems, take a look at the log for useful information. This file should be deleted periodically, as it will become large.

Please note that this is an early version of this interface. Problems are sure to arise. Send any questions or problems to jack@graphics.cis.upenn.edu or call the University of Pennsylvania Computer Graphics Research Laboratory at (215) 898-1488.

17.12 Simple Psurf Editor

Currently, there is one contribution, a simple psurf editor, under the contrib ⇒ simple psurf editor menu. It allows you to view and edit a psurf as a list of vertices, edges, or faces. A hint: when editing a psurf, make sure that it is the only segment in the environment, and save frequently! Jack's eternal nemesis, the proverbial core dump, is known to frequent this area of the code...be forewarned.

17.13 Collision Avoidance Behaviors

The collision avoidance (CA) system is designed to avoid self collisions (e.g., hand-thigh collision), as well as the collisions of human figures with the environment. The main idea is that the obstacle (anything you want to avoid collisions with is an obstacle) produces a repulsive force when it is in contact with the human body. This force will push the body away and thus avoids collisions. Currently, CA is able to avoid collisions occurred in the human figure’s upper body (from waist and up, including hands, arms and the head).

17.13.1 Interactive Manipulation and Motion

Currently, CA works well during interactive manipulation. It should work equally well with the motion system after replacing the interpolation scheme of the motion system (which will be done soon). In the current implementation, CA is part of the human behaviors and it will not be active if human behaviors are turned off. However, CA is independent of other behaviors. So it can be made to work while other behaviors are off.

CA supports the default human figure in 5.8: human.fig.

17.13.2 Commands

The following commands are provided by CA on the contrib ⇒ human menu:

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>avoid collision</td>
</tr>
<tr>
<td>add one obstacle</td>
</tr>
<tr>
<td>turn all obstacles on/off</td>
</tr>
<tr>
<td>turn one obstacle on/off</td>
</tr>
<tr>
<td>turn collision avoid behavior</td>
</tr>
<tr>
<td>change collision avoidance params</td>
</tr>
</tbody>
</table>

---

7 Xinmin Zhao
17.13. COLLISION AVOIDANCE BEHAVIORS

avoid collision  This command initializes the internal data structure used by the collision avoidance system. If there are more than one human figure, the initialization should be done once for each human figure.

add one obstacle  CA handles obstacles at the segment level. An obstacle in CA is a segment the human agent wants to avoid. CA maintains an obstacle list. When CA is initialized (by executing the avoid collision command), it automatically adds segments in its own body to the obstacle list to avoid self collisions. These obstacles are initially inactive and they can be activated by the turn one (all) obstacle(s) on command (see below). In order to avoid collision with a segment in the environment, you need to add the segment to the obstacle list using the add one obstacle command.

turn all (one) obstacles on/off  CA only avoids collisions with active obstacles. These commands allow you to activate/deactivate one (all) obstacle(s).

turn collision avoid behavior  You may turn part or all of the CA on/off using this command. First, you are asked to identify the part of the body you want to change the collision avoidance behavior. It has the following choices:

| head   |
| left hand |
| right hand |
| torso |
| all parts |

Then you can specify when the behavior should be on:

| off       |
| on during animation only |
| on during interaction only |
| on all the time |

change collision avoidance parameters  The user has control over the following parameters:

move center of mass?  If this parameter is set to false, the human figure will avoid collisions by using upper body only and the center of mass stays at the same height. Otherwise, the human figure may lower its center of mass if necessary to avoid collisions. Default Value: true

restore posture?  If this option is set to true, the human figure will try to restore its original posture (standing up posture) when there is no obstacle preventing him from doing so. Currently, it only restores the center of mass and the joints on hand, arm, shoulder, head, neck. The torso is controlled by the “torso behavior”. (Note: If collision avoidance behavior is inactive or its restore posture flag is false, the torso behavior will not be active either.) Default Value: false

"using random-walk to escape from local minimum?"

CAS uses optimization techniques to avoid collisions. One problem associates with optimization procedures is that they may be trapped in a local minima position. In most cases this problem can be solved by random-walk. If this option is set to true, CAS will performs random-walk when it is trapped in a local minima position. Default Value: true

The following parameters are used to control the internal optimization process of CA. Their values should be changed only after the user understands the meaning of each parameter.
number of iterations for interactive session CA avoids collisions by minimizing potential energy. The underlying optimization routine is invoked many times until the potential energy is lower than a given threshold. In general, more iterations means better results and more time (worse performance). This parameter controls the number of iterations the optimization routine is invoked during interactive manipulation. In this situation performance is usually more important. So a small number should be given under this circumstance.

Default Value: 1

number of iterations for motion The same parameter as the previous one, except that it controls the optimization process during animation.

Default Value: 5

search range for interactive session in (0, 1) This and the next parameter are used to control the smoothness of the human figure's reaction after a collision. In general, larger value will cause a strong (or even over) reaction, while smaller value will cause a mild (or even no) reaction. All parameters are normalized to be in the range (0, 1).

Default Value: 0.05

search range for motion in (0, 1) Default Value: 0.02

17.14 Retina Windows

The retina window illustrates what the human figure "sees". The create retina window command opens two windows, respectively for the left and right eyes, that show the projections of the objects in the environment on the human figure's retina. You can open multiple pairs of retina windows for different human figures. The background of the retina window gives angular grid lines (concentric circles in 10 degree increments).

The retina windows will really slow rendering down, so you may want to make judicious use of the freeze window and thaw window commands. The reason for this: straight lines in the environment will create curves on the retinal projection; to achieve this, we must draw the curves piece-wise from the edges in the environment. This takes a relatively long time to do, and is done each interactive frame.

17.15 Interactive Mirrors and Shadows

There are several prototype commands on the contrib ⇒ visual effects menu which allow you to model interactive mirrors and shadows. They are not perfect, but we wanted to demonstrate this feature to generate interest among Jack users.

To demonstrate shadows and mirrors, there is a simple example in $JACK/jacklib5/demo/visualeffects/visual.jcl. If you read this file, it will exercise the commands on the visual effects menu. It reads a simple environment, with one light, a chair, and a ground plane. The shadows will be turned on and off, and the floor will be made reflective. Note: Shadows and mirrors are only drawn in the "effects window", not the regular peabody windows.
Appendix A

The Peabody Object Representation

Peabody is a representation for complex articulated geometric objects. It represents figures composed of segments connected by joints, also under the influence of constraints.

Jack is an interactive system for constructing and manipulating peabody objects. It is crucial to understand peabody before using Jack, since so many of the features of Jack deal intrinsically with peabody objects and rely heavily on the syntax of the peabody language. You may design peabody environments without using Jack at all, but Jack was designed to make this task easier.

A.1 Introduction to Peabody

A peabody environment consists of a number of individual figures, each of which is a collection of segments. The segments are the basic building blocks of the environment. Each segment has a "geometry." It represents a single physical object or "part", which has shape and mass but no movable components. The geometry of each segment is represented by a psurf, which is generally a polyhedron or a polygonal mesh.

The term "figure" applies not only to articulated, jointed figures such as a human body: any single "object" is a figure. It need not have moving parts. A figure may have only a single segment, such as a coffee cup, or it may be composed of several segments connected by joints, such as a robot. The term "object" is used here very loosely; it has no special significance. We use it only to denote some part of the peabody environment.

The term psurf refers only to the representation for the geometry of a segment, which is the graph of nodes and edges typically drawn as the wireframe or shaded image of the segment. In the case of a figure with a single segment, it is sometimes convenient to refer to it as a "psurf", but that is not technically correct.

Joints connect segments at attachment points called sites. A site is a local coordinate frame specified relative to the coordinate frame of its segment. Each segment may have several sites. Joints connect sites on different segments within the same figure. Sites need not lie on the surface of a segment, but generally they will. A site is a coordinate frame which has an orientation as well as a location.

A.1.1 The Connectivity of Peabody Objects

The peabody environment can be visualized as a directed graph. The segments are the nodes of graph, and the joints form the edges. Figures are maximal subgraphs spanned by joints. It is important to remember that joints do not connect segments directly. Joints connect segments through sites.

A.1.1.1 The Hierarchy of Peabody Objects

Peabody avoids representing objects with a strict hierarchy by encouraging you to think of figures as collections of segments and joints. However, there is an underlying hierarchy, and as in the case of moving a figure or adjusting a joint, it is important to remember what the hierarchy is because it has an effect on which objects remain fixed and which move when the joints change angles.
In order for the location of an articulated figure to be well defined, we must designate some element of the figure as its origin. The location of a peabody figure is specified through a site designated as the root. The root site roughly corresponds to the “origin” of the figure, and it provides a handle by which the location of the figure may be specified. Viewing the figure as a tree, the root of the figure is the root of the tree. The root site of a figure may be changed interactively in Jack from time to time depending upon how you want to manipulate the figure. This is important because the root of the figure serves as the origin of rotation and translation when you manipulate a figure interactively.

It is not allowable to define figures with closed loops of joints in peabody, although the syntax of the peabody language does not prevent you from doing so.

A.1.1.2 A Metaphor for the Connectivity of Peabody Objects

To visualize the graph of a peabody environment, imagine a collection of simple objects, such as machine parts, floating around in zero-gravity space. Several of these objects are connected to each other with hinges. The objects are the segments and the hinges are the joints. The placement of the hinges on the objects is described by the placement of the sites on the segments. A collection of segments hinged together form a figure. There may be several figures floating around. Some figures may consist of lots of segments and hinges. Other figures may have only a single segment. No segment is part of a figure unless it is hinged to the rest of the figure’s segments. You need not think of the segments and joints in the figure as having a strict hierarchy. The joints connect segments in completely arbitrary ways.

Each figure is nailed in place in space through its root site. The global placement of this field in space defines in turn the placement of all other segments in the figure. When you rotate a figure interactively, it rotates around its root site. When you translate a figure, it may move along the global coordinate axes of this frame. Jack allows you to interactively change the root, so if it becomes convenient to nail the figure down in a different way, it is possible to do so.

When you bend a hinge at one of the joints of a figure, the segments on one side of the joint will remain fixed, and the ones on the other side will move. Which side moves and which remains fixed depends on how the figure is rooted: the ones on the rooted site remain fixed.

We described the graph of the environment as directed, and the joints have a distinct direction. However, the directed-ness of a joint does not affect which segments stay fixed and which move as the joint angles change. The direction defines only the order in which the rotations of a joint are applied to produce the complete transformation between the sites which the joint connects. The details of joint definitions and degrees of freedom will be discussed in Section A.3.5.1.

A.1.2 The Geometry of Peabody Objects

It is very important to understand how the geometry is associated with a peabody figure. Each segment may have a psurf associated with it, but the shape of the psurf itself does not affect the topology of the figure. In Jack, it serves only as the image for the segment. The underlying topology of the figure, in terms of the “lengths” of the segments and the placement of the joints, is defined by the site locations relative to each segment, and it is completely independent of the psurf geometry. Sometimes sites will lie on the surface of the psurf, but this is by design rather than requirement.

There is no enforced relationship between the geometry and the length of a segment. In fact, segments don’t really have a “length.” If you define the length of a segment as the distance between the joints at either end, then this is only well defined when a segment has only two joints. But peabody allows segments to have any number of joints.

The geometry of a psurf is specified relative to the coordinate origin of the segment. This means that the \((x, y, z)\) coordinates of the vertices of the polygons of the psurf are interpreted and drawn relative to the axes of the coordinate frame of the segment, not the world coordinate frame. Each psurf is designed in its own coordinate system.

Take a moment to consider what \((x, y, z)\) cartesian coordinates really mean. These coordinates are only meaningful when interpreted in the context of a coordinate frame. Frequently, the coordinate frame is implicitly the world coordinate frame, but this not necessarily true. The \((x, y, z)\) coordinates of a point define the location of a point relative to the coordinate frame by specifying displacements from the origin
A.1. INTRODUCTION TO PEABODY

of the frame along the \( x \), \( y \), and \( z \) axes of the frame, respectively: start at the origin of the frame, travel \( x \) units down the \( x \) axis, then travel \( y \) units along the \( y \) axis, then travel \( z \) units along the \( z \) axis.

As an example, consider the two cubes shown in Figures A.1 and A.2. Along with each picture is the syntax for the psurf which defines the geometry. The details of the syntax for the psurf files is described in Section A.4. These two objects have the same "shape" but their origins are different. In Figure A.1, the origin is at the one of the corners. In Figure A.2, the origin is in the middle of the cube. It is important to realize that this difference is in the nodes of the psurf, not in the definition of the location of the figure itself. The origin of the psurf is implicit: it is with respect to this frame that the polygons of the object are interpreted. Peabody gives you great freedom in how psurfs are designed, since the sites may be placed anywhere on the segment. This gives you the ability to attach joints to segments in various places.

### Figure A.1: A Cube

```
0.00 0.00 0.00
0.00 100.00 0.00
100.00 100.00 0.00
100.00 0.00 0.00
0.00 0.00 100.00
0.00 100.00 100.00
100.00 100.00 100.00
100.00 0.00 100.00
;
1 2 3 4;
1 4 8 5;
3 7 8 4;
1 5 6 2;
5 8 7 6;
2 6 7 3;
;
```

### Figure A.2: Another Cube

```
-50.00 -50.00 -50.00
-50.00 50.00 -50.00
50.00 50.00 -50.00
50.00 -50.00 -50.00
-50.00 -50.00 50.00
-50.00 50.00 50.00
50.00 50.00 50.00
50.00 -50.00 50.00
;
1 2 3 4;
1 4 8 5;
3 7 8 4;
1 5 6 2;
5 8 7 6;
2 6 7 3;
;
```

This can sometimes be very confusing because it is the sites on the segment, and the joints which connect them, which define the location of the segment relative to other segments in the figure. Many times, it is convenient to define a site which lies at the coordinate origin of a segment. This is particularly true of an
elongated segment like an arm, which has a distinct proximal and distal end. In this case, the proximal end may lie at the coordinate origin of the segment, and the distal end may lie down the $z$ axis, for example. In this case, it may seem that the geometry is specified relative to the proximal end, but remember that the geometry is relative to the origin of the segment. The proximal site just happens to lie at that origin.

Drawing a diagram may solve lots of confusion. It is usually a good idea to draw a diagram of the segment with an explicitly labeled origin. Then draw the sites, such as proximal and distal, away from the origin, even if in fact they are coincident. This will help to reinforce the fact that they are specified relative to the origin of the segment.

### A.1.2.1 A Metaphor for the Construction of Peabody Figures

When designing the geometry of the segments of an articulated figure, it is best to proceed logically in a manner similar to what we might do if we were constructing a hinged mechanism out of wood, metal, or plastic. The most logical thing to do is to design each part one at a time. We begin by choosing an origin for the part and crafting its shape in terms of dimensions measured from that origin. After designing all of the individual moving parts, we proceed to drill holes in each part into which to bolt the hinges. Where do the holes go? We again calculate the location of the holes in terms of displacements from the origin of the segment. Next, we bolt the hinges into the holes and bend them to the correct angles (Let's assume that the hinges are stiff enough hold the figure in place). Finally, we choose a special point on the figure and place this point at the proper place so that the contraption is located in the desired position on the table, floor, or wall.

In the translation of this metaphor, the parts are the segments and the hinges are the joints. The shape of the part is defined in terms of a psurf, whose coordinates are interpreted relative to the coordinate origin of the segment. This coordinate origin is not relative to anything: other things are relative to it! The holes for the hinges correspond to the sites: they are measured from the coordinate origin of the segment. The special point on the contraption through which we fix it to the wall or floor corresponds to the figure's root site.

### A.2 The Mechanics of the Peabody Language

The syntax of the peabody language somewhat resembles a programming language, except that it defines static elements, not actions.

#### A.2.1 Arithmetic Expressions

The peabody language employs a powerful arithmetic expression parser and symbol table, so that any part of the language which requires a numerical value accepts a general arithmetic expression. The syntax of the expressions is similar to an ordinary programming language. The operators and their precedence are shown in Figure A.3. The operators at the top have the greatest precedence.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>parentheses</td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
</tr>
<tr>
<td>^</td>
<td>exponentiation</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
</tr>
</tbody>
</table>

Figure A.3: The Precedence of Peabody Arithmetic Operators

Variables need not be declared before use: their type is determined by context, although a warning message will be issued if a variable is used before being assigned a value.
A.2.2 Units

Many numerical values in the peabody language refer to physical measurements: angle, distance, mass. Peabody allows such values to be quantified by their type, so that values may be entered in any particular units. Whenever a value is specified, its units should also be specified so there is no confusion about which units are being used. This convention will save much confusion as files are written as used at later dates.

For angles, the legal types are radians and degrees, using the keywords `rad` and `deg`. The default is degrees. For distances, the legal types are millimeters (`mm`), centimeters (`cm`), meters (`m`), inches (`in`), feet (`ft`), and yards (`yd`). The default is centimeters.

A.2.3 Homogeneous Transformations

Peabody relies heavily on the specification of homogeneous transformations, and the peabody language has a rather simple mechanism for describing such transformations. Transformations may be expressed as a sequence of simpler, primitive transformations such as rotation and translation. The translation operator is `trans`, and it takes three arguments, giving the translation in x, y, and z. Rotation may be described with the `xyz` operator, which specifies rotation in terms of angles around the local x, y, and z axes, in that order. For example,

\[
t = \text{xyz}(10\text{deg},20\text{deg},30\text{deg});
\]

specifies a rotation transformation which is formed by a rotation of $10^\circ$ around the x axis, followed by a rotation of $20^\circ$ around the rotated y axis, followed by a rotation of $30^\circ$ around the rotated z axis. This operator can be used for simple rotations around a single coordinate axis by using zeros for two of the angles.

![Figure A.4: The xyz Rotation Operator](image)

A.2.3.1 Multiplying Homogeneous Transformations

The peabody language represents homogeneous transforms as $4 \times 4$ matrices in the form:

\[
\begin{bmatrix}
z_0 & z_1 & z_2 & 0 \\
y_0 & y_1 & y_2 & 0 \\
z_0 & z_1 & z_2 & 0 \\
p_0 & p_1 & p_2 & 1
\end{bmatrix}
\]

The translation vector $(p_0, p_1, p_2)$ is in the bottom row of the matrix.

When transformations are multiplied, the product may be interpreted in two ways. The most intuitive way corresponds to local transformations when applied right to left. For example, the transformation given by

\[
t = \text{xyz}(90\text{deg},0,0) \cdot \text{trans}(0,100\text{cm},0);
\]
may be interpreted as first a translation of 100 cm along the y axis, followed by a rotation of 90° around the translated x axis. The ordering is critical, since transformations don’t commute.

Alternatively, the product of transformations may be interpreted in global coordinates when read left to right. The above transformation may be interpreted as a rotation of 90° around the original y axis.

When Jack writes a transformation, it writes the rotation part followed by the translation part, expressed in terms of the xyz operator, no matter how the expression was originally specified. This product can be interpreted as a rotation followed by a translation with respect to the base coordinate frame, or alternatively a translation followed by a rotation around the translated axes.

The exponentiation operator may be applied to transforms, with the usual meaning. Raising a transform to the power of -1 yields the inverse of the transform.

A.2.3.2 Manipulating Transformations Interactively in Jack

You can experiment interactively with these transformations in Jack by moving a figure. As you move the figure, the peabody description of the figure's root appears at the bottom of the window. You can see the description of the transformation as it is being manipulated.

Another valuable exercise is to enter the transformations from the keyboard by hitting -K while moving the figure. You will be prompted in the message window to enter the transformation. Experiment with several transformations to get a feel for how this syntax works, particularly in terms of the ordering of the transformations. The details of how to enter values from the keyboard in Jack are described in Section 5.8.

A.2.4 Names and Identifiers

All constructs in the peabody language have names, and each name is local to the construct to which it belongs. This ambiguity may be resolved by prefixing the name of construct with the name of its parent, separated by a period. For example, each figure in the environment has a distinct name, and each segment within each figure has a name which is unique only within that figure. Therefore, two figures, say fred and ethel, may have segments named head. The identifier head does not uniquely specify which head, so the use of head must be prefixed with either fred or ethel, as in fred.head or ethel.head.

Likewise, joints are local to the figures to which they belong, and sites are local to the segment to which they belong. In the case of sites, a double prefix must be used to uniquely specify a site in the environment. This follows a general rule of thumb that when constructs are named, they must be properly qualified given the context. When naming segments within a figure, the figure name is usually understood and may be omitted.

One exception to the above rule is the pseudo-segment world, which is the base coordinate frame for the environment. The world is not a part of any figure, so a reference to one of its sites has only one prefix, as in world.base.

Names of constructs and variable identifiers must consist of upper and lower case alphabetic characters, digits, and underscores, and must not begin with a digit. It is important that names of constructs in peabody do not conflict with keywords. Since most keywords are frequently used terms, this rule must be carefully considered when constructing peabody figures. The peabody keywords are listed in Table A.1.

It is important to remember the difference between identifiers and strings, especially in places in the peabody language where file names are required. All strings, such as file names, must be enclosed in double quotes. Otherwise, they would be interpreted as variable references.

A.3 Peabody Construct Declarations

The term “construct” refers to any of the basic elements of the peabody object representation: figures, segments, sites, joints, attributes, and lights. The peabody language consists primarily of assignment statements which define the properties of the peabody constructs. Each assignment must be qualified with the construct to which it applies. This is generally done by grouping the assignments together into blocks which set forth the construct name and type, similar to a data structure declaration in a programming language. The elements within the block are then automatically associated with that construct.
The peabody construct declarations create objects implicitly, not explicitly. Figures, segments, sites, and joints are created when they are defined for the first time. In this sense, the “definitions” of the constructs are not really definitions by declarations. It is entirely legal to have duplicate declarations for the same object. You should think of the process of reading a peabody file as a sequential stream of assignment statements which set the values of object parameters. The objects are created when they are first referenced. Duplicate assignment statements reset the parameter values.

A.3.1 Figure Declarations

A figure declaration begins with the keyword `figure`, followed by an identifier. A figure declaration may not occur inside any other block. Inside the figure block there may be:

- segment declarations
- site declarations
- joint declarations
- light declarations
- attribute declarations
- posture declarations

Site declarations must have names qualified with the segment name unless they are contained in a segment block. In addition, figures have the following assignment fields:

- **archive** The archive specifies an archive file in which to look for psurf files for the segments in the figure. This field is optional, but if it is used it should occur before any segments which reference psurfs in the archive. The value is a character string giving the name of the archive file.

- **root** The root specifies the root site for a figure. This is the site with respect to which the figure’s `location` field describes its global placement. The value is the name of the site.

- **location** The location gives the transform which defines the global placement of the figure’s root. The placement of all other sites and segments in the figure is determined from this. The value is a homogeneous transformation.
A.3.2 Segment Declarations

A segment declaration begins with the keyword segment, followed by an identifier. A segment declaration may occur inside a figure block. If it lies outside a figure block, its name must be qualified with a figure name. Site declarations and attribute definitions may occur inside the segment block.

In addition, segments have the following assignment fields:

- **psurf**: The psurf gives the file which describes the geometry of the segment. The value is a character string which is the name of the psurf file.
- **attributes**: The set of attributes describes which surface attributes are to be associated with the psurf. The value is an array of attribute names, separated by commas. The number of attributes in this array must match the number of attribute indices used in the psurf file. If the psurf has a single attribute, then the value may be just the attribute name, and you may use the keyword attribute, without the 's'.

Peabody has a built-in set of rules for where it looks for psurf files. See Section 9.3 for a description of these rules. See Section A.4 for a description of the syntax of the psurf files.

A.3.3 Path Declarations

A path is a set of n sites on a segment, named \{pnt0,pnt1,\ldots,pntn\}, with one additional site called point. The peabody description of a path looks like:

```plaintext
segment paths {
    site base->location = trans(0cm,0cm,0cm);
    site point->location = trans(0cm,0cm,0cm);
    site pnt0->location = trans(0cm,0cm,0cm);
    site pnt1->location = xyz(0deg,0deg,-38deg) * trans(3cm,62cm,-11cm);
    site pnt2->location = xyz(-180deg,-75deg,141deg) * trans(45cm,62cm,-86cm);
    site pnt3->location = xyz(-180deg,-10deg,141deg) * trans(112cm,62cm,-1cm);
    site pnt4->location = trans(136cm,62cm,75cm);
    path = ("spline", 31, (site)point,
           (site)pnt0, 0,
           (site)pnt1, 0.25,
           (site)pnt2, 0.50,
           (site)pnt3, 0.75,
           (site)pnt4, 1);
}
```

The path field of the segment defines the interpolation method ("spline"), the number of generated samples (default 31), the point site, and the set of sites with their corresponding time values. The point site is used when animating a path, and the point site moves to a position on the path corresponding to a time fraction between 0 and 1 (i.e. the spline is evaluated at some time between 0 and 1, and the point site is moved to that location.

A.3.4 Site Declarations

A site declaration begins with the keyword site. A site declaration may occur inside a segment block. If it lies outside, its name must be fully qualified with the segment name. If it does not lie within a figure block, the name must also be qualified with the figure name.

A site has a single following assignment field, giving its location.

- **location**: The location is a transform describing the position and orientation of the site with respect to the coordinate frame of its segment. The value is a homogeneous transformation.
A.3.5 Joint Declarations

The declaration of a joint begins with the keyword `joint`. A joint connects two sites on two segments within the same figure, and represents a transformation between segments.

The principle statement in a joint declaration is the `connect` statement, which is not in the form of an assignment. It specifies the two sites which the joint connects, and it takes the form of "connect site1 to site2"

The joint has the following assignment fields:

- **type**: A functional expression describing the degrees of freedom of the joint. The value of the expression is described below.
- **displacement**: A vector of numbers specifying the current angles of the joint, having as many elements as the joint has degrees of freedom.
- **ulimit**: A vector of numbers describing the upper limit of the angle or distance of each degree of freedom, having as many elements as the joint has degrees of freedom.
- **llimit**: A vector of numbers describing the lower limit of the angle or distance of each degree of freedom, having as many elements as the joint has degrees of freedom.

By default, the transformation at a joint is an arbitrary homogeneous transformation, which specifies the relative position of the two sites which the joint connects. The `connect` statement specifies that the joint connects one site to another, and this defines the direction of the joint and the joint's displacement. The first site in the connect statement is sometimes called the “from” site; the second site is called the “to” site. The joint's displacement transform defines the global placement of the “to” site relative to the global placement of the “from” site. The directionality of the joint is defined in terms of how the joint would normally behave in the figure, when the figure is rooted in its normal place. By “normal” behavior, we mean that when you interactively adjust a joint, the “from” side remains stationary and the “to” side moves. This directionality remains fixed even if the figure is re-rooted and the actual figure hierarchy is changed.

For example, in the human figure model, the shoulder is defined to connect the clavicle to the upper arm. This definition is convenient since we normally expect that when we change the transformation at the shoulder, the arm will move and the upper torso will remain fixed. This will not be the case, however, if the figure is rooted through his hand, possibly hanging from a rope or floating in zero gravity. In this case, we still define the transformation across the joint in the direction “from” the clavicle “to” the arm.

A.3.5.1 Degrees of Freedom

Joints may have specific degrees of freedom which restrict the transformation across the joint. The type of the joint is specified by the `type` field, whose value is an arbitrary expression composed of primitive rotation and translation operators. The rotation operator is R, and the translation operator is T. Each rotation and translation specifies an axis. The axis must be a coordinate axis of unit length: (1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1), or (0,0,-1).

The current angle associated with the axis comprises the `displacement` of the joint and is not part of the type. A simple joint may be defined as:

```plaintext
joint elbow {
    connect lower.arm.proximal to lower.arm.distal;
    type = R(1,0,0);
    displacement = (90deg);
    ulimit = (180deg);
    llimit = (0deg);
}
```

This joint rotates only around the x axis. The displacement field specifies that the transformation is a 90° rotation around the x axis.

---

1The global placement of the “from” site is determined in turn by its location relative to its segment’s base coordinate frame. This in turn is determined by other site locations and joint displacements, up the figure hierarchy to the root of the figure, where the global placement is fixed.
Joints may have up to three degrees of freedom by multiplying primitive operators. In this case, the displacement field has the same number of arguments as the type expression has primitive elements. The complete transformation at the joint is the product of each primitive operator instantiated with the appropriate angle. As in the case of homogeneous transforms, the operators should be interpreted right to left as local transformations, i.e., with respect to the local, or current, transform. Alternatively, the displacement at the joint may be interpreted from left to right as primitive transformations with respect to the parent coordinate frame, that is, the site on the "from" side of the joint.

For example, the transformation at the joint defined by:

```
joint luceille.ball.joint {
  connect torso.shoulder to arm.base;
  type = R(0,0,1) * R(0,1,0) * R(1,0,0);
  displacement = (90deg,45deg,30deg);
  ulimit = (180deg,90deg,60deg);
  llimit = (0deg,0deg);
}
```

may be interpreted as a rotation of 30° around the x axis of site torso.shoulder, followed by a rotation of 45° around the rotated x axis, followed again by a rotation of 90° around the rotated y axis. It may alternatively be interpreted as a rotation of 90° around the x axis of the site torso.shoulder, followed by a rotation of 45° around the y axis of the same frame, followed by a rotation of 30° around the x axis of the same frame as well.

### A.3.6 Constraint Declarations

Constraints specify desired geometric relationships. The parameters of constraints are described in Chapter 10. The syntax of the constraint declaration in the peabody language has an assignment statement for each of the constraint's properties. There can be no blocks within a constraint block.

A constraint has the following assignment fields:

- **type** The relationship type. The expression can be either a single string giving the position or orientation type if there is only one, or it is a 3-vector, with the first element a string describing the orientation relationship type, the second a string describing the position relationship type, and the third element a number giving the position/orientation weight.
- **end** The end effector. The value may be a site or a node. It must be typecast to determine which type it is.
- **goal** The goal. The value may be a site, node, face, or a matrix. A matrix type specifies a hold constraint. Otherwise, it must be typecast to determine which type it is.
- **startjoint** The starting joint. If this field is absent, then the constraint is a rooting constraint.
- **weight** The constraint weight.

### A.3.7 Surface Attributes

Each polygon in the peabody environment has a surface attribute associated with it, and the surface attributes may be specified in the peabody file. Surface attributes are defined in a block structured manner similar to the other peabody constructs:

```
attribute brown {
  diffuse = (0.48,0.26,0.00);
}
```

Surface attributes have the following assignment fields:

- **ambient** This is a triplet of real numbers describing the ambient color of the surface in RGB coordinates.
- **diffuse** This is a triplet of real numbers describing the diffuse color of the surface in RGB coordinates.
specular This is a triplet of real numbers describing the specular color of the surface in RGB coordinates.

glossiness This is an integer exponent describing the glossiness.

Alternatively, you may specify the ambient and diffuse parameters as scalar values and give the attribute a color with the rgb field. This makes the ambient and diffuse parameters different intensities of the same color. This is the most common way of describing attribute parameters.

The way in which the surface attribute parameters affect the shading of the surface is described in Section 12.

A.3.8 Light Source Declarations

Light sources are necessary for rendering and are an integral part of a computer graphics and animation environment. Light sources are represented in peabody as special types of segments. A light declaration begins with the keyword light. Usually, a light source will consist of a figure with a single segment, but it is possible to give light source properties to any segment on any figure. The origin of the light is the origin of the light segment, and the light is emitted equally in all directions. Lights are not directed. In the peabody grammar, lights are interchangeable with segments:

```peabody
figure foo {
  light bar {
    site base->location = trans(0cm,0cm,0cm);
    color = (1,1,1);
  }
}
```

Lights have the following assignment assignment fields:

ambient This is a triplet of real numbers describing the ambient color of the light in RGB coordinates.

color This is a triplet of real numbers describing the color of the light's emission in RGB coordinates.

The way in which the light source parameters affect the shading of a surface is described in Section 12.

A.3.9 Peabody Block Structure

The block structure of the peabody language facilitates prefixing constructs with their “parent” construct by effectively prefixing everything within the block with the appropriate construct name. Thus, within a figure declaration, all segment names are taken relative to the figure, so there is no need to prefix them. Within a segment declaration, the site names are taken relative to the segment. The joint declaration references sites, which must always be qualified at least to the segment level.

The block structure exists primarily for convenience, since all constructs may occur in any arrangement provided they are fully qualified. For example, a segment block normally belongs inside a figure block, but it may occur outside, provided the segment name is prefixed with the proper figure name.

The curly braces serve to prefix each assignment statement in a construct with the proper construct name and type. They may be replaced by the “arrow” notation:

```peabody
segment cube->psurf = "cube.pss";
```

This short form may be used for any assignment field in the construct.

The traditional block structure resembles a “definition,” but actually each construct is defined whenever it is first used. There is no formal distinction between definitions and references. Any subsequent references to the construct refer to the original rather than creating a new one. This allows the same file to be read multiple times without side effect. The effect of reading a peabody file multiple times is to reset the values of the assignment fields rather than create new constructs. A common use of this is for separating joint displacements from figure definitions.

This also allows for very terse definitions. For example,
is a valid figure definition.

As Jack reads a peabody file, it sets the given construct field regardless of its previous value. An exception to this rule applies to psurfs. If a psurf for the segment already exists and it was read from a file of the same name, then the psurf is not re-read. This allows figure definitions which include psurf specifications to be re-read without the overhead of re-reading the psurfs.

A.3.9.1 Figure Files

Sometimes it is convenient to create a definition for a certain type of figure, that is, a template which can be used to create different instances of the figure.

A single environment file cannot be used to create different instantiations of the same figure definition, since the names are associated directly with the figures, and figures must have unique names. Peabody allows you to define figures in special figure files. Different figures may be instantiated from a figure file with different names.

A figure file is simply a file containing the definition of a single figure, without the identifier following the figure keyword. For example, consider a file called thing.fig:

```plaintext
figure {
    segment x {
        psurf = "cube.pss";
        site base->location = trans(0,0,0);
        site p->location = trans(100,100,100);
    }
    segment y {
        psurf = "pyramid.pss";
        site base->location = trans(0,0,0);
    }
    joint elbow {
        connect x.p to y.base;
    }
}
```

Now, another file called, say things.env, could contain:

```plaintext
figure ["thing.fig"] a;
```

The syntax of the figure instantiation specifies the name of the figure file, thing.fig, in square brackets, followed by the name of the instantiated figure, a. Note that the name of the file must be enclosed in double quotes.

The file things.env may include an additional “thing”:

```plaintext
figure ["thing.fig"] a;
figure ["thing.fig"] b;
```

A.3.9.2 Figure Parameters

Figures may have parameters, much the same as function parameters in a programming language. This makes it possible to generalize a figure template to be instantiated with different sizes, types, etc. For example, the above figure may have been defined as:

```plaintext
figure (height,width) {
    segment x {
        psurf = "x.pss";
        site base->location = trans(0,0,0);
        site p->location = trans(width,height,0);
    }
    segment y {
        psurf = "y.pss";
    }
}
```
A posture block is declared within a figure block. It's purpose is to capture a posture (joint angles, figure location, root, and constraints and behaviors) for a particular figure. There are two forms of a posture declaration inside a figure block. The first one is called a posture reference, and uses the keyword postureref, and looks like:

```plaintext
figure human {
    postureref ["standing.post"] (10cm) standup;
    postureref ["sitting.post"] sitdown;
}
```

The above declaration adds two named postures to the figure human, and they are named standup and sitdown. The important point to know here is that a postureref declaration will not read the posture files (in this case, .post) when the postureref is read. It only adds the named posture (along with filename and arguments) to the list of postures attached to the figure. This is important, since the posture definition file (.post file) may contain references to peabody objects that do not yet exist in the environment. The posture file is read when a Jack command requests the posture to be instantiated. This also allows postureref declarations to be stored in figure definition files (.fig files).

The other form of a posture declaration is called a posture instantiation and looks just like above, except postureref is replaced with posture. It would look like:

```plaintext
figure human {
    posture ["standing.post"] (10cm) standup;
    posture ["sitting.post"] sitdown;
}
```

The difference here is that when the declaration is read, the posture file is read, and therefore the posture is instantiated (the figure is moved to the corresponding posture). In the above example, the figure would be moved to the standup posture, then the sitdown posture.

Posture definition files can be parameterized just like figure files and motion group files (see above and below). In the example above, the posture file standing.post takes one parameter, a length measurement. The file standing.post could look like:

```plaintext
posture (length) {
    root = lower.torso.distal;
    joint left.shoulder->displacement = ... 
    ... 
    location = trans(length, 0cm, 0cm);
}
```

A posture file must always start with posture, not postureref. Inside a posture block, you may have joint displacements, site locations, constraints, behaviors, root specifications, etc (any field or block which may appear in a figure block may appear in a posture block). Note that all segment/site/joint names are relative, they are not prefixed with the figure name. This allows posture files to be shared between similarly structured figures (i.e. the human).
A.3.11 Motion and Motion Group Declarations

The declaration of a motion begins with the keyword `motion`, followed by a name for the motion. The naming convention is the same as that for segments. A motion should not be declared inside any other peabody construct, except a `motiongroup` or at the top level by itself (in which case it belongs to the `default` motion group. A sample motion in peabody format looks like:

```peabody
motion hand {
    figure = (figure)human5;
    type = "hand control";
    starttime = 0sec;
    duration = 2sec;
    off = 1;
    velocitycontrol = "ease in/ease out";
    data = ("right","site",(site)hand.paths.point,"waist");
}
```

Every motion type has a set of common fields (figure, type, starttime, duration, off, and velocitycontrol). Additionally, each motion type defines a field called data which is simply a vector of values. This vector is generated by the motion to save whatever parameters it needs. It is also parsed by the motion when being read from a peabody file. The only restriction on the values is that it must be a valid parse-able peabody value. The assignment fields for a motion must appear in the order that they are shown above. The assignment fields for a `motion` are:

- **figure**: This is a figure reference to the figure to which this motion is applied. Since this is a figure reference, you must precede a figure name with the `(figure)` casting operator. If this is a peabody variable, it must be a figure reference. (See the `motiongroup` below).

- **type**: The type field is a string describing the type of this motion. The type tells peabody how to interpret the data field below. Please see the table below for the values of type, and the corresponding format of the data field.

- **starttime**: This is the time, in seconds, when the motion will start. This may be any valid peabody expression which resolves to a number.

- **duration**: An expression, evaluating to the duration, in seconds, of the motion.

- **off**: This field is optional, and if 1, means the motion is actually turned off (it won't execute).

- **velocitycontrol**: For motions which interpolate between a beginning and ending value, this is the method for controlling the velocity function of the interpolation. Valid values are: "constant", "accelerate", "decelerate", "ease in/ease out".

- **data**: The data field is a vector of values representing the information needed by each motion type. It is different for each type. Below is a few examples from the simpler motion types:

  - **joint**: \((n, jointname_1, (displacement_1) \ldots jointname_n, (displacement_n))\)
    
    \(n\) is the number of joints involved. For each joint involved, the `jointname_i` is listed, followed by a vector giving the goal joint displacement.

  - **figure**: \((xform)\)
    
    The `xform` is the global transform for the goal of the figure motion.

  - **camera**: \((xform, windowname, vrdp)\)
    
    The `xform` is the global transform for the camera, the `windowname` is the name of the peabody window, and `vrdp` is the view reference point depth for the camera.
light \((segmentp, (s_R, s_G, s_B), (e_R, e_G, e_B))\)

The \(segmentp\) is a segment pointer (to the light source), followed by the starting rgb color vector \((s)\), and the ending color vector \((e)\). If \(s\) is \((-1, -1, -1)\), the starting color is just the current color of the light source (when the motion starts).

item[\(\text{path}\)] \((segmentp, \text{traversal})\)

The \(segmentp\) is a segment pointer to the path, followed by \(\text{traversal}\), which can be either \'forward\' or \'reverse\'.

item[\(\text{figure} \ \text{path}\)] \((segmentp, \text{traversal}, \text{figurep})\)

Same as above, except \(\text{figurep}\) points to the figure which moves along the path.

command \((\text{preaction JCL}, \text{apply JCL}, \text{postaction JCL})\)

These are three JCL command strings. The \(\text{preaction JCL}\) is executed on the first frame of the motion, the \(\text{apply JCL}\) is executed on each frame of the motion, and the \(\text{postaction JCL}\) is executed on the final frame of the motion. If the JCL has double-quotes, they must be appropriately escaped (i.e. " becomes ").

Each motion can also be a member of a motion group. Motion groups can also be parameterized similar to the way figure files can be parameterized (See Section A.3.9.1). An example motion group file:

```
motiongroup (fig, start, durate) {
  motion arm4_arm1 {
    figure = fig;
    type = "joint";
    starttime = start + (0 \* durate);
    duration = 0.67 \* durate;
    velocitycontrol = "constant";
    data = (3, jointref(fig,"joint4"),(0.00deg,0.00deg)
           ,jointref(fig,"joint3"),(0.00deg,0.00deg)
           ,jointref(fig,"joint2"),(0.00deg,0.00deg)
           );
  }

  motion chain {
    figure = fig;
    type = "figure";
    starttime = start + (0.33 \* durate);
    duration = 0.67 \* durate;
    velocitycontrol = "constant";
    data = (trans(5.62cm,0.00cm,-202.27cm));
  }
}
```

If this motion group were in a file call \(\text{move.mgp}\), then a valid peabody reference to create this motion would be:

```
motiongroup ["move.mgp"] ((figure)chain, 0sec, 1.50sec) move;
```

The motiongroup structure is useful for grouping related motions together, and the intention of it is also to be used for creating composite motion templates. But there is a major problem ... The problem is that peabody name references within the data field of motions need to be relative to the figure of the motion or relative to some parameter of the motiongroup.

A solution to this problem is to use the peabody built-in functions (siteref), segmentref, and jointref. These function takes two arguments, the first can either be a figure name (string) or a figure reference, and the second is a relative site/segment/joint name (i.e. of the form segment.site, segment, or joint respectively). These function will return a reference pointer to the actual object (site, segment or joint).
A.3.12 The Include Statement

The `include` statement nests a peabody file in another file, similar to the `#include` preprocessor control statement in the C programming language. Its format is simple:

```
include "file.env";
```

The effect is identical to inserting the entire contents of the file into the original file at the location of the `include` statement. Files may be nested to a level of 8.

A.4 The Syntax of Psurf Files

Psurfs may be described syntactically in text files. A psurf file is a textual representation for the nodes and faces of the psurf. By convention, these files have the suffix `.psf`. The file lists a set of nodes and faces.

```plaintext
{ a cube: 8 nodes, 6 faces }
0.00 0.00 0.00
0.00 100.00 0.00
100.00 100.00 0.00
100.00 0.00 0.00
0.00 0.00 100.00
0.00 100.00 100.00
100.00 100.00 100.00
100.00 0.00 100.00

;; { end of nodes }
1 2 3 4; { back }
1 4 8 5; { bottom }
3 7 8 4; { right side }
1 5 6 2; { left side }
5 8 7 6; { front }
2 6 7 3; { top }

;;
```

Figure A.5: An Example psurf

A psurf file begins with a list of nodes, which are specified as triplets of real numbers. There may be an optional comma between the triplets. The numbers may contain decimal points, but the decimals are not necessary. No leading 0 is required for fractions less than 1.0. The nodes are numbered implicitly starting at 1. The node table is terminated with two semicolons.

Following the nodes are the faces, which are lists of indices into the nodes. Each list specifies the vertices of the face, and is terminated with a semicolon. The faces are terminated by an empty vertex list, i.e. two adjacent semicolons. There is a predefined limit of 256 vertices in each face. The vertices should be in counter-clockwise order. This is discussed in greater detail in Section A.4.5.

Comments may appear anywhere in the file and are delimited by curly braces. The indices listed in the psurf file all start at 1 for historical reasons. A single psurf file may contain several individual "psurfs." In other words, it is legal to concatenate several psurf files into one, which will make a single psurf out of several disconnected components.

Psurfs may be in any general scale, but it is best to make the coordinates correspond to centimeters. Jack assumes that all objects are scaled in centimeters, and it initializes the view so that objects in the same general scale as human bodies are displayed conveniently on the screen. You can change this scale if necessary, but it is still best to keep all object in the same basic range of sizes.
A.4. THE SYNTAX OF PSURF FILES

A.4.1 Surface Attributes

Between the index of the last vertex of each face and the semicolon which ends the face, there may be an attribute specification, which is the keyword attribute followed by an index into the psurf’s attribute table, all delimited by square brackets. By default, the attribute index is 0, and its value carries over from one face to the next, so the attribute specification actually sets the “current” value, to be assigned to all subsequent faces until its value is changed again.

This is only part of the attribute information. The psurf itself does not define the material properties associated with each face, but it does specify which faces are made of the same material. When a psurf is associated with a segment, it will be instantiated with a set of surface attributes. The number of attributes will match the number of attribute indices specified in the psurf file, and the attributes will be associated with the faces according to each face’s attribute index. The details of how attributes are associated with faces is described in Section A.3.7.

A.4.2 Face Smoothness

Along with the attribute specification, each face has a smoothness flag. This flag tells whether the face models a flat surface like a polyhedron, or whether it is being used to model a small piece of a curved surface. In computer graphics, it is sometimes convenient to model curved surfaces with a mesh of small polygons. The polygons themselves are an artifact of the model, not a geometric property.

When an object is drawn in wireframe, it’s drawn with its edges. When it is shaded, its faces are filled in. If a face is flat, it has a constant surface normal across the entire face. If the face is smooth, the surface normal is computed at each of the vertices of the face in terms of the normals of the adjacent faces, and the normal at points in the interior of the face is interpolated from the values at the vertices. This is phong shading.

It is important not to use smooth shading with psurfs which are not polygon meshes. If a psurf consists of a few big faces and the angles between them are great, then the object will not look right if it is smooth shaded.

A.4.3 Node Coloring

It is also possible to assign rgb color values to individual nodes in a psurf file. The format is to place an rgb value after the node definition in the psurf file. A cube, with 7 white vertices and one red vertex, would look like the following:

```
0.0 0.0 0.0 [rgb(1.0, 1.0, 1.0)]
0.0 100.0 0.0 [rgb(1.0, 1.0, 1.0)]
100.0 100.0 0.0 [rgb(1.0, 1.0, 1.0)]
100.0 0.0 0.0 [rgb(1.0, 1.0, 1.0)]
0.0 0.0 100.0 [rgb(1.0, 1.0, 1.0)]
0.0 100.0 100.0 [rgb(1.0, 1.0, 1.0)]
100.0 100.0 100.0 [rgb(1.0, 1.0, 0.0)] {the red one}
100.0 0.0 100.0 [rgb(1.0, 1.0, 1.0)]

1 2 3 4 [smooth] [attribute 0];
1 4 8 5 [smooth] [attribute 0];
3 7 8 4 [smooth] [attribute 0];
1 5 6 2 [smooth] [attribute 0];
5 8 7 6 [smooth] [attribute 0];
2 6 7 3 [smooth] [attribute 0];
```

You can use the command set node rgb to edit the color of a node. Note that you should use make segment smooth to properly blend the node colors across a face, using Gouraud shading. Note that lighting will
not effect segments that have node colors. Jack uses the node colors to display the results of a radiosity rendering, therefore the lights are disabled in the environment for those segments which have node colors.

A.4.4 Big Psurfs

A psurf file may also have an optional specification [big] on the first line of the file. This specification will inhibit Jack from generating display lists for the edges of the psurf. This is useful if the psurf is imported from other CAD system that can't break up geometry and must send it as one long list of nodes and faces. Jack usually creates edge lists for displaying psurfs in wireframe. This spec is only necessary if the psurf has more than 1000-2000 nodes.

A.4.5 Face Orientation

Psurf faces have a specific orientation, which means that a face really has just one side. When viewing an object in wireframe, this is of no consequence since the object is displayed with its edges. However, when an object is shaded its intensity is defined in terms of the angle between the face and the light sources. The orientation of the face depends upon the order of the vertices. The surface normal is defined to point in the direction given by the right hand rule. This means that a counter-clockwise traversal by the right hand rule yields an outward pointing normal. In other words, place your right hand on the polygon and sweep your fingers around the vertices in the order they occur in the file. Your thumb then points in the direction of the surface normal.

It is imperative that the faces of psurfs be oriented consistently. When objects are shaded in Jack, only the side of the surface normal will be shaded correctly. The other side will be effectively shadowed and will be illuminated with ambient light.

Psurfs usually represent collections of polyhedra, so the nodes, edges, and faces form a planar graph. This places some important restrictions on how the faces may be defined:

- Each edge (pair of nodes) should be contained in at most two faces.
- No pair of nodes (an edge) should be contained in two faces in the same order.

This means that if you traverse each face of a psurf, you should never travel along the same edge more than twice, and you should never travel along the same edge twice in the same direction. If you have, the psurf does not define a valid polyhedron, and if you try to shade it, its surface normals are not well-defined.

There are exceptions to these rules, particularly in the case of open objects, i.e. sets of faces which are connected but which do not bound a closed region in space.

A.4.6 Binary Psurfs and BPS

For faster reading and display, you may pre-process psurf files and convert them into “binary psurfs” using the program bps. bps reads a text psurf file and produces another file with the same base name but with the suffix .bps. This is a binary psurf file, and it represents the same geometric information about the psurf in a form which is easier for Jack to read and more efficient for it to display.

---

2A planar graph is a graph which is capable of be drawn in a plane without crossing edges. It does not mean that all nodes of the graph lie in a plane in space.
Appendix B

Ray Tracing

Dawn Vighott
Welton Becket

B.1 Introduction

This guide is intended for those just beginning, who are familiar with the basic concept of raytracing but are unaware of how to use the existing software and utilities available in the Graphics Lab associated with the process of generating a scene and "preparing" it to be raytraced.

When I say "prepare" I mean it in a technical as well as an aesthetic sense. This guide will cover some of the technical aspects, such as the basic syntax and structure of certain commands and files, while also including some hints, tips, tricks, etc. that I’ve come across, mostly through experimentation, and have personally found to be very helpful. As you experiment on your own, you may find some of my ideas helpful while others you may chose to ignore...there’s really no right or wrong from an artistic point of view.

B.1.1 Hardware Shading vs. Raytracing

The raytracer is capable of generating very complex images, including shadows, metallic surfaces, mirror reflections, texture mapping, bump mapping, solid texturing, and antialiasing. None of these effects can be achieved with the hardware shading facility in Jack. The shading in Jack uses the special-purpose graphics hardware of the IRIS workstations, which uses a very simplified shading algorithm. It can't produce shadows, can't shade textured surfaces, and can't do much antialiasing. The appearance of a scene when it is raytraced will differ significantly from the way it looks in Jack. The IRIS hardware can, however, shade images very quickly, so Jack can help you in setting the parameters necessary to achieve the look you want.

The raytracer described in this guide, is the “new and improved raytracer,” an updated version which is faster than ever before. The complexity of the ray casting operation is $O(\log n)$, where $n$ is the number of polygons in the scene. For example, scenes of average complexity, without too many light sources and reflective surfaces, generally take less than one hour. There is an image from the bar scene in the animation entitled, Don't Play Ball in the House, which has 9000 polygons and is highly-reflective. It previously took approximately one week to raytrace supersampled, now it takes less than four hours.

Even with the dramatic increase in speed, raytracing certain scenes will still take what may seem like forever, especially if you’re used to interactive hardware shading, but be patient. The results are worth the wait. Look in the directory $\$JACK/jacklib5/demo/rtrace for some examples of raytraced and hardware shaded images that you can display and compare.

B.1.2 Displaying Images

The raytracer generates images and stores them in files with the suffix .rle. These files are run-length encoded, to save space. These rle files may be manipulated with the commands in the Utah Raster Toolkit, developed at the University of Utah.
rle files may be displayed on the IRIS with the program display. display takes the name of an rle file and displays it on the screen.

B.1.3 Designing an Environment in Jack

The "scenes" are actually peabody environment files, which have the suffix .env. Jack is an interactive tool for designing and manipulating peabody figures. The raytracer reads peabody files, which you may either create by hand or with Jack.

If you don’t already have an environment that you want to raytrace then you have to get one. You can use Jack to:

- create objects, both primitive objects such as cubes and ground planes, and also much more complicated objects if you wish. Be creative! You will probably want to include a ground plane in your scene unless you want your objects to look as if they are floating or falling.
- read in existing objects, such as people or other neat things like a piano or a bus from existing figures and/or psurfs.
- position and move your objects around (including the joints in the figures).
- put in and position light sources, perhaps using hardware shading to see the effects of the light.
- position the camera in the desired view.
- write out your environment as a .env file

B.1.4 Lights, Camera, Action . . .

Be sure to include both a light and a camera in the scene you're planning to raytrace. Jack will write out a camera with your environment even if you don’t specify one, so you can just save the environment with the view that you like. In general, the lights you create should be positioned far from the objects in the scene.

Imagine the objects in the scene as objects on a table and try to position the light so as to illuminate everything on the table, maybe high up and off to the side, or straight up in the center. Alternatively, you might like to position the light close to the scene. This will make it very bright near the light, but it will get increasingly darker in the other parts of the scene. There might even be a strong glare on some of the objects very close to the light.

You can direct the light like a spot light by encasing part of the light in a “lamp shade,” which you can make out of cylinder with a hole in one end. This is helpful for controlling glare in the scene. Move the light into your new lampshade by positioning the “cross” marking the light's position into the center of the lampshade.

While you're positioning your light remember that the object that you see representing the light while working in Jack is just for show! In Jack, the light has the shape of a flashlight, but this is just an icon. It won't appear in the raytraced scene.

B.2 Raytracing

The first step in raytracing an environment file is to preprocess the peabody file into a form the raytracer can read. This is done with the program parsepea, which reads a .env file and produces a .bin file which is readable by rtrace, the raytracing program. rtrace generates .rle image files. The program display can be used to display these images. The programs in the Utah Raster Toolkit can be used to manipulate these images.
B.2. RAYTRACING

B.2.1 Parsepea

parsepea takes the name of an environment file and an output filename (conventionally, filename.bin) as its command line arguments. For example:

```bash
% parsepea test.env test.bin
```

If the .bin file is omitted, then parsepea writes to standard output. This is useful since rtrace can read directly from standard input, so the output of parsepea can be piped directly into rtrace.

The .bin files are meant to be a temporary means of transferring data to the raytracer. They are generally fairly large, so when you are finished raytracing, you should delete the .bin file. The important information about the scene is all described in the environment file, and another .bin file may be generated if need be, so there is no need to worry about deleting it. There is also a special option to rtrace to do this automatically.

Large, complicated scenes can take a long time to read into Jack as well as the raytracer. One way to speed things up is to make any psurfs associated with your scene binary. There is a useful utility called bps that will do this for you. The utility takes the psurf name and creates a .bps file.

One last thing: be sure to use the latest version of Jack on the environments, and not Jack3 (an older version). If your environments were designed with the old version, you may get pages and pages of warnings sent to your screen during parsepea. This will slow things down dramatically, or worse, it might not work at all. If for some reason you have a .env file previously created with Jack3 that you want to raytrace, read it into Jack and use the line numbers associated with the errors to edit the .env directly. Most of the errors and/or warnings you will receive have to do with the attributes (number of and placement of). Specifically, you will have to change the attributes associated with the polybodies and skinnybodies in the scene. To find out the correct number of attributes of a figure, read it into Jack, write it out in a test file, and then look at the file.

B.2.2 Rtrace

rtrace takes command line arguments which specify the output image file name and the image size. The first argument gives the name of the .rle file. The second argument gives the height of the image, in pixels. The third argument gives the aspect ratio of the picture, which is the ratio of the height to the width. These two parameters specify the size and shape of the image.

rtrace reads a .bin file from standard input. The input to rtrace may come from either a .bin file or directly from the output of parsepea:

```bash
% rtrace test.rle 480 1.33 < test.bin
```

or

```bash
% parsepea test.env | rtrace test.rle 480 1.33
```

Bigger pictures look better but take longer, so if you're just testing a scene set the vertical dimension to be around 150. Final pictures can be set to as much as 900, depending on the aspect ratio. Set the vertical dimension to 480 if rendering frames for an animation. The aspect ratio is usually set to 1.25, except when the frames are being rendered for an animation. In that case, set it to 1.33. Note that making the aspect ratio too big can distort your picture.

rtrace is sometimes used interactively. It can display the image as it raytraces, pixel by pixel, starting at the bottom. After it reads in the .bin file, it will prompt you to open the window with the mouse. After the rtrace is complete, the .rle file will contain the raytraced image.

In general, for small, “test” raytraces, rtrace can be run interactively. For example:

```bash
% rtrace test.rle 150 1.25 < test.bin
```

If you want to kill an interactive raytrace before it's done, use Control-C.

For larger images (maybe final pictures) you would run rtrace non-interactively, in the background:
If you want to kill a raytrace running in the background, use `ps -ef` to get the PID, and then use `kill -9`.

**B.2.3 What Are My Options?**

In addition to the command line arguments, `rtrace` takes the following options. These should appear after the arguments on the command line.

- **-noport**
  Use this to prevent the image from being displayed on your screen during the raytrace. Use this option when raytracing on one of the machines without graphics capabilities (for instance, use it on `graphics`). It will print out the current scan line.

- **-start < n >, -end < n >**
  Use these options to raytrace only a portion of the image. The `-start` option defines the starting line number; `-end` defines the ending line number. This is useful for splitting the raytracing of an image up between different machines. This can save time when raytracing large images.

  The vertical dimension command line argument is actually the number of lines in the picture. Split up the lines between machines by starting a separate raytrace on each. Be sure to name each outfile with a different name because the IRIS’s share the same disk. Specify the starting and ending lines for each machine. You may leave out the `-start` if it’s actually the first line. Similarly, you may omit the `-end` out if it happens to be the last line.

  For example, instead of saying:

  ```
  % rtrace finalpic.rle 600 1.25 -noport < finalpic.bin > junk.out
  ```

  you can issue the following three commands on separate machines:

  ```
  % rtrace test1.rle 600 1.25 -end 200 < test.bin
  % rtrace test2.rle 600 1.25 -start 200 -end 400 < test.bin
  % rtrace test3.rle 600 1.25 -start 400 < test.bin
  ```

  You don’t necessarily have to break the image into even blocks. In fact, it would be to your advantage to give faster machines bigger chunks or more difficult sections. Next, you should use the utility called `rlecomp` to put the pieces of your picture together. In the test example above, you would say:

  ```
  % composite test3.rle test2.rle > temp.rle
  % composite temp.rle test1.rle > finalpic.rle
  ```

  Also, if you’ve raytraced a picture and would like to redo just a section of it (for instance, if a scan line is missing). Use the `display` program to display the image, then press the left mouse button and the x-y coordinates of the position of the mouse cursor in the image will be printed. The y-value received when the arrow points to the missing scan line is the line in the image you want to start re-rendering from. Afterwards, the two images can be composited.

  The `rlecomp` utility can also be used for other things: for example, compositing a scene with no background onto a background (such as a scene with a person falling onto a background of the sky). See the Utah Raster Toolkit documentation for a full explanation of `rlecomp` in addition to the other utilities available.
B.2. RAYTRACING

-rmf file.bin This option removes the .bin file specified after the raytracer is done reading from it. The .bin files are only temporary; be sure to delete them as soon as they are used since they take up a lot of disk space. Don’t worry: the important information is stored in the peabody file, and the .bin file can be generated again if necessary!

The -rmf option should not be used when rtrace is reading from standard input. For example, don’t say:

```%
rtrace test.rle 600 1.25 -rmf test.bin < test.bin
```

-ss

Using the -ss option invokes stochastic supersampling, which antialiases the image, removing the “jaggies”. Using this option will remove most of the jagged edges from the picture, making the scene look a lot “cleaner”. Supersampled images take exactly 4 times longer to raytrace than non-supersampled ones.

-cull

If you know that the faces of your psurfs are all oriented consistently, you may use this option to speed up the raytracing. It may raytrace 10% to 40% faster, but if the normal of a surface is inward facing, the -cull option will cause it to disappear in the final picture! Consult the Section A.4.5 for a discussion of face orientation.

Actually, there might be an instance where you would like a surface to be missing from the scene, such as to get the effect of a “cutaway”. Your scene might take place inside of a room, and you might not be able to obtain the view that you want without finding yourself outside the room. In this case, there’s a little trick you can use. The -cull option will cause the raytracer to omit the walls of the room that were blocking your view. But realize that it may also leave out some other faces that you didn’t plan on because of the way the normals were facing.

-bgcolor < r > < g > < b >

This option lets you change the default black background to some other color of your choice. The r, g, and b arguments give the red, green, and blue components of the color. Each number is in the range 0.0 to 1.0.

Sometimes the background can be seen in the final picture because of the way it was set up. There might be some color other than black which you’d like to be showing. For example, if you have a black object, it may blend in with the background.

Another reason for using this option might be as a quick trick for making the scene look more alive (which was the original reason for this option’s implementation). My suggestion is that each scene should be encased in either a room (if appropriate) or in a huge sphere with a worldmap (mapping to spheres will be discussed later). If for some reason you didn’t include one of these encasements in your picture (or something equivalent), the reflections in your scene will probably end up looking awkward because they have nothing to reflect except that black background. Changing the background color to be grey or white or another color appropriate for the scene will make the reflections look a lot more natural. Take a look at the picture in $JACK/jacklib5/demo/rtrace/scenelraytr.rle.Z. This scene was raytraced with the bgcolor set to pink (-bgcolor .9 .05 .4) since there was no room and no sphere encasing the scene. Instead, the clouds in the scene were just mapped to a backdrop (a large plane directly behind the scene).

When choosing a background color (or finding a color for any object that you may have, for that matter) it’s easiest to use Jack. Look in the color menu and use the potentiometers to choose exactly the perfect red, green, and blue values. Those numbers can be used as the red, green, blue values for the -bgcolor option as well as being plugged directly into the .env file to change the color of objects in your scene.

-compress
This option will compress the .rle file after the raytrace is done. This conserves a lot of space especially since .rle files can be very large. Remember, the display and rtrace programs can automatically read compressed files.

- noshadow

This option will cause the shadows to be left out of a raytraced scene. Since having no shadows speeds up the raytracing a lot, you can use this option to save some time while you’re in the “testing stage” of a particular scene. You can also save some time by using this option if you happen to be raytracing a scene without a background or ground plane.

- conc

If you know you have concave faces in your scene you might want to use this option. Although, beware! This option will not work for scenes that have many, tiny concave faces. Your best bet is to either be sure while creating a scene that your faces are all convex, or to get rid of any concave faces in a scene that is already created. It will also render faster if all faces are convex. Although, if you’re in a hurry you can attempt to use this option even though it’s not recommended.

B.3 What Should I Attribute This To?

```
attribute cool.pink {
  rgb = (0.90, 0.05, 0.40);
  ambient = 0.10;
  diffuse = 0.70;
  specular = 0.20;
  glossiness = 0.0;
}
```

The notation used in the sample attribute, given above, is the notation I will be using in this documentation. In Jack each of these parameters is defined in terms of separate red, green, and blue components, representing the ambient, diffuse, and specular reflectance properties for each of the three color channels. The method Jack uses gives you a lot more power and flexibility, although, it is rather non-intuitive. Therefore, we will use the above syntax, consisting of an rgb vector and a single number representing the ambient, diffuse, and specular properties of the object.

B.3.1 Ambient, Diffuse, and Specular

The specular value can be thought of as the “percent mirrored.” It’s the “shiny and reflective” part of the scene. A specular value of 1.0 means completely mirrored, whereas a specular value of 0.0 means no reflections at all. A good specular value to use is 0.2. For metals, you should use 0.3.

The ambient value associated with an attribute can be thought of as the light on any surface as a result of bouncing, as opposed to direct light. Without any ambient, expect very dark shadows, since the surfaces in the shadows are illuminated only with ambient light. With full ambient (no specular or diffuse) the scene becomes very bright regardless of light position. For backdrops, like a sky with clouds, you wouldn’t want any shadows so the ambient should be set to 1.0 while both diffuse and specular should be 0.0.

The diffuse value can be thought of as how much of an effect direct illumination will have on a surface. A diffuse value of 1.0 will create harsh shadows, with a high response to light. Alternately, a value of 0.0 will give no response to light. A good diffuse value to use for non-mirrored surfaces with no specular component is 0.9 with an ambient of 0.1. Otherwise, in general diffuse should be set to 0.7, with a specular of 0.2 and ambient of 0.1.

Look at the example environments named ex1.env and ex2.env in $JACK/jacklib5/demo/rtrace. In the first environment, everything is full diffuse. In the second environment, the diffuse is lowered and ambient and specular are added. If you raytrace them, you will be able to get an idea of the effects of varying the ambient, specular, and diffuse values in a scene.

Note that unhappy things will happen if the ambient, diffuse, and specular values of an attribute add up to a number greater than 1.0.
B.3. WHAT SHOULD I ATTRIBUTE THIS TO?

Also, the glossiness parameter is used only by Jack and is completely ignored by the raytracer.

B.3.2 Fresnel

```plaintext
attribute shiny_silver {
    rgb = (1.00,1.00,1.00);
    ambient = 0.10;
    diffuse = 0.30;
    specular = 0.30;
    fresnel = ('silver');
}
```

It is very convenient to assign attributes using the fresnel parameter because all you have to do is specify the name of the material you want. Jack will display the fresnel color you specify and ignore the rgb term, although, the rgb must be assigned (1.00,1.00,1.00) in order for the fresnel to work correctly when the image is raytraced.

The different materials available for the fresnel parameter are:
### APPENDIX B. RAY TRACING

#### metals

- bronze
- copper
- carbon
- gold
- nickel
- silver
- steel
- tin

#### lamps

- daylight_fluorescent_lamp
- zirconium_lamp
- daylight_lamp
- coolwhite_fluorescent_lamp
- warmwhitelamp

#### rock, precious gems, and building materials

- obsidian
- ruby
- sand
- chinaclay
- lunardust
- lunarrock
- concrete
- redbrick
- asphalt
- darkloam
- sandyloam
- clayeyloam

#### miscellaneous things

- water
- breakingwave
- rust
- olivedrab
- teflon
- mirror
- vinyl

#### wood and live things

- elm
- deadleaves
- maple
- mixedwoods
- mulberry
- wildcherry
- hemlock
- ash
- grassanddeadleaves
- pine
- willow
- beachgrass
- sparselawn
- lichen
- lichen
- lawn

#### bizarre things

- protopane
- deuteranope
- tritanope

#### sunlight and outdoor things

- red_clouds
- blue_sky
- clearblue_sky
- zenith_sky
- partly_cloudy_sky
- directsunlight
- overcast_sky
- north45degree_sky
- sunairmass0
- sunplus_horizontal_sky
- sunairmass1
- sunairmass2
- sunairmass3
- sunairmass4
- sunairmass5

---

Yes, there are some strange ones. The best thing to do is to set up a simple environment, like a cube on a plane, and experiment with the different options. Of this set of colors, the metals are the most exact.

#### B.3.3 Phong Shading

You can make curved objects appear smooth with *phong shading*. This smooths out the pointy, creased surfaces created with polygons. In the attribute, add the line:

```
shademode = "phongshading";
```

and whatever has that attribute will be phongshaded. For some objects, you won’t want every single face to be phong-shaded, just particular ones. For example, when phong-shading a cylinder you would like the side faces to all be phong-shaded and the the top and bottom to just be flat-shaded. If the top and bottom were also phong-shaded, you would get a curving of the top and bottom faces toward the sides and the cylinder would lose its shape. In this case, you can modify the psurf and every face you want phong-shaded should
B.3. WHAT SHOULD I ATTRIBUTE THIS TO?

have:

[smooth]

Note that the shademode specified in the attribute declaration will always override the [smooth] option in the psurf.

Also, the raytracer will automatically phong-shade any patches you create with Jack to make them look smooth, since the raytracer approximates the patches with triangles. For a detailed explanation of patches see the appropriate secondary documents.

B.3.4 Light Attenuation

```plaintext
figure lightsource {
  light lightsource {
    psurf = "light.pss";
    attribute = attribute33;
    rgb = (1.00,1.00,1.00);
    site base->location = trans(0.00cm,300.00cm,0.00cm);
  }
}
```

In general, this is the way the light declaration will appear in your environment.

The raytracer can take another parameter in the light segment called attenuation:

- `attenuation = (const,pwr);`

The equation used for light attenuation is:

\[
\text{attenuation} = \text{intensity} \times \frac{1}{1 + \text{const} \times \text{pwr}}
\]

In real life, \(pwr = 2\), and \(const = 1\), although when these values are used for raytracing, the light decreases much too quickly. Therefore, we suggest setting \(const\) to be very low, about 0.01, and \(pwr\) to be about 0.8. See the example environment called `light_atten.env` which can be found in `rtrace/examples`.

B.3.5 Transparency

There are two more fields that can be added to an attribute declaration to describe an object's transparency, `rtransparency` and `refraction`.

- `rtransparency = scalar;`
- `rtransparency = (1.0,1.0,1.0);`

- `refraction = scalar;`
- `refraction = (0.0,0.0,0.0);`

Some sample scalar indices of refraction are:
1. water = 1.33
2. quartz = 1.46
3. glass = 1.52
4. diamond = 2.42

```plaintext
attribute sample.trans.attr1 {
  rgb = (1,1,1);
  ambient = 0.0;
  diffuse = 0.0;
  rtransparency = 1.0;
  refraction = (1.5,1.57,1.68);
}

attribute sample.trans.attr2 {
  rgb = (1,1,1);
  ambient = 0.0;
  diffuse = 0.0;
  rtransparency = (1.0,1.0,1.0);
  refraction = 1.0;
}
```

### B.4 Textures

The raytracer can also perform texture mapping. Texture mapping is the process of defining the appearance of a surface based on some other data, usually an image of some sort of texture. The texture mapped surface will appear to have the texture pasted onto it. The surface may be textured polygon by polygon, or the texture may be mapped directly onto an entire object.

There are four types of texture mapping which the raytracer uses: picture maps, normal maps, recursive attribute maps, and solid textures. The first three describe the texture on a polygon by polygon basis, using texture coordinates which must be specified in the psurf file which defines the polygons.

Solid textures are somewhat different. They make a surface seem like it is carved out of a hunk of solid material. The material may have stripes or checkers in the $x$, $y$, or $z$ directions.

Picture mapping is the simplest: it's the process of taking an image and mapping it onto a single polygon, like a picture into a picture frame. The input is an .rle image file. This file can be any image file: it could be a raytraced image, or a digitized image, such as faces, clouds, wood floor, etc. Normal mapping is the process of automatically altering the orientation of a surface to make the surface appear to have bumps. The input is also a .rle file, but the contents of the file define the parameters for the bumps, so it's not really an image you would want to display. Recursive attribute maps give you the ability to make the ambient, diffuse, specular, and fresnel parameters vary across a surface. This can be used to make object look "dirty."

Textures are also defined in the attribute declaration, with the `texture` field. The value of the expression is a list of values describing the parameters of the texture. The figure argument is a single-character string giving the type: "p" for picture texture, "n" for normal map, and "a" for recursive attribute map. The format of the remaining arguments depends on the type.

Each texture takes an integer specifying the number of repetitions. This allows you to make repeated patterns of textures using one basic image. A repetition count of 1 uses the image as is. A repetition count of 2 makes the texture consist of 4 copies of the original texture replicated in a square. A repetition count of 50 makes the texture consist of $50^2$ tiny copies of the original texture replicated in a square.

```plaintext
texture = ("face.rle","p",1);
texture = ("bumps.rle","n",50);
texture = ("tex.rle","a",1,wood,metal);
```

The first example is a picture texture, defined by the second argument "p". The image for the picture is contained in the file `face.rle`. The third parameter is an integer giving the number of repetitions. Note
that in the case of picture maps, if an rgb is also specified in the attribute declaration, then that color will
be added into all the colors of the picture map. For instance, a blue and white picture map, with an rgb
of red, will give you a picture map of purple and pink. This can be very useful, but if you'd like to use the
picture as it is, be sure to set the rgb to (1.00,1.00,1.00).

The second example is a normal map, defined by the second argument "n". The image for the picture is
contained in the file bump.rle. The third parameter is an integer giving the number of repetitions. In this,
faces which have this texture will appear to have 50 rows and 50 columns of little bumps.

The last example is a recursive texture which interpolates between the attributes wood and metal, using
tax.rle as an interpolation file.

B.4.1 Specifying the Texture Coordinates in the Psurf

In addition to adding one of the texture definitions to an attribute declaration, you must also define the
texture coordinates for the faces in the psurf file. The texture coordinates define how the texture will be
mapped onto the face.

The format of the texture coordinate information is:

\[
\text{atexture 2 (0.0,0.0) (1.0,1.0)}
\]

The easiest way to orient the picture is to bring the psurf into Jack. Choose the info-menu and look at
the nodes. From the node that you specify, to the next node, will define the bottom of the image. Therefore,
if you were mapping a picture to a plane, you would want to find the name of the node in the bottom-left
corner of the plane because the bottom-left corner to the next node (the bottom-right corner) would define
the bottom of your image. This is important for picture maps since the orientation of the picture usually
matters. Once you find the node number that you want, you find it in the face list but remember, the
textures take an index into the face list. So in our example texture above, if we wanted to put a picture
map on the face of the psurf, and we found using Jack that the name of the node in the left-hand corner of the
face was node 11 then we would say:

\[
8 \text{ 11 12 9 [atexture 2 (0.0,0.0) (1.0,1.0)}
\]

because 2 would be the index into the face list that would correspond to node 11.

If we wanted the mirror-image of the picture, we would switch the beginning and ending u-coordinates,
and if we wanted to flip the picture upside down, we would switch the beginning and ending v-coordinates.
If the picture is too stretched, you can adjust it by decreasing the beginning and ending u or the starting
and ending v, depending on which dimension is stretched. The values must stay between 0.0 and 1.0.

B.4.2 Stextures

Stextures, short for "solid textures", will look continuous across the faces of the object. Stextures are nice
because they don't use texture coordinates and you can map to anything you want. In the attribute you
add the following definition:

\[
stexture = ("type", \text{attribute1, attribute2, size});
\]

The different types are: "xstripes", "ystripes", "zstripes", or "checkers". The stripes are just different orientations around the axes. The attributes are any attributes that you want for the stripes or checkers. The size is either the size of 2 checkers or the size of 2 stripes. This is useful because in segments of different sizes, you can have the same size stripes or checkers. The stripes and checkers are both defined according to the local coordinates of the object so that when the segment moves (for instance, in an animation) the stexture moves with it.

B.4.3 Normal Mapping

The normal mapping is now generalized and improved to consider the given map as a height field on the
surface. In other words, any rle file can be interpreted as a normal map. The syntax is also augmented for
textures in general to allow scaling of the maps by adding a last argument which is a floating point scale factor.

Instead of:

\[
\text{texture} = (\text{"file.rle"},\text{"n"},1);
\]

You can say:

\[
\text{texture} = (\text{"file.rle"},\text{"n"},1,50.0);
\]

The scaling factor makes the map much more intense. This is particularly useful with the new normal maps. For an example of this mapping see \texttt{rtrace/examples/norm.map.env}.

\section*{B.4.4 ShrinkWrapping}

When we shrinkwrap to an object the raytracer constructs an encasing sphere for the object and gives the effect of collapsing the sphere around it. Actually what you will see is the texture that you’ve specified wrapped around an object (like aluminum foil around a bowl). Shrinkwrapping is a way of doing texture mapping without having to specify the texture coordinates explicitly in the psurf. Shrinkwrapping can be used for everything except spheres.

The shrinkwrap is defined with a \texttt{wrap} statement, which gives a homogeneous transformation defining the position and orientation of the encasing sphere. In addition to the \texttt{wrap} assignment statement, the attribute should have a texture. The attribute’s texture will then be shrunk around the object with the given transformation. For example,

\[
\begin{align*}
\text{texture} &= (\text{"texture.rle"},\text{"p"},1); \\
\text{wrap} &= \text{xyz}(90\text{deg},60\text{deg},30\text{deg}) * \text{trans}(127.35,401.78,23.65);
\end{align*}
\]

The \texttt{xyz} and \texttt{trans} values associated with the \texttt{wrap} specify the orientation of the \texttt{wrap}. The \texttt{xyz} is a rotation and the \texttt{trans} is a translation.

Because you can shrinkwrap to an arbitrary object, you can shrinkwrap picture maps of people’s faces to either contourbodies or polybodies. You can orient the maps on the faces using rotation and translation. The \texttt{display} program for displaying an image also lets you check the color value of pixels in the image. This option can be used to match the skin color of the face map to the skin of the figure body. Face mapping is an interesting concept and the faces look really neat so if you find some extra time, just experiment. There is an example environment called \texttt{face.map.env} in \texttt{rtrace/examples} which you can look at. Ask appropriate lab members for any more information.

There is also an example environment containing all of the aforementioned types of texture mapping called \texttt{ex3.env} in the same directory.

\section*{B.4.5 Face Mapping}

We have developed and tested a simple technique for mapping scanned 2D images of human faces to 3D laser-scanned human heads. From this we can gain more realistic images, as well as a potential approach to facial animation of real human faces within a computer graphics environment.

\subsection*{B.4.5.1 The Tools}

The human body data, including the data for the head, was obtained through laser-scanning actual human figures. The 2D face maps were scanned from video images of real human faces. The final face mapped body images are raytraced. The raytracer takes a face map for the front of the head and a second map to cover the back of the head and can produce, through shrinkwrapping, convincing human facial features on a head from all viewpoints.
B.5. ALGEBRAIC SURFACES

B.4.5.2 Preparing A Face Map

After the face maps are scanned they are color corrected. Making the colors look more alive makes the face maps seem more realistic and makes the faces blend in better with their surrounding computer graphics environment.

The second map for the back of the head is usually just a solid color map which is matched with the color of the hairline to give the effect of hair. Another alternative is to omit the second map and add the hair portion to the head as a 3D object. We use the latter method since our emphasis is on the face and not the hair.

B.4.5.3 Positioning

The important characteristics of the head, for instance the nose, chin, eye sockets and forehead are aligned with the map through specifications of a rotation and a translation to the raytracer. This is actually a convenient method to use since no texture coordinates are specified by the user. All the positioning as of now is done manually, although there is work being done to create code to position the maps automatically.

It should be noted that this approach does not preserve the original face appearance. Our goal is only to achieve convincing human faces, not necessarily to represent real people in a computer graphics environment.

B.4.5.4 The Syntax For Rtrace

```c
/* attribute for face map */
attribute faceattr {
    rgb = (1.0,1.0,1.0);
    wrap = xyz(0.00deg,0.00deg,82.00deg) * trans(0.0cm,0.0cm,1.9cm);
    texture = ("face.rle","p",1);
    shademode = "phongshading";
}

/* attribute for head map - both face and hair */
attribute faceandhairattr {
    rgb = (1.0,1.0,1.0);
    shademode = "phongshading";
    wrap = xyz(0.00deg,0.00deg,89.00deg) * trans(0.00cm,0.00cm,1.90cm);
    texture = ("face.rle","1",1);
    texture = ("black-hair.rle","2",1);
}
```

B.5 Algebraic Surfaces

The raytracer is particularly good at generating images of spheres, so it has a special technique for representing spheres without using a psurf. The sphere definitions for rtrace go inside the segment:

```
psurf = {'sphere.pss'} * scale(50.0,50.0,50.0);
sphere = 50;
```

The above psurf definition is for a unit sphere.

The psurf specified is ignored by the raytracer, but Jack uses it and will display what’s there so you might as well specify a sphere. If you have lots of spheres in your scene, then you might want to use a very simple psurf that generally approximates the volume set up by the sphere — like a cube with its center defined at its center as opposed to one of its corners. The number following the sphere declaration is the radius which is what is used by the raytracer.

A sphere with a texture map around it encasing a scene is called a world map. A worldmap will give you much more natural reflections as opposed to a black background or even a backdrop. Typically the texture used for the worldmap is a cloud picture map which can be found in rtrace/textures/cloud.rle
For instance, you would say something like the following in the attribute declaration:

```
texture = ("cloud.rle", "p", 1);
```

The clouds will be mirrored about the equator of the sphere so you might want to line up the groundplane of your scene with the equator of the sphere so that you don’t see the mirroring. When texture mapping to spheres, the most distortion occurs around the equator and the least around the poles. Therefore, if you’re not satisfied with the way the map looks, try rotating the whole sphere.

Remember, you can use stextures as well as textures with spheres.

**Peabody** has been extended to accept algebraic definitions of cylinders and disks as well as spheres. The notation has been generalized from the `sphere = radius` notation to `asurf` declarations within the segment. **Jack** still uses the `psurf` declaration to approximate the object for wireframe or hardware shaded display, but **rtrace** uses the `asurf` declaration.

The syntax for a cylinder is:

```
asurf = ('cylinder', radius, height, rendertype);
```

where `rendertype` specifies the way the cylinder is texture mapped (if it’s attribute has a texture map), and is 0 for a mirrored texture (the texture is mirrored about each hemi-cylinder) or 1 for a complete wrap with a seam. Both `rendertypes` have even distortion, but type 1 has a single line of discontinuity.

The syntax for a disk is:

```
asurf = ('disk', radius);
```

**Peabody** still accepts the `sphere = radius` sphere notation for compatibility, but new code should use the `asurf` syntax which is:

```
asurf = ('sphere', radius);
```
Appendix C

Radiosity

Min-Zhi Shao

C.1 Introduction

Basically, radiosity global illumination solution consists of three stages: environment modeling, radiosity calculating, and environment displaying. The major functions of our current radiosity system include:

- interactive and automatic meshing in environment modeling;
- fast over-relaxation progressive refinement algorithm;
- accurate analytical form-factor determination;
- adaptive patch-element subdivision in radiosity calculation;
- texture mapping;
- output decomposition for walking through complex environments.

C.2 Environment Modeling

We use Jack to build a radiosity environment. There is, however, one special requirement: you need to write every psurf (.pss) in your environment to your current working directory. That is it. After finally writing your environment to a peabody file (.env), what you need to do next is to mesh the environment to finer grids and then to convert it to a radiosity environment description file (.rad). There are two ways to do this:

1. You may mesh each psurf separately by using program meshpsurf. The format is:

```
meshpsurf file.pss
```

[input: .pss file; output: .pss file]

Press the middle button of the mouse to enlarge the window. You can begin to mesh a psurf by pressing the Ctrl key. The program will go through the psurf face by face and for each face it will ask you for a mesh density (\(u, v\) for rectangle and recursive depth \(w\) for triangle). At any time, you can change your view by using the mouse after pressing the Esc key and then go back to the mesh mode by pressing the Ctrl key. The output is also in psurf format and it will overwrite the original psurf file. If you are not satisfied with what you have meshed, you can always type “n” to start the whole mesh process again or type “q” to quit. Otherwise, simply type “y” to output the desired mesh. It should be noted that the current program meshpsurf cannot process a psurf which has a face with more than four vertices. If that is the case, you should use the uniform mesher. We will discuss it later.
Having meshed the psurfs, you can convert your peabody file to a radiosity environment description file by using the program envtorad. The format is:

```
envtorad -l lightmult -e file.env -r file.rad
```

[input: .env file; output: .rad file]

Here `lightmult` is an integer multiplier for converting the Jack light values, normally in domain [0,1], into the range of radiosity light sources. The default value of `lightmult` is 10000.

(2) You may also use the program `envtorad` with `-m` option to mesh the whole environment uniformly avoiding the tedious psurf by psurf mesh. The format is:

```
envtorad -m meshsize -l lightmult -e file.env -r file.rad
```

[input: .env file; output: .rad file]

Here `meshsize` is an integer for measuring the fineness of mesh. We determine the maximum allowable length of any edge in the environment by dividing the diameter of the entire environment with `meshsize`. The default value of `meshsize` is 20. This program enables you to get a quick, though even mesh.

Please make sure that your environment is an enclosure and all patch normals are pointing into the environment. You can reverse the psurf normals with Jack command `reverse segment faces`. You can set psurf reflectance with Jack command `set attribute color` and set psurf emmitance with Jack command `set light color`.

In radiosity, any object has to be closed and solid. It also applies to light source. Unfortunately, the Jack standard light source `light.pss` is not a closed solid object. The solution is simple: create a closed solid light source psurf and replace it for `light.pss` in your peabody file. You can also edit the emmitance column of radiosity environment description file to specify any patch you like to be a light source.

### C.3 Radiosity Calculating

Our major iterative solution scheme is a fast over-relaxation progressive refinement algorithm. We have also integrated the analytical form-factors and the adaptive patch and element subdivisions into our radiosity calculating program `radiosity`. The format is:

```
radiosity -h hemicube -d meshdepth -r file.rad
```

[input: .rad file; output: .env and .pss files]

Here `hemicube` is an integer for hemi-cube resolution with a default value of 200; `meshdepth` is an integer for recursive adaptive subdivision a default value of 3. When the program is running, you can type “s” to shoot the patch radiosity, “o” to output the current iterative solution, or “q” to quit the program.

If the program is killed by the system, it would be most probably caused by a main memory overflow. In our experience, environments with less than 50,000 elements so far have no problem with most workstations in the lab.


### C.4 Environment Displaying

We use Jack to display the radiosity solution which is in peabody format. You may find the following Jack commands to be useful: turn background off, turn projections off, environment backface off, environment textures on, make environment smooth, make everything shaded, change view.

The program, a sample environment, and some demos are in: $JACK/jacklib5/demo/radiosity. Please send your question, bug report, comment, and suggestion to: jack@graphics.cis.upenn.edu.

Let There Be Light!
Appendix D

SASS

Anthropometric Spreadsheet and Database

Francisco E. Azuola,
Sussana Wei,
Teo Kok Hoon

D.1 Abstract

This is the user’s manual of SASS v.2.2.1, the Spreadsheet Anthropometric Scaling System (version 2.2.1). It describes the usage of SASS, a spreadsheet-like system which allows flexible interactive access to all anthropometric variables needed to size a computer-based human figure, described structurally by a Peabody file.

Data that may be accessed is organized into the following “groups”: segment dimension ("girth"), joint limits, center of mass, and strength, all of which work based on statistical population data. SASS creates generic computer-based human figures based on this data.

SASS also is an anthropometric database and interactive query system that works upon anthropometric data of real individuals. Scaled computer-based human figures created by SASS can be displayed directly, and interactively changed, within the Jack software.

D.2 Introduction

D.2.1 Using This Manual

Welcome to SASS v.2.2.1, the anthropometric spreadsheet and database. If you are already familiar with the program from an earlier version, you should proceed to try the tutorial in section D.3, which includes all the main functions now available in the program. You should then consult D.4 if you need further information on a particular topic (check the table of contents at the end of this manual).

The rest of this section (D.2) introduces you to the program, tells you how to install the program, and about the notational conventions used throughout this document.

Section D.3 is a quick introduction to SASS, arranged somewhat like a tutorial. If you wish to use the program casually, this section will probably contain all the information you need. The final section of the tutorial is very different, and is a sort of advanced tutorial for people who do not like tutorials.

Section D.4 is a complete reference guide to the features of SASS. It is not a pure reference guide, as some extra examples and explanation appear with the more complicated commands. The table of contents and appendix may help you to find things. This reference manual is intended to be complete, and all the features of SASS should be documented.

Section D.5 is a collection of “tricks” that have proven to be useful for SASS users.
Section D.7 contains a table of hot-key mappings for all the keyboard keys. Also, a summary table of pull-down menu functions is presented. Also in there you can find the format of the database files. Finally, there is a topic-oriented table of contents at the very end.

**D.2.2 For Experienced Users**

If you know what a mouse is and how to work with a spreadsheet, section D.2.3 tells you how to install the program, then try reading section D.2.4, run SASS, and try to figure out how it works.

There are a few non-obvious, powerful commands, and there is a final section of the tutorial (see D.3.5) that is written for people who do not find reading manuals very amusing (rather be lying on the beach and having fun!). Give it a try, and you'll probably be well on your way to becoming a SASS expert.

Also note that at the very end of section D.7 there is a special table of contents that is organized by command function for easy access to the reference manual.

SASS (Spreadsheet Anthropometry Scaling System) is a very fast mouse-based anthropometric spreadsheet and database to run on Silicon Graphics computers. It is most useful for scaling peabody figure definitions for Jack.

The main features of SASS v.2.2.1 include:

- Easy to learn.
- Power tool for power users.
- On-line help facility.
- More hot keys for fast access.
- Mouse-based improved interface.
- New Polybody psurf geometry.
- Up to date with Peabody human figure definition in Jack.
- Fully re-definable front and back end configurations.
- New flexible interface with geometry.
- Hierarchical figure definition.
- Smart body scaling.
- New improved anthropometric database facility.
- New interactive connection with Jack.
- Army and Air Force population models and NASA individual-based model available.

SASS was originally developed with one idea in mind, i.e., generating the dimensions of each segment of a human figure based upon population data supplied as input. The human model used by the current version of SASS consists of thirty one segments (body structures), of which twenty four have a geometrical representation. For each of those twenty four segments, there are three dimensions which are required, namely, length, width, and thickness. This means that at least these seventy-two measurements should be available. SASS is distributed with default data, for your convenience.

The psurf (polygon surface) geometry of each segment must be scaled by real measurements for a person or percentile measurements for some specifiable population. SASS generates figure files with the appropriate description of the segment dimensions and joint limits, so that Jack can display the resulting figure.

SASS uses population statistic data to create generic human figures. Alternately, SASS has a built-in anthropometric database that stores anthropometric data for (real) individuals and provides an interactive query system for database access.

SASS allows flexible interactive access to all variables needed to size a human figure described structurally by a Peabody body file. SASS screens, as shown in Fig. D.1 and more diagrammatically in Fig. D.2, are
divided into different sections including anthropometric group selection, global data, command menu, local data.

Data that may be accessed is organized into anthropometric "groups". The current version handles four groups: segment girth, joint limits, segment center of mass, strength.

The global data section of the spreadsheet is intended to allow a "whole body" view of the current figure parameters. Currently the items considered are: population, sex, figure type, mass, stature, and overall percentile of any human figure. It is important to realize that since SASS is a relational spreadsheet, modifying any data in this section will affect the values of the individual segments. For example, changing the figure’s percentile will cause the data to be scaled in other appropriate segments contributing to stature.

The data section is used for the display of individual segment data and their corresponding percentiles. The leftmost column is reserved for the segment names, while the other six columns are used for the data and percentile display. The segment name column cannot be modified, but the figure definition file can be customized if needed (more on this later). The data is read in from a selected population input file.

Data and its corresponding percentile is modified by simply moving the locator device to the desired cell and pressing on a button. Changing any segment percentile will change its corresponding dimension. SASS keeps a current measurement unit type for each group (e.g., in, cm, deg, rad). Unit conversion is performed when necessary. All SASS functions are accessed through pop-up menus, direct spreadsheet cell manipulations, or through the hot keys.

In the case of database queries, you request the desired information through pop-up menus and SASS
translates it into the actual query recognized by the database system. The query results are displayed in spreadsheet format.

**SASS** is written entirely in C/C++, and makes use of UNIX system calls and calls to the Silicon Graphics graphics library (the GL).

To invoke **SASS** from the shell, you simply type `sass`. In this document, however, we refer to it as **SASS** (i.e., uppercase). **SASS** v.2.2.1 was written by Francisco Azuola. The original version of the program is documented in [Grosso88]. This technical report, even though superseded in many aspects, is still a valuable resource regarding **SASS**'s philosophy. The database facility was designed by Francisco, and written by Teo Kok Hoon. It has been updated by Francisco for the current version. The strength sheet was designed by Francisco and Sussana Wei.

All the bugs are due to Francisco. Please send any bug reports or suggestions to jack@graphics.cis.upenn.edu.

### D.2.3 Installing the Program

**SASS** program is distributed in the form of an executable file, and several data directories. If you installed Jack from a distribution tape, it will be installed already.

The executable file is: `sass`.

To execute **SASS**, you need to type `sass` at the UNIX prompt. The executable has been compiled to be used under IRIX 4.0 and above. The current version of the program is 2.2.1.

The executable, `sass`, is put in your path, and the data usually is in `$JACK/jacklib5/sass/`.

You must have the following environment variable in your `.cshrc` file:

```
JACKSASSLIB = /pkg/jack/jacklib5/sass/
```

pointing to where all the data files are, i.e., files with extensions:

```
(xx.dat, xx.fig, xx.db, xx.lbc, xx.ntc, xxffc).
```

The **SASS** directories are:

- **data/**: Contains default data files used by **SASS**. In principle, all these files could be modified, but we recommend not to do so unless you know what you are doing.

  In this directory, there are four main groups of files:

  - *.fig files
  - *.db files
  - *.dat files
  - *.ffc files

  The *.fig files are `pconthum.fig` and `cconthum.fig`, and they are used as control figure files. These two files are parsed by **SASS** to know the details of the output file it is expected to produce. In a sense, one can think of **SASS** as a compiler of Jack's figure description files. As output, another figure file is generated with the same structure as `pconthum.fig` (if polybody figure is created) or `cconthum.fig` (if contour figure is created), but containing the appropriate values.

  The *.db files are files associated with the database. These files contain data from real people, gathered by NASA [Pandya 92a, 92b].

  The *.dat files are divided into 5 categories:

  - girth.dat, jnltmt.dat, cmass.dat, napgth.dat. The first 3 files correspond to the spreadsheets' default values. The files girth.dat, and napgth.dat, (and ngfgth.dat which is a copy of girth.dat), correspond all to girth information. 1 jnltmt.dat and cmass.dat contain the joint limits data and the center of mass data 2.

---

1 Girth Data corresponds to Army 1988 Anthropometry Survey [Natick 88]

2 This data has been taken from NASA-STD-3000, Man-Systems Integration Standards [NASA 87]
D.3. A SHORT SASS TUTORIAL

- *trans.dat* and *scale.dat* are files used by SASS to scale the figure files it produces. They have been produced by applying SASS to the control *.fig files above.

- xref.dat is a cross reference file containing a listing of the joints in the control figure files and the corresponding joint limits values available in SASS, if any. _void_ is used to signal that SASS doesn’t support a particular joint, and thus, SASS only copies the joint values verbatim from the control figure files.

- _s.fig_def_*.dat are files containing a figure description file to be parsed by SASS at the beginning of any session. These files contain the necessary information for SASS to create the data structures it needs. These files are given to you as default, but can be changed if necessary. However, if you decide, to do so, make sure you understand the mechanics of SASS internal operation.

- poly_fact*.dat and cont_fact*.dat are files containing the definition of SASS’s geometric interface.

The *.ffc files are files associated with the strength sheet. These files have been provided by NASA. Programs by NASA for manipulating the aforementioned data were adapted into SASS.

D.2.4 Notational Conventions

You can do most of your work using just the mouse, but your working efficiency will improve dramatically if you use hot-key combinations for some of the more common commands, once you know your way around. The SASS hot-key commands are produced by holding down specific keyboard keys.

When <Shift> is required as part of a command specification as in <<?>, you must hold that key down, i.e., <Shift> <<?>. The key marked <Enter> on the keyboard allows to complete a data input.

Clicking a mouse button refers to a quick press and release of a mouse button without moving the mouse. Typically, the left mouse button is used to select commands or to select a specific sheet cell for input. The right mouse button is used with pull down menus.

In this document, file names, commands that might be typed to the UNIX shell, used to illustrate some SASS features are displayed in a typewriter-like font (like this).

Certain commands in SASS can be invoked in different ways. When this is the case, a command name is often used rather than the specific key to indicate that any method for invoking that command yields the same result. Command names appear in a bold version of the standard Roman font, and are typically capitalized, for example: **Create Fig**.

A particular key is denoted within < -> (e.g., <m>). When a mouse button that must be clicked is involved, it is denoted within [.] (e.g., [OK]).

D.3 A Short SASS Tutorial

The tutorial assumes that this is the first time you are using SASS.

D.3.1 Starting the Program

To run SASS on one of the Silicon Graphics Iris 4D workstations, type **sass** after the shell prompt:

```
% sass
```

An alternative way for running SASS from within Jack, is discussed in section (D.4.3), but for now you should not be concerned about it.

It takes a few seconds for the program to initialize. if you’ve never used a mouse-based spreadsheet before, skip to the next paragraph; otherwise, you can probably figure out a lot right now by just playing around. After you’re comfortable, try out the last section of D.3 — it’s an advanced tutorial for people who normally hate tutorials. The left mouse button does almost everything. It does selection, and runs the scroll bar. The right mouse button is used to select items from the pull down menus. Hot keys are gotten by holding

---

3 All strength information is according to [Pandya 92a, 92b]
specific character keys. If things go pretty well, you may want to skip to the last section of this tutorial or scan the section headings in the table of contents to figure out what sorts of things SASS can do.

After you typed `sass` and pressed `<Enter>`, parsing of the scripts (front end figure definition files) and reading of population data will take place and a grid or frame will then appear on your screen. Moving the mouse, center the grid to the desired position on the screen, and then press any mouse button. After a few seconds, the anthropometric spreadsheet will appear on screen. The “hourglass” cursor will change shape into an arrow when the initialization process is over. If you missed it, tough.

The spreadsheet is filled with numbers, corresponding to the default data. See section (D.2.3).

Dialog Line. At the bottom of the spreadsheet is the Dialog Line which is used for displaying any messages about what SASS is doing and in certain cases to input data from the user. The first message that should appear in this line is:

```
Press LEFT Mouse Button to select items.
```

Different operations and commands on the data in the spreadsheet may now be performed.

Selecting cells. If you need to select a cell, you must use the left mouse button. Use the mouse to move the cursor to the cell of interest, and press the left mouse button down. Notice how the selected cell is highlighted, even before you release the mouse. The highlighting before the cursor release indicates what will be selected if you release the mouse button at that point (the highlight color is black).

Selecting a cell does nothing to it. If you make a mistake and select the wrong cell, or change your mind, just press `<Enter>` key or click the [OK] button.

Using the Dialog Box. If you want to experiment a bit, try clicking and changing the cell values. Using the Dialog Box is simple. To move the cursor, use the arrow keys, or point with the mouse. To erase (part of) the current number drag the mouse on top of it with the left button pressed. This should highlight the number. Press the <Delete> or the <Backspace> on your keyboard. An easier way is to press <ESC>. This will completely clear up the Dialog Box entry.

Spreadsheet types. There are two types of spreadsheets. One is the Anthropometric Spreadsheet which is used to display the anthropometric data for a population based person in different anthropometric groups. This spreadsheet allows you to browse and modify any anthropometric data. The other type of spreadsheet is the Database Query Spreadsheet. This other spreadsheet is used to build queries for the anthropometric database and to display the results of the queries. This database works with data from particular individuals. Each of them is described in the reference section.

D.3.2 Scrolling

Since the amount of items on the screen usually is longer than the SASS window size you can scroll through the rest of the items using the left mouse button in the scroll bar to the right upper corner of the sheet. In the database, use `<p>` and `<n>` to move across sheets horizontally, and use the funny looking buttons (concentric rectangles) in the upper right corner to move between pages vertically. For the strength spreadsheet the upper funny looking button to move between pages.

D.3.3 Creating output files

By now, you’ve probably trashed the spreadsheet cells pretty well, but pretend for a minute that the modifications you’ve made were useful, and that you’d like to use this information to create a figure file, or a scaling file, to be used on an existing figure.

Press `<X>`. The dialog line should show the PORT OFF message. If it reads PORT ON, press `<X>` again. Select Create Fig entry, in the command pad (upper right corner). If you use this command button, you will be asked to choose a figure type. Click [Polybody]. Then you will be asked for the type of file you want, that is, a [Scale Format] or a [Figure Format] file. Try [Figure Format]. Now the File Dialog Window should appear on the screen. You must provide a “valid” name for the file. This file will be created and saved in the directory under which SASS was invoked, or you can use the dialog box to indicate a new directory. The process takes some time. Progress can be observed by watching the “hourglass” cursor. You can use Jack to display the figure you’ve created.
D.3.4 Saving the spreadsheet contents and Quitting

After a joyful time working with SASS, you are ready to quit the program. But what about all those nice numbers you have stuffed in? Well, you probably don't care too much about them. After all, you were just "playing" a bit with the program. However, pretend again that the modifications you've made were useful.

In order to save the spreadsheet contents you would type <S> or select the menu item [Save sheet] (click the lower funny looking button in the upper right hand corner of the sheet to reach the menu).

However, since right now you want to exit the program, press [Quit]. Click [Yes]. Then click [Yes] again to save the spreadsheet contents in a file. Now you must provide a valid name. You need not to worry about the "sas" extension, which is appended automatically by the program to the name you provide.

D.3.5 Advanced Tutorial

If you pretty much know what you're doing, try out some of the commands in this section. Just do what it says, and you'll be introduced to some of the powerful things that SASS v.2.2.1 can do. Play around, and you'll probably understand the commands well enough to use them yourself. If not, complete details are available later on in this reference manual. For your convenience, the appropriate reference section appears following each entry in parentheses. Also notice the margin titles telling you the purpose of each exercise.

Here are some exercises you might want to try:

1. (D.4.1.12) Press <?> (do not forget that this is a shifted key) to activate the help sheet. Press any key to go back to the main sheet. Press <E> (don't forget shift key). This will bring up the help menu. This is a more extensive help. Select one of the topics in the pull-down menu by dragging the mouse while pressing its right button and releasing it when you are on top of the specific item. A rubber band window will appear on screen (basically its frame). You must press the right mouse button and drag the mouse simultaneously to make it big enough for you to read its contents.

2. (D.4.1.3) Select any cell in the middle sheet, i.e., a value cell. This is done by dragging the mouse with the left button pressed to the particular cell, and then releasing it. The cell will be highlighted. The Dialog Box should be on screen. Enter any number and press [OK]. Observe the changes.

3. (D.4.1.2) Press the <f> key or click in the [Male] cell in the header of the sheet. This will switch the gender. Press then <n> or click the [Female] cell to go back to the Male case.

4. (D.4.3.1) To change body model, press the <n> key or click the [Poly body] cell in the header of the sheet. This will switch the figure model. Press then <p> or click the [Contour body] cell to go back to the polybody case.

5. (D.4.1.1) You are currently in the Girth group. To activate other groups, i.e., Joint Limits, Center of Mass, or Strength, press <j> or click the [Joint Limits] cell (in the top line of the sheet). This will activate the Joint Limits group sheet. Press <c> or click [Center of Mass]. This will activate the Center of Mass group sheet. Press <t> or click [Strength] to proceed to the Strength group sheet. Press <g> or click the [Girth] cell to go back to the Girth group sheet.

6. (D.4.1.10) First, make sure you are in the Girth group spreadsheet. Pressing <g> will take you there. The scroll bar will appear in the upper right corner of the sheet (gray color). Use the mouse to select this bar (use the left mouse button) and drag it up. Release at some point the mouse button. The sheet will now show more segments from the list of segments, hiding some others from the beginning of the list. Click the left mouse button on the scroll bar to go back to normal display.

7. (D.4.1.13) Press <o> or the top "funny looking" button in the right corner of the sheet to activate the segment - object selection display. In the later case, press [Object]. Try again, but now press <s> or [segment]. Segments and objects are closely related, as the objects (or body parts) are composed of segments.

8. (D.4.1.5) Press lower "funny looking" (concentric rectangles) button in the right corner of the sheet to activate the pull-down menu. When the menu shows on screen, do not forget to use the right mouse...
button to select an item. Try selecting the item “Male” from the “Figure” drawer of the menu (leftmost
drawer).

9. (D.4.1.2) Try modifying the group percentile cell. By default its value is 50th percentile. To do so click
the [Group Percentile] cell in the sheet header, and input another value.

Units.

10. (D.4.1.4) Change units by clicking the [cm -> in] button on the command pad of the sheet. Go back
to cm by clicking [in -> cm].

Query Sheet.

11. (D.4.4) We will use the query sheet to find the following subsets of individuals:

- Those who are heavier than 65kg and have zero fatigue level
  - Click on the Query DB button to begin. Wait a few seconds. This is necessary only the first
time you access the data base under a given session. Click on the Query DB button again.
  Click the right mouse button to bring up the Query Menu and select Global Information.
  Use the right mouse button again to select Mass and again to select the > operator. When
the “Type in value:” prompt appears in the dialog line, type 65 <enter>. Click on right
mouse button now for the Conjunction Menu, select the and operator. Repeat the process
of clicking on the right mouse button to select Global Information, fatigue, Level 0 and
finally Done. You will see the query string “mass > 65 and fatigue = 0” incrementally form
as the query is being constructed. At any point of time, you can select the Previous Menu
item to redo a selection. Note that the Abort command is available only in the Query
Menu and the Conjunction Menu.

- The list of people found in the previous query (“those who are heavier than 65kg and ...”)
  who have a percentage of lean mass greater than 80
  - Assuming that you have successfully executed the previous query, click on the [Extend
  Qry] button in the command box. Now, right mouse button would bring up the Conjunction
  Menu. Continue to build the query by selecting and followed by Global Information, %
  Lean Mass, >, type 80 <enter> and select Done to execute.

- Those people found in the second most recent query
  - This can be done easily by invoking the query history dialog by clicking on the Qry History
  button. A dialog box containing a scroll-able list of the 50 most recent executed queries pops
  up. The first row refers to the current query, the second refers to the previous query and so
  on, click on the third row followed by the [Ok!] button.

- The bottom 5 people of the currently displayed list.
  - Click on the [Select] button, all names will be highlighted. Now click on the middle mouse
  button, all names will be un-highlighted. Click on the bottom 5 individuals using the left
  mouse button to bring back the highlight. When ready, click on the right mouse button to
  commit the selection. All individuals whose names are not highlighted will be discarded from
  the current list.

- People taller than 175 (cm) and who has shoulder flexion strength greater than 30 (ft-lbs) and
  knee extension strength greater than 40 (ft-lbs)
  - Begin this query by clicking on the Query DB command once again. Select in order, Global
  Information, stature, >, 175 <enter>, And, Anthropometric Groups, strength,
  shoulder, type in the required shoulder velocity and angle, select flexion, >, 30 <enter>,
  and, Anthropometric Groups, strength, knee, type in the required knee velocity, angle,
  continue to select extension, >, 40 <enter>, Done.

- Notice that in our previous example, there are more attributes than can be displayed. The
rest of the attributes can be seen by using the horizontal scrolling keys, namely <n> and
<p>. <n> scrolls to the next horizontal section if there is any while <p> scrolls to the one
before. The Project button can be used to filter out unimportant attributes. Click on the
Project button now to highlight all the currently displayed attribute labels. Remove the C1,
12. (D.4.2) Press <t> or click on [Strength] with the mouse from the group menu line (at the top of the sheet). The strength sheet is a bit different from the other spreadsheets. The values you see on screen correspond to angles in the diagonal, and to strength values in the last column. If you modify an angle value, in the diagonal, the corresponding strength is computed. Try modifying any of the angles. Notice that the angle you input could be clipped if its value is beyond the particular joint angle limits (the joint limits used are those found in the joint limits sheet). Try changing pages, by pressing the upper funny looking button in the top right of the sheet. Also, try changing the strength mode (currently possible only in the first page of the strength sheet). There are two possible modes, predict and individual. Try modifying the motion speed field. To do so, click in the “motion speed” value cell, and enter a number at the dialog box. The range of speed goes from 60 to 240 deg/s, by default. Values outside those limits will be clipped. Finally, try modifying the lean body factor field (10.4% by default). To do so click in the “lb factor” value cell, and enter a new value.

13. (D.4.1.4) This is population based figure creation. Before going any further press <X>. The dialog line should now read PORT OFF. If it reads PORT ON, press <X> again. Click on the Create Figs button. You need not to supply a filename for each of the individuals you see listed, as the program will automatically provide names for you, unless you want to give the figures special names. At any point of time, click on Cancel to discontinue the operation.

14. (D.4.3) This exercise will enable you to do interactive use of SASS with Jack. You must go into Jack and set up an environment containing (at least) a human figure (polybody). Basically, load into Jack the file “test.fig” that you created in the last exercise. Then in Jack you must select the Receive Data from SASS command (in human-anthropometry sub-menu). This will make Jack enter a wait mode. Then go to SASS. Change the group percentile (see above) to 75% and press <w>. A figure name is requested. If the environment in Jack contains only one figure, press <OK> (this is our case). Otherwise specify the name of the figure to be scaled (in our case the name would be “test”). You will also be asked for the type of figure you want, namely, polybody or contour. Choose polybody by clicking on it. The Dialog box will ask you then for a file name. Type in a name, say, “test.fig". The .fig extension is optional. The creation process takes a few seconds.

15. (D.4.3.1) More output creation. Click on [Create Figs] and click <scale>, or just press the <k> to select the scaling option. You will be prompted for a figure type. Choose Polybody by clicking on its button. Now, the Dialog Box will ask you for a file name. Type in a name, say, “test.jcl". The .jcl extension is optional, since the program will put it anyway if you omit it. The Dialog Box will ask you for a figure name. At this point it is not important to come up with a meaningful name. Just press [OK].
Figure D.2: Anthropometric Spreadsheet Screen Layout

16. (D.4.1.13) Press <d> <v>. This will select the dynamic stature model. Press <d> <s>. This will select the static stature model. For now you need not to worry what this commands do.

17. (D.4.6.3) Try modifying the stature percentile cell value. Select that cell by clicking with the left mouse button. The Dialog box will prompt you for a new value. Input a value, say 75%.

18. (D.4.1.15) The contents of the spreadsheet can be saved into a file. To do so, press <S> (shift - s) or click [Save sheet] in the pull-down menu (which is activated by clicking at the lower funny looking button in the upper right corner of the sheet). A Dialog Window appears asking you for a file name. Type in a name, say, "test.sas". The .sas extension is optional. A file will be created to register the contents of the spreadsheet. The database saves query history automatically, so you need not to worry about it.

19. (D.4.1.16) Now clean the spreadsheet current contents by pressing <G>, which "cleans" up all the sheets for all the group types, or pressing <L>, which only cleans the sheets associated with current group (i.e. girth, joint limits, or center of mass).

20. (D.4.1.14) Now try reading the previous contents of the spreadsheet stored in the file "test.sas". To do so, press <R> or click [Read file] in the pull-down menu. A Dialog Window appears asking you for a file name. Type in "test.sas". The .sas extension is optional.

21. (D.4.5) To quit the program you either select the [Quit] button from the command panel, or press <Q>.

D.4 SASS Reference Manual

This section is a more or less complete reference manual for SASS. It's probably a bit painful to read, since it goes into detail about each command. Your best bet for using it is to scan through the table of contents, or better still, through the "Contents by Topic" section at the end of this manual, and read about only those features that interest you. Some attempt may be made to increase the size and quality of this section in future versions of the documentation.

D.4.1 Anthropometric Spreadsheet

The anthropometric spreadsheet screen is divided into five main sections: anthropometric group status, global data, command menu, data section, and Dialog line. Each of these screen sections is described below.

D.4.1.1 Anthropometric Group Status

The present version handles four groups: girth, joint limits, center of mass, and strength. When a group has been selected, the color of the group's cell will be changed, and the desired group data file will be loaded and then displayed in the data section. The function to load in any desired group file will be explained in the Command Menu section.
D.4. SASS REFERENCE MANUAL

D.4.1.2 Global Data

This section of the spreadsheet is used to display the “summary” of the data being displayed. It is intended to allow you to have a “global” view of the human figure that you are working on. Currently, the eleven labels are: population, figure type, gender, mass, stature, group percentile, strength type, motion speed, strength mode, lean body factor, and fatigue level. Some of the labels are activated at the click of the mouse. When this is the case we will refer to such a cell as “being active”. A group percentile indicates the percentile of each anthropometric group. Changing the group percentile will cause the data for all segments, body parts, or joints of the displayed group to be scaled to the newly specified percentile value. However, it will not change percentiles for the other groups. The group percentile will allow you to create a human figure with different percentiles in different anthropometric groups if desired.

You cannot modify the label of the currently displayed population, because it is read from an input file. This label is used to indicate the current population data file that you have selected. However, you can choose the Input Data option to load another population file. The definitions of other fields in the Global section are summarized as follows. The two default populations are Natick (Army) General Forces and Natick Air Pilots Data.

- Figure Type
  This indicates the type of human figure model which you want to create using the Jack interface after exiting SASS. Currently, there are two human figure models, Contour and Polybody, which can be selected in SASS. This cell is active.

- Gender, Mass, and Stature
  These indicate the respective current values of the human figure (defaults are male with 50th percentile mass and 50th percentile stature). Gender and stature fields are active, and so they can be modified to fit your needs. Mass fields are dependent on stature, so you can only modify them indirectly through stature modification.

- Strength Type, Motion Speed, Strength Mode,lean body factor, and Fatigue Level
  These parameters are related with strength computation [Pandya 92b]. Motion Speed, Strength Mode, and Lean Body factor are active fields only under the Strength group.

D.4.1.3 Data Section

This section contains the data for each of the items included in the body description model.

The data section for girth, and center of mass is reserved for the display of body parts (objects) or individual segment or joint data for the corresponding percentiles. The leftmost column is reserved for the body part of segment, joint, and object names, while the other six columns are used for the data and percentile display. The segment, object joint name column can be modified by editing the figure files “sfig_def_xx.dat” which contain a description of SASS human body model. sfig_def_gx.dat correspond to figure files for male and female girth. sfig_def_cx.dat correspond to figure files for male and female center of mass. Default files are provided for user convenience. If you need to modify any of these files, make a copy of them in your local directory and perform the desired changes. If you run SASS from this directory it will read the new version of the files rather than the default one. Make sure you understand what you are doing before modifying these files. Any modifications to these files would require to also consider the rest of the *.dat files.

Data and its corresponding percentile can be modified by simply moving the mouse to the desired cell and pressing on the left-mouse button. The color of the selected cell is then changed and a new data value for the selected cell can be entered in the Dialog Box. Pressing the <Enter> key or the [OK] button without typing a new data value leaves the cell unchanged. Changing any percentile will change its corresponding value, and vice versa.

D.4.1.4 Command Menu

The top right corner of the display contains commands for the manipulation of the spreadsheet. These commands allow you to go to database query spreadsheet screen, change the measurement units, read in...
Figure D.3: Command Menu for Anthropometric Spreadsheet

<table>
<thead>
<tr>
<th>Query DB</th>
<th>cm —— in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Data</td>
<td>HELP</td>
</tr>
<tr>
<td>Create Figure</td>
<td>Quit</td>
</tr>
</tbody>
</table>

different input data files, get Help from SASS, create Peabody figure files, and Quit the program.

Fig. D.3 shows a diagram of the Command Menu.

- **Global Conversion of Measurement Unit**
  Depending on the group that one has selected, the conversion units appearing on this command will be different. For example, if one is working on the joint limits and the measurement unit is in degrees, then the command will be `deg —— rad`. This implies that in the current group, one can only convert the measurement unit from degrees to radians. On the other hand, if the current unit is already in radians, then the command will be `rad —— deg`. If one is working on the girth, then the conversion unit will be `centimeters to/from inches`, and for joint limits it will be `degrees to/from radians`.

- **Input Data**
  One of the most important features of this spreadsheet is its ability to display, modify and create generic human figures from different populations. **Input Data** allows you to load different population statistic data files into the spreadsheet. These data files must have a syntax that is recognized by the input functions of the spreadsheet otherwise it will be rejected (see the *.dat files in SASS's data directory). The default input data files of girth, joint limits, and center of mass are `girth.dat`, `jnlmt.dat`, `cmass.dat`, respectively. Make sure you understand what you are doing before modifying any of these files. Any modifications to these files would require to also consider the rest of the *.dat files.

- **Create Figure**
  After specifying the girth, joint limits, and center of mass of a human figure, it can be built and displayed. The command **Create Figure** will create a Peabody figure file that is recognized by Jack. Once this file is created, we can go into Jack and display the newly created figure file. Once Create Figure is active, click on Figure File. You will be prompted for a file name.

- **Query**
  The **Query** command is used for changing spreadsheet screens from the Anthropometric Scaling to the Database Query. After the Query command is executed and the Database Query screen is displayed, one can query the database. The details of the Database Query spreadsheet will be described later.

- **HELP**
  The **HELP** command brings out the extended help pull down menu. Use the rightmost mouse button to select a topic. Then use the mouse to drag the rubber window to a big enough size. If the complete help text fits in the current window, no scroll is necessary. Otherwise use the scroll bar to display more text. Pressing <Exit> the help window disappears.

**D.4.1.5 SASS Hot Keys**

*Hot keys.* Hot keys have been provided for your convenience. Hot keys are an alternative to the mouse oriented selection of commands.

The following is a description of the hot keys:

- `<dv>` Select dynamic stature model.
- `<ds>` Select static stature model.
- `<o>` Select object model.
- `<s>` Select segment model.
D.4. SASS REFERENCE MANUAL

- Select girth group.
- Select joint group.
- Select center of mass group.
- Select strength sheet.
- Select contour model.
- Select polybody model.
- Select male sheet.
- Select female sheet.
- Create figure file.
- Create scaling file.
- Write to local port (single event).
- Write to local port (multiple event).
- Toggle local port output.
- Change Strength Subject.
- Local clean up.
- Global clean up.
- Read file.
- Save contents.
- Goto Database.
- Reload person.db file.
- Reload strgh.param.db file.
- Reload jntlim.db file.
- Reload cmass.db file.
- Terminator quit.
- Show the help message.

In most cases the Dialog line will show the option selected. If in doubt whether an option is ON or not, do the selection again.

D.4.1.6 SASS Pull Down Menu

The pull-down menu is another way to select commands or access help topics, or build up a query (in the database).

- Figure D.4 shows the pull-down menu as it would appear on SASS window.
- In the spreadsheet (girth, joint limits, center of mass), press the lower funny looking button to activate the command pull-down menu and use the right mouse button to select an item from the menu.
- The help menu is accessed by clicking the <HELP> button, or by pressing [H].
- In the database, click <Query DB> to start the query building mode.

D.4.1.7 Dialog Line

SASS's dialog line is used to display messages or to collect information keyed in by you. It comes in a few different flavors, but the interface is consistent across all of them.

- Pure messages appear on the line with the message for a short time, and then disappear. If you miss the information message, tough. Such messages are usually informational, or of the sort where you can re-issue the command and re-read the message.
- Some message lines display more important information, and remain on the screen until you enter the required information.
- Some Dialog lines require some text input (numeric values).

D.4.1.8 Dialog Box

As the dialog line it comes in a few different flavors, but the interface is more or less consistent across all of them. In a file name Dialog box, you must specify a valid Unix path name. In a numeric Dialog box, you should enter a valid number (i.e., no alpha characters).
The box is totally mouse-based. Using the Dialog Box is quite simple. To move the cursor, use the arrow keys, or point with the mouse. To erase (part of) the current number drag the mouse on top of it with the left button pressed. This should highlight the number. Press the <Delete> or the <Backspace> on your keyboard. A faster way is to press <ESC>. This will clear up the Dialog Box entry.

If selecting a file, type the name of the file, or click twice in the adjacent window on a specific file name. Figure D.5 shows display of the Dialog Box used for file name input.

D.4.1.9 Resizing the Window

SASS window cannot be resized.

D.4.1.10 Scrolling

Usually, the files you view will have more items than can appear on the screen. For this reason, a scroll bar appears along the right upper edge of your window. Use the mouse to activate the bar. In the database, use <p> and <n> to move across sheets horizontally, and use the funny looking buttons (concentric rectangles) in the upper right corner to move between pages vertically. For the strength spreadsheet use the upper funny looking button to move between pages.
D.4.1.11 Making Selections with the Mouse

To place the cursor in a specific cell, move the arrow cursor with the mouse until it points to the desired cell, and click with the left mouse button. You don’t have to be exactly accurate — SASS lets you hit anywhere in the cell.

D.4.1.12 On-Line Help

There is on-line summary of the hot keys available through the hot-key <?> At any moment, you may invoke the Help Screen. A summary of the hot keys is displayed on screen. (See D.4.1.5 for further details).

Also, there is an extended on-line help that can be accessed by pressing <H> (notice this is shifted). A menu of help topics appears. Use the rightmost button to open the drawers of the pull down menu, and select the particular topic you are interested in. A rubber band window appears. You must use the mouse to open it to a fair enough size. If the text for the help window does not fit in the space provided, a scroll bar shows up as well, which must be dragged with the mouse to display more text on the current window. Press <Exit> when you are done.

D.4.1.13 SASS Special Modes

The Anthropometric Spreadsheet works under four computational models. The dynamic or the static height models, and the object or segment models.

- **Dynamic height model.** Under this model, the global field stature varies dynamically as the segment (or object) dimensions in the stature path are modified. A limit is imposed, though, to disallow the possibility of body heights beyond or under that of the population range (see D.4.6.3).

- **Static height model.** The global stature is fixed while the segments’ (or objects’) dimensions in the stature path are adjusted to keep the value of the stature constant (see D.4.6.3). If it is not possible to achieve this goal, the stature will vary up to the first closer goal. You may correct this to the desired stature (via the stature field). See section D.5.

- **Segment and Object Models.** For the Girth group, there are two alternative working models. The segment model displays the information in a segment by segment basis. This is convenient when there is a need to visualize the information for a specific segment of the body. If you want to work at a
higher level of abstraction, the object model (default) provides a way of modifying dimensions in a (body) part by part basis (e.g. arm, leg, torso, etc). The spreadsheet will automatically adjust the internal components of each of the (body) objects (see D.4.6.4).

D.4.1.14 Reading a file

To read a spreadsheet contents file (i.e., .sas file), press <R> or click [Read file] in the pull-down menu. A Dialog Window appears asking you for a file name. Type in a valid name. The .sas extension is optional. Alternatively, click twice in the name of the file you want to select. The only files that can be read by means of this command are those created through the "Save" facility of SASS. (See D.4.1.15).

D.4.1.15 Saving the spreadsheet contents

The contents of the spreadsheet can be saved into a file. To do so, press <S> or click [Save sheet] in the pull-down menu. A Dialog Window appears asking you for a file name. Type in a valid name. The .sas extension is optional. Alternatively, pick the name of an already existent file, which you want to overwrite. The files created through this command can be read by means of the "Read" facility of SASS. (See D.4.1.14).

D.4.1.16 Clearing up the sheets

To clear up the spreadsheet current contents press <G>, which "cleans" up all the sheets for all the group types, or press <L>, which only cleans the sheets associated with current group. The default percentile is 50th. The database cannot be cleared, but a new query will bring up a (possibly) new set of results. What can be cleared is the query history. To do so, delete the "sass-qh" file that you can find in your root directory.

D.4.2 Strength Sheet

Figure D.6 shows a sample of the strength sheet display.

The strength sheet compute strength values [Pandya 92b], for a particular set of angles and strength equations. A regression model, which allows the prediction of dynamic isolated joint torques from an individual’s lean body mass, is used.

Collected (from a 14 subject population [Pandya 92b]) data for each joint, direction and velocity is normalized with respect to that joint’s average, and merged into files used to perform least squares regression. As a result, a table of normalized population curve coefficients for each joint axis direction and velocity is obtained. Additionally, another regression table is built that relates average torque to lean body mass for an individual. These two tables are internally maintained by SASS.

There are two possible sources for strength equations:

- INDIVIDUAL, which are equations based on a particular person's measurements
- PREDICT, which are equations that consider a population of individuals

To select between modes, press the lower funny looking button at the top right of the sheet. To move across pages use the upper funny looking button.

These equations vary depending on the velocity, for INDIVIDUAL case, and on the velocity, weight, lean body factor, for the PREDICT case.

The PREDICT case generates new equations on the fly, based on the current weight, lean body factor, and velocity. The velocity parameter ranges between 60 and 240 rad/s.

The INDIVIDUAL case uses a fixed set of equations, of a given person (by default there are 14 subjects), but the equations chosen depend on the current velocity. By default, the individual selected is “sub01”. To change subject, press <V> and provide the name of the new subject.

When you create a figure, the current strength equations are appended to the figure file. A few things must be noticed. Currently, PREDICT equations consider only the upper body joints. So the lower body joints can be displayed only in INDIVIDUAL mode. Also, when creating a figure, you must go into the mode whose equations you wish to be included in the figure, i.e., INDIVIDUAL or PREDICT. This change
of mode can only be done while in the strength sheet. Finally, the strength equations for the lower body joints are included in the figure only if the figure is created while being in the lower body joints section (last page) of the strength sheet.

D.4.3 SASS-Jack Interface

SASS v.2.2.1 can now work with Jack in an interactive way. The protocol to be followed is described next. Under this proviso, SASS is run from within Jack. To do so, execute the command run sass from within Jack (human-anthropometry sub-menu). This will cause SASS to be executed. You do not need to do this step if you prefer to run SASS directly from the command shell.

SASS figure creation functions are now, by default, directed to Jack. If you wish to send figures to a file, rather than to Jack, press <X> from within SASS. The dialog line should now read PORT OFF. If it reads PORT ON, press <X> again.

If you do not wish to use a human figure created through SASS, but perhaps a human figure from the library of figures, or from any other source, you should set this human figure in a particular Jack environment, and use SASS interactively to scale this figure.

Whatever the source of the human figure, in order to do an interactive scaling, a communication socket must be established between SASS and Jack. Notice that Jack must be an earlier process (i.e., must be run first) than SASS for the connection to be successful.

In Jack, an environment containing (at least) a human figure must be set up, as we just mentioned. Then receive data from sass command within Jack must be executed. This will make Jack enter a wait mode.
In SASS, a specific scaling (i.e., the desired girth values) must be first defined, or else the default ones will be used. Then <w> must be pressed. The name of the human figure to be scaled is requested. This is only for the case of having multiple figures within the environment.

Once the port is opened, SASS sends scaling commands to Jack. When it is done, the process must be repeated if another scaling needs to be set up.

An alternative to individual scaling is the REPEATED scaling mode. This mode is useful if one needs to browse among the possible height percentiles available, to decide which one fits the best in a particular environment. The protocol for this is as follows. Jack must be set into the waiting mode, as described before, and then in SASS <r> must be pressed, to make the program enter the desired repeated scaling mode. Once in this mode, the program will repeatedly request percentile values. For each of the values provided, a new scaling is applied to the human figure displayed in Jack. When the desired percentile is found, press <Enter> or click <Ok>, without introducing any new value.

### D.4.3.1 Figure Scaling

The reason you should consider using direct port communication with Jack rather than loading figure files is simple. Consider the situation in which you, working with Jack, want to determine the percentile (dimension) ranges of the human figure to comply with a given task, that is, the problem of finding the specific figure (%) that can fit in a particular working environment. One can attempt to read each of the possible figure files out of Jack libraries and try to keep the figure in the position we want. The other (more sensible) option is not to load different figure files, but instead, to apply different scalings to a given figure. Then the (same) figure can be scaled using different scalings to find the one that best suits the given environment. This is not only faster but it is even more appealing to you, as you can see the changes right away on screen.

### D.4.3.2 Figure Creation

Figures D.7 and D.8 shows display in Jack of figures created in SASS.

Why use figure files rather than scale files? Scaling files are good when you require easy access to multiple sizes for the same figure. If, on the other hand, you need multiple figures of different sizes, then figure files should be used. Also, figure files are a precise image of SASS's segment / object dimensions, while the scaling file is a global approximation of the body parts, i.e., the body is scaled by Jack based on the global (body part) scaling values defined by SASS.

As a good alternative to producing scaling files, you can use the port option (see sections D.4.3 and D.4.3.1) to interactively scale an already existent figure in Jack. This is perhaps the easiest way of finding the right scaling for a body within a given environment.

To do interactive figure creation, press <X>. The dialog line should read PORT ON. Otherwise press <X> again.

If this is interactive creation of figures, you must prepare Jack to wait for data from SASS. To do so execute in Jack the command receive data from sass.

Now click [Create Fig]. Then click on [Create Figure] and specify the type of figure you wish to create, namely, polybody or contour.

If not interactively creating figures, (i.e., figure will be stored in file) you must then specify a file name. The extension „.fig” is optional.

If you are in the database sheet, the procedure is similar, but you will be asked additionally to provide velocity values for some joints. If you don’t care for this option (only if strength computation is relevant), just click [OK]. Otherwise, provide the desired values.

To create scaling files, click on [Create Fig] button, select a figure type, and then click<scale format>, or just press the <k> to select the scaling option. The Dialog box will also ask you for a file name. The .jcl extension is optional, since the program will put it anyway if you omit it. The Dialog Box will ask you for a figure name. This should be the name of the figure you intend to apply the scaling to. Just press [Ok].

To create figure files, click on [Create Fig] button, select a figure type, and then click<figure format>, or just press the <b> to select this option. The Dialog box will also ask you for a file name. The .fig extension is optional, since the program will put it anyway if you omit it.
D.4.4 Database Query Spreadsheet

This spreadsheet is used to build a query and display the result of the query. By default the data used by this sheet is that collected by NASA [Pandya 92b] on fourteen subjects (6 females and 8 males).

As shown in Figure D.9, this spreadsheet is divided into five sections: summary data, command menu, scroll buttons, query list, and Dialog Line. Each of these sections is described below.

Figure D.10 shows a sample of the query sheet display.

D.4.4.1 Summary Data Section

This section displays the query requested by you under the label “Query”. The typical query is to find the list of individuals who satisfy certain constraints on global or local anthropometric data. Example queries are: “find a person who has a right upper arm longer than 28 centimeters” or “list any right handed females who have elbow flexion strength greater than 20 ft-lbs.” The summary data section also displays the summary of the result of the requested query. The label QUERY: shows the last query executed. It the is a scrollable line, for there may be queries that are longer than the space provided. To scroll the query line to the right, click the left mouse button on the red < character beneath the query line. The red > has the opposite effect, i.e., scroll back. The label Current Display Page indicates the currently displayed page and the total number of pages available for vertical scrolling. Current Display Section indicates the currently displayed section and the total number of sections available for horizontal scrolling. Total Data in Query List, indicates the total length of the list currently displayed in the Query list section.
D.4.4.2 Query List Section

This section displays the list of individuals that satisfy the requested query. The leftmost column shows the ID number and the name of an individual while the subsequent columns show the set of related attributes found in the query. For example, a query such as "find all persons with mass >= 65kg" would display the mass attribute as well as global information like gender, stature, handedness, and age. Each row contains the relevant information of an individual.

D.4.4.3 Scroll Buttons

These are used for paging purposes in the Query List section. Click on the upper button to scroll to the previous page, or the lower button to the next. For the purpose of horizontal scrolling, use <n> to get to the next section and <p> to return to the previous.
D.4.4.4 Command Menu

As in the Anthropometric spreadsheet, the top right corner of the database query display contains commands for the manipulation of the spreadsheet. These commands allow you to further restrict the amount of information displayed in the Query List section as well as the list of individuals found in the current query. The present Command Menu section of Database Query Spreadsheet is shown in Figure D.11. The function of each of these commands is described below.

The following relations are implemented in the current version of SASS. The attributes (fields) of each relation are listed under the corresponding relation together with brief comments. For example, Girth relation contains the data of thickness, width, length, and mass of each segment in the body for an individual (person). It includes attributes of id.num, seg.name, thickness, width, length, and mass. Id.num is an identification number used to identify an individual uniquely. Every relation in the database has this attribute. It allows us to link two relations together and find all the data stored in the database for an individual if desired.

Each relation is stored as a “flat” file in fixed format. The order of the fields “must” follow the one in the corresponding relation. See appendix for more details on files format.

The default data is stored in the files person.db, girth.db, jntlim.db, cmass.db, strth.param.db

The strength data files are stored in the $JACKSASSLIB directory. Note that each subject has a subdirectory within $JACKSASSLIB.

- Project
  Project allows you to remove some of the attributes (columns) in the Query List section. This command
comes in handy when you need to juxtapose a selected number of attributes which are displayed in separate sections for comparison purpose. Clicking on this command button causes the currently displayed attribute labels to be highlighted. Click on an attribute label to toggle the highlight, all attributes which are not highlighted will be subjected to removal. Middle mouse button reverses the status of each attribute, right mouse button removes all attributes which are not highlighted.

- **Select**
The Select command operates in a similar fashion as Project. It allows an on-screen selection or removal of individuals (rows) from the Query List section.

- **Query DB (Query Database)**
This command allows you to make a query request and get the answer back from the database. After you select Query DB from the Command Menu section, the following message will show on the Dialog line:

```
Building Query:
```

This indicates that you can press the right-mouse button to build a query from pop-up menus. The query that you are building will be displayed on the Dialog line. You can edit the query anytime before the command Done on the pop-up menu is selected. Once the command Done is chosen, the query built by you will be executed.

The result will then be displayed on the Query List and the Summary Data sections of the spreadsheet. If no person is found for the requested query, only the attribute-labels of the queried relation(s) will be displayed.

There are five types of pop-up menus for building a query. Each of them is described below.

- **Query Menu**
The present Query Menu looks like the following:

```
Query Menu
Global Information
Anthropometric Groups

Global Information
```

This is the first pop-up menu which appears in building a query. The command Global Information or Anthropometric Groups allows you to build a query on these data. Selecting either of these two commands will lead to the second type of pop-up menu. The command / allows you to construct complex query strings through the use of parentheses, an auto-completion feature is available to match all unclosed parentheses. The command Abort allows you to quit the loop of Query DB without executing the query. This is useful for scrapping the query and starting over if you do not like the one under construction.

- **Global Information or Anthropometric Group Menu**
This is the second pop-up menu that will show up when building a query. The present Global Information Menu looks like the following:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project</td>
<td>Create Figs</td>
</tr>
<tr>
<td>Select</td>
<td>Qry History</td>
</tr>
<tr>
<td>Query DB</td>
<td>Extend Qry</td>
</tr>
<tr>
<td>Exit Query</td>
<td>Quit</td>
</tr>
</tbody>
</table>

Figure D.11: Command Menu for Database Query Spreadsheet.
This menu allows you to query the global information listed. Selecting any item from this menu will lead to the selected global data menu which provides you with the available choice of values. For example, picking Strength Type will lead to the Strength Type Menu that provides three possible strength types for you to choose from. The Strength Type Menu looks like this:

<table>
<thead>
<tr>
<th>Strength Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isometric</td>
</tr>
<tr>
<td>Isokinetic</td>
</tr>
<tr>
<td>Isotonic</td>
</tr>
<tr>
<td>Previous Menu</td>
</tr>
</tbody>
</table>

The present Anthropometric Group Menu looks like the following:

<table>
<thead>
<tr>
<th>Anthropometric Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Girth</td>
</tr>
<tr>
<td>Joint Limits</td>
</tr>
<tr>
<td>Center of Mass</td>
</tr>
<tr>
<td>Strength</td>
</tr>
<tr>
<td>Previous Menu</td>
</tr>
</tbody>
</table>

This menu allows you to choose an anthropometric group to query. After picking the group from this menu, the selected anthropometric group menu will appear. For example, picking Girth will lead to the Girth Menu which lists all the segments of a human figure. You can choose any one of these segments to query. The present Girth Menu is shown below.

<table>
<thead>
<tr>
<th>Girth Menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top Head</td>
</tr>
<tr>
<td>Bottom Head</td>
</tr>
<tr>
<td>Neck</td>
</tr>
<tr>
<td>Upper Torso</td>
</tr>
<tr>
<td>Center Torso</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>Lt Toes</td>
</tr>
<tr>
<td>Rt Fingers</td>
</tr>
<tr>
<td>Lt Fingers</td>
</tr>
<tr>
<td>Previous Menu</td>
</tr>
</tbody>
</table>

Axis Menu

In a human figure, segments are defined in terms of width (x axis), thickness (y axis), and length (z axis) and joint limits are defined in three different rotation axes. This menu allows you to specify the segment or rotation axis. It is shown below.
Axis Menu
- X-Axis-Value
- Y-Axis-Value
- Z-Axis-Value
- Previous Menu

- Relational Operation Menu
  This menu contains operations that are allowed in a query. The present Relational Operation Menu looks like the following:

<table>
<thead>
<tr>
<th>Relational Operation Menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less Than (&lt;)</td>
</tr>
<tr>
<td>Greater Than (&gt;)</td>
</tr>
<tr>
<td>Less Than or Equal (≤)</td>
</tr>
<tr>
<td>Greater Than or Equal (≥)</td>
</tr>
<tr>
<td>Equal (=)</td>
</tr>
<tr>
<td>Not Equal (≠)</td>
</tr>
<tr>
<td>Previous Menu</td>
</tr>
</tbody>
</table>

After an operation in this menu is selected, you will get the following prompt:

**Type in value:**

You are expected to enter a number and then hit the *enter* key to get to the next pop-up menu.

- Conjunction Menu
  The Conjunction Menu is shown below.

<table>
<thead>
<tr>
<th>Conjunction Menu</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
</tr>
<tr>
<td>Or</td>
</tr>
<tr>
<td>)</td>
</tr>
<tr>
<td>Done</td>
</tr>
<tr>
<td>Previous Menu</td>
</tr>
</tbody>
</table>

The conjunctions *and* and *or* allow you to combine simple query constructs to form a more complicated one. If the conjunction *and* or *or* is selected, the first menu (i.e. the Query Menu) will appear again and the loop of building a query will be repeated. Previously inserted left-parenthesis can be matched using the *)* command. *Done* will stop the loop and send the query that was just built to be executed. Note that every menu except the Query Menu contains the command Previous Menu. This command allows you to go back to the previous menu and erase the unwanted part of the query. This also enables you to edit the query while building it. If the query is too messy to edit, you can execute the command Abort in the Query Menu to scratch it and start over.

- Exit Query
  This command allows you to get back to the Anthropometric Spreadsheet that is left before entering the Database Query Spreadsheet.

- Create Figs
  This command creates a Jack figure file for each of the individuals in the query list. Destination file names are requested from the user via a pop-up dialog box. The type of figure to generate has to be specified in the Anthropometric Spreadsheet.

- Qry History
  A history list is kept internally for all executed queries. You can go back to view the result of a particular query by invoking this command. A dialog box with a scrollable list shows the 50 most recent queries, select one of them by clicking the left mouse button directly over it, followed by the OK! button. Click on the Cancel button to abort this operation.
• Extend Qry

This command allows you to build on the currently displayed query. The Conjunction Menu will be invoked once again so that either of And or Or connectives are once again available for extending the current query.

D.4.5 Exit SASS

Clicking [quit] exits SASS. Also, you may use the hot key <Q> to exit. The difference between this methods of exiting the program is rather important and should be kept in mind at all times. Using <Q> to terminate the program doesn’t prevent you from losing the sheet contents. Clicking [quit] always goes through a protection mechanism that verifies whether changes have been made to the spreadsheet contents, and if so, allows you to save those changes in a file.

D.4.6 SASS - A closer look

This section of the manual goes into some of SASS’s implementation details. It should be skipped in a first reading of this manual. In fact, you should not be concerned too much with how SASS operates internally. The following pages are only intended to give you an idea of what goes on.

SASS has been designed and implemented keeping an object oriented philosophy in mind. What we mean by object oriented is reflected in the fact that a hierarchy has been established to define the human figure components.

D.4.6.1 The hierarchy of the human figure

The structure used is a hierarchical one (a tree). At the bottom of the tree, the leaves correspond to the segments. The internal nodes correspond to, what we call, the body parts or body objects. The root of the tree is reserved for storing the figure’s information. The body parts are composed of body segments and the figure itself is composed of body parts. One can think of the figure as corresponding to the complete tree. There is another category, the joints, which has been appended to the root of the tree. The joints are the linkages among the segments, and also among the body objects.

A figure can be defined as a collection of body parts, joined together by joints. Each body part, in turn, can be defined as a collection of body segments put together by joints. Each segment has been specified with an access code, a segment type, and a list of sites. Joints are defined similarly. Body objects are specified by a type, an access code, and a list of components, namely a list of segments, joints, and sites related to that object.

This hierarchical definition is user definable, as SASS parses it each time it is run. Indeed, you can modify the specification of the objects, segments and joints and even the figure itself by just writing down a figure description file _s.fig.def.xx.dat. In this way, it is possible to create or delete objects and/or segments as needed. Be careful in doing so, as all the *.dat files are interrelated.

The geometry interface poly факт*.dat and cont факт*.dat, as well as the Peabody figure models pconthum.dat and cconthum.dat, are intended to leave an open end, so that the user can modify them if necessary. They have been made external, so that they are available to the user.

The geometry interface, consists of a set of factors that maps the values established by SASS to the current psurf geometry available in Jack. Should you decide to create your own human geometry, or modify the existing one, by modifying this interface, and the figure models, you will be able to reconfigure SASS.

To summarize, SASS can be seen as a black box, which provided with a hierarchical body definition, appropriate population data, a figure file following the Peabody grammar, and a geometry interface, generates figure files to be used by Jack. Of course, even though SASS is open ended, it has been specifically designed to manipulate human-like figures, i.e., there must be a head, a couple of arm, a couple of legs, etc.

D.4.6.2 The rule system

SASS not only works on relations but also is rule based. As an example, SASS currently defines a rule for computing the height of an individual as the sum of the segments’ lengths in a path that goes from head to feet. For those segments in the path, the rule allows variation in their lengths if the stature changes and vice
versa, to change the stature if the length of any of the segments in the path changes. There is an alternate rule that keeps the stature fixed and adjusts the segments' lengths accordingly, if the length of one of them varies.

Other rules include changing the mass according to the stature and rules for checking proper bounds in segment (object/figure) dimensions.

D.4.6.3 Rules in SASS: Stature Definition

The stature of the human figure is computed using two different rules. In one case, the stature is kept variable. If the stature varies, the segments' lengths in the stature path vary accordingly. Similarly, if the length of any segments in that path varies, then the stature changes. The underlying criterion for doing these changes is a linear one. The segments in the stature path have been defined as: (head, neck, upper torso, center torso, lower torso, upper leg, lower leg, feet). The length of each of these segments, except for the feet, is computed as the girth value in the z coordinate. For the feet, the length is computed as twice the girth value in the x coordinate (since for the feet, the girth in the z coordinate is the longitudinal dimension, and the width corresponds to their height). The doubling factor is due to the fact that in SASS the width and thickness (s, y) values are displayed halved. It should be noticed that the thickness and width of the segments are not affected by these changes, for there is no rule to decide the effects of stature changes in these parameters.

The updating process is done carefully, for it might happen that modifying the length of a given segment violates the range of possible stature values, or conversely, if the stature is changed, this change might not be satisfiable by variations in the segments' lengths.

The other case considers fixed stature. The idea is to adjust the segments' lengths along the stature path if the length of one of them varies, such that the global length (stature) remains constant. While this might appear easy to do at first, it is not a trivial matter. To understand why, we must study how the segments' dimensions are obtained. Each segment's dimensions can be seen as a triple (x, y, z) of values. This triple of values, is obtained by interpolation from actual data you provide. This "real world" data corresponds, in fact, to the value of the girth in each of these coordinates for a set of different percentiles (e.g., 5th, 50th, 95th percentiles). SASS provides a given triple (x, y, z) for percentiles in the range 1-99 by means of interpolation (also, if you specify a triple, SASS provides a percentile value corresponding to that triple). Thus, a segment's dimensions is constrained by the "real world" values range. Furthermore, the stature itself is restricted by a "real" set of values (for each of the percentiles). When you specify a particular change in the triple (z coordinate) of a given segment, the underlying rule attempts to satisfy the constraint of fixed stature, that is, it tries to keep the stature value constant.

For example, assume the length of the head has decreased. To keep the stature fixed, the lengths of the other segments in the stature path must vary in the opposite way. Currently, the modification is done in a linear way, since there are no rules to define this otherwise. But it might be the case that in the updating process, one of the segment's dimensions (namely length) cannot be satisfied. In other words, the resulting dimension is out of the range established by the 1-99th percentile values. In this situation, the rule sets up the length to its closest limit value (11th percentile value or 99th percentile value), and tries to satisfy the requirement of fixed stature by modifying the remaining segments in the path. (See section D.5). Notice that there is a possibility that the stature cannot be kept constant. There is one more step involved in the updating process that will be discussed later. In this mode (fixed stature), if the stature is globally varied, the segments change correspondingly (if possible).

D.4.6.4 Object level

As we discussed before, a figure is built up as a hierarchy. The segments in the lowest level, the body parts (objects) in the next level, and the figure itself as the root level. The body objects are defined (by the user) as sets of segments and joints. For instance, the object 'leg' is defined by default as a set containing two segments ('upper leg' and 'lower leg') and two sites, and six associated joints. For the matter of the following discussion, it is not relevant what the sites or the joints are, but one can think of a simplified object involving only a set of segments.

The object level is an abstraction of the idea of body parts. So we associate each object with a body part. It is important to keep in mind that the "real world" measurements are done on a segment basis. The
D.4. **SASS Reference Manual**

Objects (body parts) are defined to provide additional flexibility to the user. As the internal structure of each body part can be specified by you, one can consider having as many parts as necessary (or as segments there are). By default, SASS has defined eight body parts, namely, head, torso, left arm, right arm, left leg, right leg, left foot, and right foot. These objects encompass most of the (user defined) segments.

Having objects allows the user to perform global modifications on a per-body part basis, as opposed to doing localized changes on specific segments. Although it is possible to change values for a particular segment, it is generally desirable to be able to do modifications on a body part, as a whole. The idea of having body parts presents some difficulties though. When body parts are introduced, the rule system must consider performing the appropriate (coherent) updates on two different levels simultaneously. If you change values on the segment level, these changes are reflected also at the object level, and conversely, when changes are done in the object level, these changes affect the segment level values. Also, recall that changing the segment level values was governed by a set of rules. There is an equivalent (compatible) set of rules for the object level. For instance, changing stature is governed by rules in the object level (and in the segment level).

**D.4.6.5 What is an object?**

In a sense, objects can be considered as clusters of segments, and each time an object is accessed, the access is redirected to the corresponding segments, and conversely, if a segment is accessed, all the objects containing that particular segment are accessed. But, there is more under the definition of a body part. Actually, the object’s dimensions are approximated by considering the bounding box around the segments of which it is composed. In this way, a body part comes to life as a triple of \((x,y,z)\) values. Why bother doing this? At first it might seem unnecessary, since the components of an object, i.e., the segments have some associated \((x,y,z)\) triples already. However, there are two good reasons that justify our approach. In the first place, using a bounding-box strategy, we can bound the dimensions of the segments (components) of a given object. But also, it allows us to have two sets of dimensions: the expected dimensions and the actual dimensions. The expected dimensions, are those determined by the bounding box approach. These are called expected dimensions since they are precisely those dimensions expected from a bounding-box setting (the usage of the term “expected” should not be confused with the term “average”; in fact, here expected case should be thought as a worst case, not an average case).

The actual dimensions are the dimensions of the body part when we think of it as a cluster of segments. Thus, the actual dimensions reflect accurately (up to the accuracy of the segments’ dimensions) the dimensions of the object.

Having these two dimension sets provides a way of constraining the growth of the body parts. The following rules come across. If a segment (member) in an object grows (or shrinks) then it should not grow beyond the limits of the object’s expected dimensions for a given percentile, if we want to restrict an object’s dimensions to be of a certain percentile. So we can, for example, try to adjust the dimensions of the other segments in the object’s segment set so that we keep the objects percentile fixed.

It is important to understand the back and forth process that goes on between objects and segments. On one hand, the global dimensions of the body can be, for instance, those of a 50% (standardized) human being, but we know that all of the body parts need not be 50%. In fact we do not have a rule yet to specify the percentile of the body parts (segment-wise) for a given global body percentile. So, keeping that in mind, we must be able to change dimensions of the body objects (segments) to comply with all the possible compositions of a 50% body, for instance, but we must be careful when specifying other rules for, let us say, stature. We need to make sure that a given change of stature does not break any other rules, that is, we must assure that the resulting body composition (i.e., the percentiles of the body parts (objects/segments)) are those valid for a 50% body. Also, we are required to comply with the restrictions on the segments’ (in this case on the stature path) dimensions, i.e. we cannot scale a segment beyond the limits established for that segment by the population data.

Also, we need to assure that the stature modification rules are complied with (i.e, those rules we mentioned before in which the segments’ lengths are modified following a specific layout; currently modifications are done linearly). It seems that a possible solution, in this particular case, is along the following lines. If the stature is modified, then a new global percentile is computed. For that new global percentile, we have a specific rule telling us what the possible compositions are. Thus, we use these compositions as our rules for
doing the segment length modification, (instead of doing it linearly as it is done in the current version).

There is no conflict, but that is only if there is a coherent definition of the possible compositions and the stature-path segments' length, i.e., the compositions must agree with the segments' length under the population data being used. In other words, the compositions are not unique, they are dependent on the population data used. To illustrate this, suppose we have the following (partial) composition set feet 30\%, legs 45\%, torso 60\%, head 40\%... for a 50\% body. Then suppose we want to change the stature in such a way that the resulting body percentile is 60\%, and the analogous (partial) composition set is (feet 40\%, legs 56\%, torso 50\%, head 40\%, ...). Then we scale the objects in the stature path (which are those listed in the composition sets) to comply with this second composition set. But, we must be sure that there is no conflict in doing so, that is, for instance, the feet might be able only to grow from a 30\% to a 40 \% under the population data being used.

There is an inherent need for the compositions to be determined under a given population, i.e., different populations will have different compositions. To solve that problem, we must make sure that the scaling (of the segments) resulting out of this complies with the object's (body part) constraint, i.e., the bounding box limitations. This should be the case if we have composition sets that agree at both the segment level and the object level. In the previous example, for instance the compositions were stated at the object level. There must be an equivalent composition at the segment level. Following this example, the segment version of the composition for the 50\% figure is, for instance, (... upper leg 45\%, lower leg 60\%, upper torso 76\%, center torso 57\%, lower torso 45\%, ...), assuming legs decompose into two pieces and torso into three pieces.

But what if the compositions, though based on a particular population data, are not available for all the possible percentiles (with good luck we hope to have one for a few of the percentiles). We would have to interpolate compositions (if it is sound to do that) and make sure a given segment's length is not violated (according to its percentile range) when trying to go from a composition for the 50\% figure to that of the 60\% figure. If we had only one such composition to work with to account for all possible compositions on the percentiles range, then it would be necessary to make sure that this composition is not violating the range of values a given segment's length can have for the population under consideration. This is basically what happens in the present version of SASS. Since we do not have a composition analysis available, we have assumed decompositions are unique for a given population (i.e., one composition for all the percentiles (not one for each)) and furthermore, this composition is linear, i.e., for a 50\% figure (feet 50\%, legs 50\%, torso 50\%,...) and similarly for the segment composition (... upper leg 50\%, lower leg 50\%, ...). This has sensibly increased the difficulty of the problem because such an assumption is far from being applicable to real world situations.

This has ended up in a need for additional rules in SASS to verify that there is an agreement among all parts. Recall, for instance, the stature problem. In that case, we are considering compositions to be linear. So we need to be especially careful not to end up with a segment's length violation. To avoid that, we limit the growth of a segment to its 1\% and its 99\% (i.e., below and above limits). If we do not achieve the desired global growth, i.e, the local segment's growth was not sufficient, then we go and adjust the other segments in the stature path. This is done in an iterative way. Also, observe that we have to keep track of two levels of abstraction, that is the segments and the body parts. It is necessary to double check, once for the segments' lengths not to violate their limits and once for the objects' lengths not to violate their limits. This is necessary because the objects' composition has been assumed to be linear. A similar situation arises when considering the other way around, that is, modifying a segment's length implies a careful set of steps along the hierarchy to keep track of this modification's effects on the objects' lengths and then the effects of these on the global length (i.e. stature) so that the resulting stature has a value between its percentile limits. (If the stature is kept fixed, then we do not go all the way up in the tree but we need to perform a readjustment of all the other segments (objects) to assure the stature is kept constant, whenever possible.)

### D.5 Tricks

There's nothing new in this section; it simply documents some tricks that have proven useful for SASS sessions.

- **SASS** girth sheet is goal oriented. When defining a particular value for the stature or for a specific segment or object, SASS will try to satisfy the current stature constraints (if the static stature mode
Because segment/objects dimensions vary within a range, SASS may find that in keeping the stature constant or in modifying the stature to a certain value, it has reached a point in which one of the segments/objects in the stature path is at its maximum or minimum range value, so the program is going to stop at this point and provide you with a stature (or segment/object) value different from what you specified. You can ask SASS again to satisfy the particular value you specified before. In fact, you may continue doing this a few times. The program will try to achieve the goal by resizing other segments/objects along the path, and stop each time one of the segments reaches one of its limits. If you want to avoid this when trying to set a specific stature, you should go to the segment sheet, <a>, and modify the segments along the path yourself. In other words, for segments/objects in the stature path, when modifying their z (length) values, the value you type in at the prompt is taken by SASS to be a goal. So you might observe that you type for instance 5th, and SASS gives you another value (usually close to the one you specified). SASS stops every time a segment lower or upper limit is exceeded. You must continue to specify the desired value until it is achieved. This shouldn’t take more than a few tries, if the goal is attainable at all.

- Because there are no rules as to what is the correct proportionality of segments for a given percentile body, selecting of a specific global group percentile, and then generating a figure file or scaling file will cause all segments in the body to be scaled to this given percentile value. For a more sensible scaling, after setting the global group percentile, use the height percentile to set the scaling generated to be within the population stature range. Once a given percentile height is established, scaling of the segments is propagated automatically by SASS. Notice that this propagation is linear, and affects only the length (z-axis) of the segments along the stature path. For example, suppose we want a figure of 95th percentile. Set the global group percentile to 95. Then correct the stature path segments by setting the stature percentile first to 95th.

- In the main spreadsheet, the values for the width (x-column) and the thickness (y-column) are displayed halved. So when considering on doing changes in this to columns, keep this in mind. The length values (z-column) are displayed in full.

- Learn to use the hot keys. They save you a lot of time. Alternatively, learn to use the pull down menu. However, it is usually faster to use the hot keys.

- If you are trying to fit a figure within an environment in Jack, use multiple scaling files, instead of creating multiple figure files for the different percentiles. Loading figure files in Jack takes a lot more time than it takes to load a scaling file. Even better, use the port command to do an interactive fitting.

- When creating figure files (i.e., not interactive creation with Jack) the .fig extension for the file name is optional (so do not type it). The same goes when creating scaling files. Using a .jcl extension is optional. Similarly, when saving the spreadsheet contents into a file, the .sas extension is optional, as it is when reading such a file.

- When using a port connection with Jack, be careful to specify receive data from sass (a Jack command) in Jack before you attempt an interactive scaling, or a figure creation. Also remember to run Jack first, then SASS. Failure to do this step will result in an unsuccessful connection.

- Observe that pressing <Q> (or clicking [Terminator Quit] in the pull down menu) will NOT allow to prevent an accidental lose of the spreadsheet contents. If you know what you’re doing, this is the fastest way to exit the program. However, if you want to stay on the safe side, exit the program by clicking [Quit] in the command pad. This method always checks for changes in the spreadsheet contents before exiting the program.

**D.6 Bibliography**

APPENDIX D. SASS

[D.7 Appendix]

D.7.1 Database Format

The general format of all .db files are as follows:

- Fields are separate by comma
- Each line correspond to a record of fields. There must be equal number of fields in each record.
- String fields can have spaces.
- The first field is always the ID number of a subject.

The individual files have the following fields:

- **person.db**
  - ID number
  - Name
  - Gender
  - Stature
  - Mass
  - Handedness
  - Age
  - Description of data
  - Data source

- **strght.param.db**
  - ID number
  - Side (Left or Right)
  - Type of Strength
  - Fatigue Level
  - % Lean Mass

- **girth.db, cmass.db, jntlmt.db**
  - ID number
  - Segment Name
  - X value
  - Y value
  - Z value


D.7.2 Hot Keys

<table>
<thead>
<tr>
<th>Command</th>
<th>Speed-key(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Help</td>
<td>&lt;?&gt;</td>
</tr>
<tr>
<td>Quit</td>
<td>&lt;Q&gt;</td>
</tr>
<tr>
<td>Select Contour</td>
<td>&lt;n&gt;</td>
</tr>
<tr>
<td>Select Polybody</td>
<td>&lt;p&gt;</td>
</tr>
<tr>
<td>Select Object Mode</td>
<td>&lt;o&gt;</td>
</tr>
<tr>
<td>Select Segment Mode</td>
<td>&lt;s&gt;</td>
</tr>
<tr>
<td>Select girth</td>
<td>&lt;g&gt;</td>
</tr>
<tr>
<td>Select jutlmt</td>
<td>&lt;j&gt;</td>
</tr>
<tr>
<td>Select center of mass</td>
<td>&lt;c&gt;</td>
</tr>
<tr>
<td>Select strength</td>
<td>&lt;t&gt;</td>
</tr>
<tr>
<td>Select male</td>
<td>&lt;m&gt;</td>
</tr>
<tr>
<td>Select female</td>
<td>&lt;f&gt;</td>
</tr>
<tr>
<td>Dynamic Stature Model</td>
<td>&lt;dv&gt;</td>
</tr>
<tr>
<td>Static Stature Model</td>
<td>&lt;ds&gt;</td>
</tr>
<tr>
<td>Database</td>
<td>&lt;D&gt;</td>
</tr>
<tr>
<td>Write to Jack</td>
<td>&lt;w&gt;</td>
</tr>
<tr>
<td>Repeated write to Jack</td>
<td>&lt;r&gt;</td>
</tr>
<tr>
<td>Create figure</td>
<td>&lt;b&gt;</td>
</tr>
<tr>
<td>Create scaling</td>
<td>&lt;k&gt;</td>
</tr>
<tr>
<td>Save contents</td>
<td>&lt;S&gt;</td>
</tr>
<tr>
<td>Read file</td>
<td>&lt;R&gt;</td>
</tr>
<tr>
<td>Local clean up</td>
<td>&lt;L&gt;</td>
</tr>
<tr>
<td>Global clean up</td>
<td>&lt;G&gt;</td>
</tr>
<tr>
<td>Strength Subject</td>
<td>&lt;V&gt;</td>
</tr>
<tr>
<td>Toggle Port</td>
<td>&lt;X&gt;</td>
</tr>
</tbody>
</table>

D.7.3 Pull-down menu

<table>
<thead>
<tr>
<th>Command</th>
<th>Menu box</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global clean up</td>
<td>Misc</td>
</tr>
<tr>
<td>Local clean up</td>
<td>Misc</td>
</tr>
<tr>
<td>Help</td>
<td>Misc</td>
</tr>
<tr>
<td>Quit</td>
<td>Misc</td>
</tr>
<tr>
<td>Select Contour</td>
<td>Figure</td>
</tr>
<tr>
<td>Select Polybody</td>
<td>Figure</td>
</tr>
<tr>
<td>Select male</td>
<td>Figure</td>
</tr>
<tr>
<td>Select female</td>
<td>Figure</td>
</tr>
<tr>
<td>Select Object Mode</td>
<td>Modes</td>
</tr>
<tr>
<td>Select Segment Mode</td>
<td>Modes</td>
</tr>
<tr>
<td>Dynamic Stature Model</td>
<td>Modes</td>
</tr>
<tr>
<td>Static Stature Model</td>
<td>Modes</td>
</tr>
<tr>
<td>Create figure</td>
<td>Creation</td>
</tr>
<tr>
<td>Create scaling</td>
<td>Creation</td>
</tr>
<tr>
<td>Save sheet contents</td>
<td>Creation</td>
</tr>
<tr>
<td>Read *.sas file</td>
<td>Creation</td>
</tr>
</tbody>
</table>
D.7.4 Contents by Topic

General information

<table>
<thead>
<tr>
<th>Command</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Select</td>
<td>D.4.1.3</td>
<td>D.4.1.7</td>
</tr>
<tr>
<td>Dialog Box</td>
<td>D.4.1.8</td>
<td>D.4.1.6</td>
</tr>
<tr>
<td>Help</td>
<td>D.4.1.12</td>
<td>D.2.3</td>
</tr>
<tr>
<td>Local clean up</td>
<td>D.4.1.16</td>
<td>D.4.1.6</td>
</tr>
<tr>
<td>Mouse Select</td>
<td>D.4.1.11</td>
<td>D.4.5</td>
</tr>
<tr>
<td>Read file</td>
<td>D.4.1.14</td>
<td>D.4.1.5</td>
</tr>
<tr>
<td>Scroll</td>
<td>D.4.1.10</td>
<td></td>
</tr>
<tr>
<td>Tricks</td>
<td>D.5</td>
<td>D.4.1.4</td>
</tr>
</tbody>
</table>

Anthropometry Stuff

<table>
<thead>
<tr>
<th>Command</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anthropometry Sheet</td>
<td>D.4.1</td>
<td>D.4.1.4</td>
</tr>
<tr>
<td>Data Section</td>
<td>D.4.1.3</td>
<td>D.4.1.2</td>
</tr>
<tr>
<td>Summary Section</td>
<td>D.4.4.1</td>
<td></td>
</tr>
</tbody>
</table>

Group Stuff

<table>
<thead>
<tr>
<th>Command</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Groups</td>
<td>D.4.1.1</td>
<td>D.4.1.2</td>
</tr>
<tr>
<td>Group Status</td>
<td>D.4.1.1</td>
<td></td>
</tr>
</tbody>
</table>

Query Sheet Stuff

<table>
<thead>
<tr>
<th>Command</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query</td>
<td>D.4.4</td>
<td>D.4.4.4</td>
</tr>
<tr>
<td>Query Section</td>
<td>D.4.4.2</td>
<td></td>
</tr>
</tbody>
</table>

File Stuff

<table>
<thead>
<tr>
<th>Command</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure File</td>
<td>D.4.1.4</td>
<td>D.4.3.1</td>
</tr>
</tbody>
</table>

Stature Stuff

<table>
<thead>
<tr>
<th>Command</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Select</td>
<td>D.4.1.13</td>
<td>D.4.6.3</td>
</tr>
</tbody>
</table>

Strength Stuff

<table>
<thead>
<tr>
<th>Command</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strength sheet</td>
<td>D.4.2</td>
<td></td>
</tr>
</tbody>
</table>

Port Stuff

<table>
<thead>
<tr>
<th>Command</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SASS-JACK</td>
<td>D.4.3</td>
<td></td>
</tr>
</tbody>
</table>
Appendix E

CAD Conversion Programs

This chapter describes the external utilities in Jack for converting Jack geometry to and from popular CAD package formats.

E.1 Guide to CAD Geometry Conversion

In order to get the most out of Jack, it is vital to get geometry that has been designed in external CAD systems into the Jack environment. This basic need has also traditionally been a frustrating and difficult procedure. This guide will inform you of some of the general principles that will make this transfer of data as smooth as possible.

Three steps are required to bring data from a CAD system into Jack. First, you must create the geometry (or alter existing geometry) within your CAD package to be as "expressive" as possible. Then, translate the data using the appropriate CAD format conversion utility. Finally, read the translated data into Jack and fully define the objects. The rest of this document will describe each of these steps in detail.

E.1.1 Object Design Within the CAD System

Jack models the world in a hierarchical manner. The environment is made up of figures, which are represented by a tree of segments connected by joints.1 Each conversion utility program attempts to extract as much relational information as possible from the CAD database. In order for this process to succeed, you must try to model your geometry in this object-oriented fashion.

Facilities for grouping drawing primitives differ from system to system. However, most CAD packages provide at least a rudimentary method for expressing primitive associations. For example, AutoCAD allows the user to define layers in the drawing. The dxf2pss utility uses these layers to decide which segment each primitive should be assigned to. By creating your designs with this in mind you can easily articulate your objects once they have been read into Jack. As another example, Pro/Engineer allows you to define assemblies and parts. This information will be used in a similar manner by pro2fig. These primitive grouping facilities will be referred to as groups for the rest of this document to express this idea in a manner independent of any particular system.

Each geometric entity should be placed in its own group. In this context, the term "geometric entity" means an object or part of an object that is completely atomic: it can not be articulated or altered in any manner. These geometric entities correspond exactly to segments in Jack. As an example, suppose you wanted to model a door in a frame. The polygons that make up the frame would be placed in one group, and the polygons comprising the door would be placed in another group. These two groups would then be translated into separate segments in Jack, and it would then be possible to create a joint between them to accurately model the door/frame assembly.

This grouping must be done at a fine level. If you wanted to model a building with several rooms, for example, each door should be placed in its own group (door1, door2, etc.). This is necessary if you wish

1See Section 3.1
to articulate any of the doors. Otherwise, all the doors would be placed in one segment and would be fixed relative to each other. It would be impossible to create a door/frame joint as described above.

Object complexity is another factor that must be considered when translating geometry for use in Jack. Any object that is going to be displayed in an interactive environment should be as simple as possible. Many CAD systems produce highly faceted objects that make interaction with the environment in Jack nearly impossible. Keep in mind that every object in the environment must be completely redrawn every time the screen is updated.

It is sometimes possible to alter the complexity of an object after it has been designed. For example, when writing a render file from Pro/Engineer, you must specify a "quality factor". This determines how finely surfaces are faceted during the procedure. Similarly, it is often possible to change the resolution parameters of surface patches in other CAD systems.

One technique for dealing with the graphics speed limitations imposed by complex geometry is to design two versions of an object: one for use in interaction, and another for use when a scene or animation is being rendered. If the two representations are defined in the same coordinate system they can be easily switched in Jack.

The amount of information that can be translated into Jack varies with the CAD system and the data format being used. Check the documentation for the translator you are using for more specific information.

E.1.2 Translating the Data

Using the Jack geometry translators is generally straightforward. There are limitations to each format and each translator though, and it is important to check the documentation for the translator you are using.

These limitations usually involve the types of entities that are supported by the translator. Native CAD formats are constantly changing, and many times a conversion utility will lag behind. In addition, many formats, such as IGES, have a very large number of entity types, and CAD packages use different subsets of these entities to express similar information.

When a translator encounters an unsupported entity, it will usually print an error message and continue, ignoring the information. When this happens, report the error message to the University of Pennsylvania Computer Graphics Research Laboratory, then try to coerce your CAD system into writing a different entity type. This can be difficult and is sometimes impossible, but it is your only recourse. Methods for doing this vary greatly. For example, if your model is defined with a polygonal mesh, and the conversion utility you are using does not support the format's polygonal mesh primitive, it may be possible to explode the mesh into discrete faces within the CAD system. It may then use the polygon primitive instead.

E.1.3 Grouping and Articulating Your Geometry in Jack

Once you have successfully read your geometry into Jack it is a simple process to build fully articulated objects. This section will provide a basic overview to this process; for more detailed information see the appropriate sections in this User's Guide.

If you have followed the recommended procedure for object grouping, each "movable part" in your environment will be represented as a segment within Jack. If this procedure has not been followed, it will probably be necessary to split the segments that were produced by the translator. Check the User's Guide for the various commands that can be used to split segments.

Initially, each segment will be considered an independent figure. In order to create more complex articulated figures, joints must be created between segments. The first step in creating a joint is to define sites on the two segments that will be joined. These sites are reference points on the segment that determine the "connection points" for the new joint. Once the sites have been created, a joint can be defined, including the degrees of freedom of the joint and the joint limits. For example, to articulate the door/frame assembly mentioned above, sites could be defined on the door and the frame that correspond to the points of connection for one of the hinges. A rotational joint could then be created between these two sites with the appropriate limits. This would completely define the door/frame as one figure, composed of 2 segments and one joint.

Other steps that may be needed to complete the translation are scaling and color/surface attribute definition. Jack assumes that all dimensions given in its data format are in centimeters. Some conversion utilities allow you to specify a scaling factor, but it may be necessary to do it by hand inside Jack. Also,
E.2 SDRC Geomod Universal \(\rightarrow\) Jack

To convert from SDRC IDEAS-Geomod Universal Files to Jack's peabody files, use \texttt{geopea}. It takes a command line argument of the name of the universal file to convert:

\begin{verbatim}
% geopea shaker2.unv
\end{verbatim}

It will create several files in the current directory. It will create a psurf file corresponding to each object, and a single peabody environment file describing the entire assembly. The name of the environment file is taken from the name of the universal file, with the suffix ".env" in place of ".unv". The names of the psurf files are derived from the names of the objects. Geopea prints messages describing the files it creates.

Sometimes, \texttt{geopea} seems to not work at all, although it still produces some useful output. If \texttt{geopea} crashes, or the peabody file that it produces cannot be read into \textit{Jack}, or it reads in but places the objects in vastly incorrect locations relative to each other, try reading all the psurfs into \textit{Jack} individually. This is easier than it sounds: just go to the directory in which you ran \texttt{geopea} and type:

\begin{verbatim}
% jack *.pss
\end{verbatim}

Each psurf will be read in as a separate figure. You can then merge the figures together using the \texttt{make environment into figure} command, or the \texttt{write environment to psurf} command.

E.3 AutoCAD DXF \(\rightarrow\) Jack

dxf2pss is a utility for converting AutoCAD DXF format files to peabody files readable by \textit{Jack}. Each layer in the DXF file will be converted to a Jack psurf. Each block entity will also be placed in an individual psurf. These segments will be linked together in a figure file. Each segment in the figure will be assigned a random color. Since DXF objects are not articulated, the converter creates a joint between each segment and an invisible base segment. Thus, this joint should be deleted before creating new joints to articulate the figure.

dxf2pss handles the majority of DXF entities, but it is not yet exhaustive. Supported entities include: blocks and insertions, 3DFaces, 2D and 3D polylines, circles, arcs, and extrusions of each of these. Polygon meshes, dimensions, and text are currently not supported and will be ignored.

Usage is as follows:

\begin{verbatim}
% dxf2pss <filename>.drf
\end{verbatim}

The output figure file will be <filename>.fig. Individual psurf files will be <layer name | block name>.pss.

E.4 (Simple) IGES \(\rightarrow\) Jack

The program iges2pss converter translates simple IGES files directly into psurfs. iges2pss is actually a shell script that calls the programs igessplit, igestype, igesmat, and igesa3i. Usage is:

\begin{verbatim}
% iges2pss <infile>.iges <outfile>.pss [scale factor]
\end{verbatim}

The script first uses igessplit to split the input file into several sub-files to minimize memory requirements. It then calls igestype to list the entity types that are used in the file. igesmat processes the necessary transformation matrices, and igesa3i creates the psurf file.

\textit{Jack} assumes that segment dimensions are in centimeters. If your files use a different unit, use the scale factor argument to compensate.

iges2pss creates .dat files while performing the conversion. These files should be deleted after conversion is complete.
This utility converts an IGES file into one psurf. Therefore, each separate object, and each segment of an articulated object, should be written to a separate IGES file. These can then be reconstructed in Jack.

This translator is not exhaustive. It will produce warning messages when an unsupported entity is encountered in the IGES file.

This translator was written by Scott Chen, A3I MIDAS Project, NASA Ames Research Center.

E.5 Pro/Engineer Render File $\implies$ Jack

To convert from Pro/Engineer Render format to peabody figure and psurf files, we use pro2fig. Usage is as follows:

```
% pro2fig <render file>
```

pro2fig takes a Pro/Engineer render file as input and produces a psurf (.pss) file for each part and a figure (.fig) file for the entire object. Each psurf file is named by appending a .pss to the part name; the figure file name is based on the render file name with its extension replaced with .fig.

Render files may be created in either Assembly or Part mode. In Assembly, any number of parts may be selected for inclusion into the Render file. The density of surface facets may be controlled by setting the "quality factor" in Pro/Engineer.

The resulting figure file defines a joint between each part and a "base" segment. Information concerning joints between parts in an assembly is not included in the Render file.

Vertex coordinates are assumed to be in centimeters by Jack.

E.6 Jack $\implies$ RenderMan RIB File

To create a RenderMan RIB file of a Jack environment, we use the program convert.

```
% convert <jackfile.env> [horizontal] [vertical] [zoom]
```

The parameters are:

- **jackfile.env** is any valid Jack environment file,
- **horizontal** is the horizontal resolution in pixels of the RenderMan image,
- **vertical** is the vertical resolution in pixels of the image, and
- **zoom** is a real number greater than zero where
  - zoom > 1.0 will zoom in on the scene,
  - zoom < 1.0 will zoom away from the scene, and
  - zoom = 1.0 will leave the camera where it is.

The only required entry is <jackfile.env>. Horizontal, Vertical and Zoom default to 400, 300 and 1.0 respectively.

The output of the program is directed to standard out. In this way the output can be redirected into a .rib file by piped directly into the renderer (usually /usr/local/prman/bin/render). Note that if a .rib file is created then the appropriate shaders must be in the same directory as the .rib file for the image to look proper.

Notes on convert:

- Jack does not write out attribute information for cad objects. For this reason cad objects may appear different colors when they are converted to RenderMan.
- The following capabilities in Jack are not implemented in the conversion program:
  - fractals
E.7 WAVEFRONT OBJ FILE ↔ JACK

- texture or bump maps
- curves and patches
- fresnel

E.7 Wavefront OBJ File ↔ Jack

obj2fig is used to convert Wavefront .obj files to Jack's peabody language. Usage is:

where file.obj is the Wavefront file to be converted, basename is the figure file name base (i.e. no .fig extension), and scale-factor is a conversion to be applied to all vertex coordinates.

Each group in the obj file is written to a separate psurf in the resulting figure. These segments can then be connected through joints to correctly articulate the figure.

E.8 (Complete) IGES File ↔ Jack

ij is an interactive viewer and translator for converting files between the IGES and Jack peabody formats. ij is IGES-based in that user-interaction during translation is applied to IGES-level entities, no matter the original source of the input file.

ij supports most 3-D geometry-related entities; labelling and annotation entities are not supported. ij also has full support for peabody structures, including environments, figures, and psurfs. Color information is also preserved.

ij provides grouping and subfigure support for hierarchical preservation of peabody segments in figure definitions. Joint information is preserved wherever possible; however information is frequently lost due to inherent limitations of the IGES format.

ij maintains several input and output options for compatibility with existing packages which import and export IGES such as AutoCAD, Catia, Pro/Engineer, and Unigraph II. Due to the overwhelming scope and ambiguous nature of IGES, varying degrees of IGES support are maintained by these packages. ij permits output on varying structural levels to compensate for these discrepancies.

The translator also provides a 3-D viewer front-end as well as many WYSIWYG control features for user-manipulation of the translation process. These include setting of tessellation parameters for converting geometry-based entities such as Spline Patches to polygonal-based Jack psurfs.

E.8.1 Conversion

Please see Section 3.1 for definitions of peabody terms.

An example of a peabody environment composed of a human figure and a chair is below. Both the human and chair are figures. The chair figure contains only one segment, whose entire geometry is defined in one psurf file "chair.pss". The human consists of many connected segments, such as "left_upper_arm", "left_lower_arm", "left_palm", etc. Different segments can access their geometry from the same psurf file, such as "left_eye" and "right_eye" both accessing "eye.pss".

"left_lower_arm" has a site at the top of the segment; "left_upper_arm" has a site at the bottom of itself. A rotational joint connects these two sites, thereby connecting the two segments as in an actual arm. "left_lower_arm" has another site defined at its bottom to which the palm is connected. It is this hierarchy which allows the palm to move with the lower arm as the shoulder is moved.
A simplified body hierarchy is:

```
--------
| head |
|--c--  |
-------------------
| right_arm | c-p | p-c | left_arm |
-------------------
| body |-------------------
|     |
|--p--c--
|     |
```  
right_upper_leg | right_lower_leg  
| | |  
| | |  
| p- | c-  
| |  
| c- | p-  
| |  
right_lower_leg | left_lower_leg  
| | |  
--- -R-

where R is the rooting site of the figure, p is a parent site (site 1) and c is a child site (site 2). An excerpt of this environment file is:

```plaintext
figure {
  ...
  attribute pants {
    rgb = (0.20, 0.20, 1);
    ambient = 0.10;
    diffuse = 0.90;
  }
  ...
  segment right_lower_leg {
    psurf = "r1leg.pss" * scale(5.35, 5.35, 34.53);
    attribute = pants;
    site proximal->location = trans(0.00cm, 0.00cm, 0.00cm);
    site distal->location = trans(0.00cm, 0.00cm, 34.53cm);
  }
  segment right_upper_leg {
    psurf = "ruleg.pss" * scale(7.13, 7.13, 40.72);
    attribute = pants;
    site proximal->location = trans(0.00cm, 0.00cm, 0.00cm);
    site distal->location = trans(0.00cm, 0.00cm, 40.72cm);
  }
  ...
  joint right_knee {
    connect right_upper_leg.distal to right_lower_leg.proximal;
    type = R(-y);
    llimit = (0.00deg);
    ulimit = (145.60deg);
  }
```
E.8. (COMPLETE) IGES FILE $\leftrightarrow$ JACK

rest = (10.00deg);
tolerance = (10.00deg);
}
...
}

E.8.2 Jack $\implies$ IGES

ij tries to preserve as much information as possible. To do this, each segment is translated as a group entity and transformation entities preserve joint descriptions. Attribute colors are translated into color definition entities. The formal correlation of Jack to IGES entities is detailed below:

Figures are traversed according to their rooting hierarchy. Each segment’s geometry is translated into a GROUP (402) entity which contains all of the segments faces. Each face of the segment is translated as a PLANE (108) entity bounded by a COMPOSITE CURVE (102) entity. The curve consists of LINE (110) entities, each line representing an edge of the face.

The joint information is expressed through associating a SUBFIGURE DEFINITION (308) with each parent segment. The SUBFIGURE contains the parent segment’s psurf in a GROUP entity (402) and the SUBFIGUREs (308s) of all of the child segments. If NO GROUP is specified in the OPTIONS menu, the SUBFIGURE contains the segment faces directly instead of containing the GROUP. This option is available for systems not allowing groups, such as Pro-Engineer. The joint information is maintained in the associated TRANSFORMATION MATRIX (124) entity in the Directory Entry. Color information is maintained for each face by the associated COLOR DEFINITION (314) in the Directory Entry for each PLANE (108). The figure itself is accessed through a SINGULAR SUBFIGURE INSTANCE (408) of the root segment’s corresponding SUBFIGURE DEFINITION (308), with the figure’s location preserved through a TRANSFORMATION MATRIX (124) entity.

In the human figure example, an excerpt of the IGES structure with correlating Jack structures in parentheses is:

```
  \\ SUBFIG 308 ------- [ TRANSFORMATION MATRIX (124)]
  (segment right_upper_leg) (joint right_leg)
  /
GROUP 402  SUBFIG 308 ------- [TRANSFORMATION (124)]
  (psurf right_upper_leg) (segment right_lower_leg) (joint right_knee)
  /
GROUP 402
  (psurf right_lower_leg)
```

Each group is further broken down as:

GROUP 402
  \ /    \ \ 
PLANES (108s)

and each plane further contains a bounding curve:

```
  PLANE (108) ------- [ COLOR DEFINITION (314)]
  | COMPOSITE CURVE (102)
  / /    \ \ 
LINES (110s)
```

E.8.3 IGES $\implies$ Jack

ij assumes grouping information as above for translation of IGES files. When this information is not present, it tries to preserve the natural hierarchy as much as possible. Any grouped entities will be written to the
same segment, and any transformation entities will generate joints. Color definition whether in the Directory Entry or as expressed in a separate entity will generate attributes. Un-grouped entities will be written as one entire segment.

More specifically, non-referenced (or top) 408s or 402s will create a figure definition and a set of segments jointed together. Non-referenced 308s will create an error report and will be ignored. Those geometry entities referenced by the 308 will be viewed as non-referenced primitive geometry entities. All the non-referenced primitive geometry entities will be combined to one figure containing one segment. There will be a wrong figure if there is a top 308 associated with a non-identity transformation.

Starting from the non-referenced 408 or 402, a Jack peabody tree is constructed through the 308s, 408s and 402s that can be referenced through the top 408 or 402. A new segment is created whenever a 308, a 408 or a 402 is met. The joint transformation is the transformation entity or identity transformation associated with the 308, 408 or 402. All primitive geometry entities referenced by the same 402 or 308 will be written into one psurf file.

Colors associated with the geometry entities are preserved in the corresponding psurf as the attribute. Because of the inborn redundancy in IGES files (i.e. common nodes are duplicated, etc.), ij tries to eliminate these redundancies. Common color information is combined into a single attribute, and generated psurf files are automatically piped through two post-processing programs to eliminate redundant nodes and incorrect faces. These two programs are pnr and pfr (see associated documentation).

All entities are reduced to polygonal faces; the tessellation is controllable through the interactive user interface (see Parameters section of User Interface).

E.8.4 User Interface

This is the menu hierarchy of the ij program. It is a Motif program, and the resource file is called IGES.

File Load Read in an IGES or Jack environment file
Quit Quit ij

Options Jack Read Options Read Segments Globally sets whether to associate transformation entities or to create geometry entities already in their global positions. Use this option when importing into packages which don't support transformation entities (134) such as AutoCad. or grouping such as ProEngineer.

Exact Attributes sets whether or not segments which share a psurf but have different colors should create separate geometry entities or share the same. When set, only one set of entities is created and both segments will have the same color. When not set, two sets of entities containing different colors will be created.

No Segment Group (402) sets whether or not to create a GROUP entity (402) for each segment or whether the faces are contained directly in the SUBFIGURE DEFINITION (308). NOTE: If selected, the Exact Attributes is automatically chosen and setting or un-setting has no effect.

Processing Triangulate Triangulate a mesh
Closed
Points Toggle points on/off
Edges Toggle edges on/off
Surfaces Toggle surfaces on/off
Entities Toggle individual entities on/off
Precision Set the floating point comparison precision
Basis Set the basis function for all splines
Auto Smooth Automatically smooth a surface

Parameters Arc Increment Set the arc increment of curve-based entities
Tolerance Zero  Fuzz for zero comparison
    Plane  Fuzz for Planar comparison
    Point  Fuzz for Point comparison

P(u)  Set the number of divisions in the u direction
P(v)  Set the number of divisions in the v direction
Flip  Reverse the face normals
Active  Turns on entity
Inactive  Turns off entity

Smoothing  Smoothing affects tessellation of curves and patches. Tessellation is accomplished by
starting at the lowest number of divisions and checking the angular discrepancy to determine
whether one more reduction is necessary. Tolerance is the maximum allowed angular discrepancy
and iterations is the maximum reduction allowed.

Tolerance  Set the smoothing tolerance for tess-ellating patches
Iterations  Set the maximum number of iterations in calculating patch tesselations (MAX_TRIES
in ij)

Viewing  Eye Point  Rotate/Zoom view
    View Reference Point  Pan view
    Center  Center the view on the currently displayed entities
    Scale  Scale the view

Rendering  Fill  Toggle shading on/off
    Axes  Toggle axes on/off
    Labels  Toggle labels on/off
    Z-Average  Toggle Z-Averaging for hidden surface removal on/off

Color  IGES Defined Colors  Use IGES defined colors
    Representation-based Colors  Use colors to represent entity types
    Color Palette  Set the color palette
    Redraw  Redraw the scene

Output  Env-Joint  Write environment file maintaining all displacements in the joint definition
    Env-Site  Write environment file maintaining all displacements in the parent site.
    IGES Native  Write as native IGES entities
    IGES Mesh  Write everything as IGES meshes
        Simple Closed
        Plane
        Trimmed Parametric

Statistics  Statistics on environment
E.8.5 Supported Iges Entities

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Undefined</td>
</tr>
<tr>
<td>100</td>
<td>Circular Arc</td>
</tr>
<tr>
<td>102</td>
<td>Composite Curve</td>
</tr>
<tr>
<td>104</td>
<td>Conic Arc</td>
</tr>
<tr>
<td>106</td>
<td>Copious Data</td>
</tr>
<tr>
<td>108</td>
<td>Plane</td>
</tr>
<tr>
<td>110</td>
<td>Line</td>
</tr>
<tr>
<td>112</td>
<td>Parametric Spline Curve</td>
</tr>
<tr>
<td>114</td>
<td>Parametric Spline Surface</td>
</tr>
<tr>
<td>116</td>
<td>Point</td>
</tr>
<tr>
<td>118</td>
<td>Ruled Surface</td>
</tr>
<tr>
<td>120</td>
<td>Surface of Revolution</td>
</tr>
<tr>
<td>122</td>
<td>Tabulated Cylinder</td>
</tr>
<tr>
<td>124</td>
<td>Transformation Matrix</td>
</tr>
<tr>
<td>126</td>
<td>Rational B-Spline Curve</td>
</tr>
<tr>
<td>128</td>
<td>Rational B-Spline Surface</td>
</tr>
<tr>
<td>134</td>
<td>Node</td>
</tr>
<tr>
<td>136</td>
<td>Finite Element</td>
</tr>
<tr>
<td>142</td>
<td>Curve On Parametric Surface</td>
</tr>
<tr>
<td>144</td>
<td>Trimmed Parametric Surface</td>
</tr>
<tr>
<td>308</td>
<td>Subfigure Definition</td>
</tr>
<tr>
<td>314</td>
<td>Color Definition</td>
</tr>
<tr>
<td>402</td>
<td>Associativity Instance</td>
</tr>
<tr>
<td>408</td>
<td>Singular Subfigure Instance</td>
</tr>
<tr>
<td>412</td>
<td>Rectangular Array Subfigure</td>
</tr>
<tr>
<td>414</td>
<td>Circular Array Subfigure</td>
</tr>
</tbody>
</table>

E.9 ComputerVision $\Rightarrow$ Jack

E.9.1 Introduction

cv2jack is a program that translates geometry from a CADDS 4X version 6.2 system to the format used for Jack. In order to convert geometry from CADDS 4X to Jack the following is done:

1. The user generates ASCII geometry files of parts on the Computervision system using commands in the CADDS package.
2. Those files and procedure files are transferred to the computer system where Jack resides.
3. The geometry translator is then invoked to create a Jack psurf file or a Jack environment file.
4. When the entire process is complete, there will be a directory containing psurf and environment files.

E.9.2 Computervision Basics

The CADDS 4X tool generates parts which are a collection of entities. Each entity may be 3D geometry, 2D figures, display information, or instance information. Nodal figures (nfigs) and subfigures (sfigs) are CADDS entities that are used for instancing other parts in a current part. For example, to create a room with a table and chairs: A set of walls, a floor, a ceiling, a table, and a chair are all created and each saved as an nfig part. Each nfig is built in x, y, and z coordinates and has its own origin point about which all the geometry for the part was built. Next, a new part is opened and the CADDS command insert nfig is used to insert the walls, floor, ceiling, and table parts in the new part. An insertion point for each nfig is either typed in or the cursor is used to choose a location in the part for each. The insertion point is where the origin for
E.9. COMPUTERVISION $\rightarrow$ JACK

each nfig is placed in the part. The chair is inserted several times in order to instance several chairs about the table. From this example the top level part (the part that had nfigs inserted in it) would be converted to a Jack environment file and each of the nfig parts would become psurf files.

Nfigs may also be used to insert procedure file shapes in a part. A procedure file is a CADDS file that contains text with shape keywords and values for parameters of each shape in the file. An example is the shape cylinder (CYL), which has parameter values for length and diameter. Scale and variable information are set in the nfig properties. When scale is omitted the scale is 1.0, 1.0, 1.0 for the x, y and z scale values respectively. The piping and ducting construction tools in CADDS automatically insert procedure file information during insertion of fittings. All of this information is converted and becomes part of the Jack psurf file.

E.9.3 What is needed to convert?

The first item needed is for the cv2jack conversion tool to be loaded on the machine on which you want to perform the conversion. It will only run on an SGI workstation.

A top level part in the CADDS data base that contains only nfig or sfig instances is needed to generate a Jack environment file. Procedure files may be instanced in an nfig of a top level part. If the procedure file has Nodal Line following parts, then Nodal Lines in the part are used to convert the procedure file. If a top level part does not exist then the part that needs conversion may be saved as an nfig. A new part could then be generated that inserts this one nfig. In this case, the environment file would have only one psurf instanced and the psurf would include the geometry of the entire part. When possible the top level part should be made up of many nfig or sfig instances so that each of the psurf files will be smaller. When the top level file has entities other than nfigs, sfigs, or Nodal Lines, they will be ignored.

In order to run cv2jack a specific directory structure is required as shown in Figure 1. This enables the converter to locate files needed for the conversion process. The base directory in most cases will be the home directory of the user. This directory contains the individual topic directories and a lib directory for shared psurf files. It is required that cv2jack be run in a topic directory. For each top level part name that is converted, a topic directory of the same name should be created in the base directory. Several required directories are needed in each topic directory (see Figure 1). These directories are pconvert, econvert, procedure, and jackdata. The pconvert directory is used to hold CADDS data files that will be converted to psurfs. The econvert directory is used to hold CADDS data files that will be converted to environment files. The procedure directory holds CADDS procedure text files. The jackdata directory stores environment files and psurf files after they have been converted. The only psurf files that are created in the jackdata directory are from procedure files that are instanced in the top level part.

The user must enter a command line option to generate an environment file, otherwise the converter will expect the CADDS data file to contain data that will be converted to a psurf. The environment file is written as “filename.env” in directory jackdata where filename is the name of the data file transferred from the CADDS system. Also written to the directory are psurf files for each of the procedure file instances that were included in the top level part. All instances of a procedure file will be grouped in the same psurf with a name such as “partname.pss7” where partname is the procedure file part name. All nfigs and sfigs that are not procedure files will not generate a psurf file but will generate a record in the environment file. These parts are either already psurf files that exist in the lib directory, or need to be converted to psurf files in a separate conversion operation. When psurfs are not found a file named missing.psurf in the current topic directory will be created with each needed part name listed.

To break up procedure file psurf files that are inserted in a top level part, the procedure file should be saved in a part and saved as an nfig. This new part would be inserted in the top level part, not the procedure file directly. For example a procedure file of a chair inserted several times directly in a top level part will create one record in the environment file and create a psurf file with several chairs in it. The same chair first saved as an nfig part and then inserted several times in the top level part will create several records in the environment file. Each will reference the same psurf file that has the chair geometry in it. Psurf parts are created in the jackdata directory for all parts converted that are not top level parts.
E.9.4 Generating a Jack Environment File

A Jack environment file consists of instances of psurfs. A CADDS 4X part file that is composed of nfig and sfig instances may be converted to an environment file. When using the environment file option of the converter only nfig entities, sfig entities, and associated nlines of the CADDS part are considered, all others are ignored. The environment file option of the converter is used to generate a Jack environment file. The conversion process is as follows:

1. Using the CADDS 4X command, OPEN PART a.b.c.part1, open the part, where a.b.c.part1 is a CADDS 4X part name.

2. Use the CADDS 4X command, DO HARDFILE outname, to store all information listed on the screen to a file, where outname is the name of the file to store the information in (this file is known as the hardfile). The hardfile name outname can not have any “.” characters in it. CADDS 4X will convert “.” in a filename to a directory level. You would not get a.b.c.part1 in the .bcd directory. You would get a directory hierarchy with directory a in .bcd, directory b in a, directory c in b, and the file part1 in directory c.

3. Issue the CADDS 4X command, LIST PART STATUS, to dump to the screen (and the hardfile) part information including the units used. This command is optional and when not issued the units are assumed to be feet.

4. Issue the CADDS 4X command, DUMP ENTITY, to dump to the screen (and the hardfile) all the entity information for the part. This command must always be issued.

5. Use the CADDS 4X command, DO HARDFILE, to close the file. When closing the hardfile a file name is not needed on the command line. At this point a file named outname (or other user defined name) will reside in the .bcd directory that is under the CADDS 4X create directory for the user.

6. Open a unix shell to enter unix commands.

7. Rename the file to correspond to the CADDS 4X name such as

   mv outname a.b.c.part1

8. Copy the hardfile created over to the econvert directory under the topic directory that this file represents. The topic directory name and the hardfile name are almost always the same name.
E.9. COMPUTERVISION \rightarrow JACK

9. Run cv2jack on the hardfile with the environment file option (i.e., \texttt{cv2jack -e a.b.c.part1}).

10. Since cv2jack was executed using the environment file option, a file named \texttt{psurf.missing} is created in the current directory to store nfig names not found in the \texttt{jackdata} directory. This list tells the user which parts need to be generated into psurf files. In the environment file the psurfs that are not found will be commented out. Once these files are converted to psurfs the user may rerun the converter on the top level part or uncomment each instance of the psurfs created in the environment file.

11. On completion a file named \texttt{a.b.c.part1.env} will be generated in the \texttt{jackdata} directory. Also any procedure files in the environment file will generate psurf files named \texttt{psurfname.pss} in the \texttt{jackdata} directory. All environment file names created will convert all "." or "-" to ".". This is necessary to follow the format needed for Jack.

Summary of commands: (in a CADDS 4X session do the following:)

\begin{verbatim}
OPEN PART a.b.c.part1
DO HARDFILE outname
LIST PART STATUS
DUMP ENTITY
DO HARDFILE
\end{verbatim}

(in a unix shell do the following:)

\begin{verbatim}
mv outname a.b.c.part1
\end{verbatim}

(Copy \texttt{a.b.c.part1} to the \texttt{econvert} directory on the system with \texttt{cv2jack} using \texttt{rcp}, a tape, etc. Also change the current directory to the specific topic directory.)

\begin{verbatim}
cd topic.directory
\end{verbatim}

\begin{verbatim}
cv2jack -e a.b.c.part1
\end{verbatim}

E.9.5 Generating a Jack Psurf File

A Jack psurf file consists of geometry information in a local coordinate system. A CADDS 4X part file that is made of entities and procedure file data may be converted to a psurf file.

The conversion process is as follows:

1. Using the CADDS 4X command, \texttt{OPEN PART a.b.c.part1}, open the part, where \texttt{a.b.c.part1} is a CADDS 4X part name.

2. Use the CADDS 4X command, \texttt{DO HARDFILE outname}, to store all information listed on the screen to a file, where \texttt{outname} is the name of the file to store the information in (this file is known as the hardfile). The hardfile name \texttt{outname} can not have any "." characters in it. CADDS 4X will convert "." in a filename to a directory level. You would not get \texttt{a.b.c.part1} in the \texttt{.bcd} directory. You would get a directory hierarchy with directory \texttt{a} in \texttt{.bcd}, directory \texttt{b} in \texttt{a}, directory \texttt{c} in \texttt{b}, and the file \texttt{part1} in directory \texttt{c}.

3. Issue the CADDS 4X command, \texttt{LIST PART STATUS}, to dump to the screen (and the hardfile) part information including the units used. This command is optional and when not issued the units are assumed to be inches.

4. Issue the CADDS 4X command, \texttt{DUMP ENTITY}, to dump to the screen (and the hardfile) all the entity information for the part. This command must always be issued.

5. Use the CADDS 4X command, \texttt{DO HARDFILE}, to close the file. When closing the hardfile a filename is not needed on the command line. At this point a file named \texttt{filename} will reside in the \texttt{.bcd} directory that is under the CADDS 4X create directory for the user.

6. Open a unix shell to enter unix commands.

7. Rename the file to correspond to the CADDS 4X name such as

\begin{verbatim}
mv outname a.b.c.part1
\end{verbatim}
8. Copy the hardfile created over to the pconvert directory on the system with the cv2jack conversion tool. This directory is under the topic directory with the top level part name that this part is instanced in.

9. Run cv2jack on the hardfile with the default psurf file option.

10. Since cv2jack was executed using the psurf file option, a file named nfig.missing is created at the current directory to store nfig or sfig names not found in the pconvert directory. These nfig and sfig instances represent a sub part. Each of these instances is itself a CADDS part. This list tells the user which sub parts need to be copied to the pconvert directory to fully convert the current part. When sub parts are not found then the sub part is skipped during the conversion of the current part. When the sub parts are copied over you must rerun the converter on the original part.

11. On completion a file named a.b.c.part1.pss will be generated in the lib directory.

Summary of commands: (in a CADDS 4X session do the following:)
OPEN PART a.b.c.part1
DO HARDFILE outname
LIST PART STATUS
DUMP ENTITY
DO HARDFILE
(in a unix shell do the following:)
mv outname a.b.c.part1
(Copy a.b.c.part1 to the pconvert directory on the system with cv2jack using rcp, a tape, etc. Also change the current directory to the specific topic directory.)
cd topic._directory
cv2jack a.b.c.part1

E.9.6 Executing cv2jack
To execute the cv2jack program you type the following:
   cv2jack [-d level] [-e] [-h] filename

   where

   d - sets the detail level from 1 to 5 (default 3). The detail level determines the smoothness of shapes and surfaces that will be generated, where 1 is least smooth (less polygons used to approximate shape) and 5 is smoothest.
   e - causes an environment file to be created, otherwise a psurf file is created.
   h - print a help message and do not run program.
   filename - name of a file in the econvert or pconvert directory that is a hardfile from CADDS 4X.

Example 1:
   cv2jack -e room

After conversion the jackdata directory will have a file named room.env that is an environment file. If any procedure files were instanced in this part then a psurf file for each would be created in the jackdata directory.

Example 2:
   cv2jack chair

After conversion the lib directory at the base will have a file named chair.pss that is a psurf file.

E.9.7 Limitations
There are several CADDS 2D entities that are used when generating surfaces. These entities are arcs, conics, and splines. In the CADDS data base these entities may be rotated out of the standard construction planes. The standard construction planes are top, bottom, right, left, front, and back. When a 3D surface uses any of these entities in its definition and they are not in one of the standard planes then the 3D geometry will be skipped. The rotations outside of the standard planes are stored in the CADDS data base as a matrix
(known as a TAB4 matrix). This matrix is not available outside of the CADDs environment in the current CADDs 4X version. The orientation is then unknown and can not be converted. Any geometry skipped due to this problem will not affect any other geometry that is being converted.

It is possible for the converter to create psurf files that are too large to be handled by Jack. When this happens the user should divide the psurf file into two or more psurf files using an editor. The environment file should also be modified to include the new psurfs and remove the old psurf instance.
Appendix F

The LISP Interface

F.1 Overview

The Lisp interface\(^1\) to Jack allows general programming of Jack internals and should simplify all forms of Jack development. It is distributed with Jack-5.8 and allows users without source code to extend or modify Jack.

The incorporation of Lisp into Jack can make development easier for a variety of reasons:

- Using an interpreted language avoids having to compile code — the Jack compile and link cycle can take up to 5 minutes. Changing or adding lisp code to a running Jack process can be done almost instantaneously and without exiting Jack.

- Lisp is a much higher level language than C or C++. It supports symbol and list processing, higher-order functions, lexical closures, and a variety of other advanced programming techniques. Data structures are also very easy to use in lisp — lisp manages memory automatically and has built-in lists, vectors, and matrices.

- Lisp will insulate code from internal Jack changes, making lisp code upward-compatible from one version of Jack to a later version.

F.2 The Lisp Interface

The particular lisp we have chosen is XLISP-STAT, developed by Luke Tierney and David M. Betz. It has the following features:

1. It implements a large subset of common-lisp.

2. It has a relatively small image (unlike the public domain full common-lisps).

3. It has built in object-oriented extensions— a prototyping mechanism, which is a generalization of class hierarchies.

4. It has built-in math (including vector and matrix operations), statistical, and interactive graphing functions.

5. It provides the standard common-lisp debugger with stepping, tracing, stack printing, and variable printing or modifying.

6. It runs under Microsoft Windows, on Macintoshes, on Amigas, and on any machine with X-windows and a C compiler. This will make it possible to do lisp development without running Jack.

\(^1\)Welton Becket (Tripp)
7. It has no liscencing fees whatsoever: permission to “use, modify, distribute, and sell” XLISP-STAT is granted without fee or restriction.

The lisp interface as it appears initially in Jack-5.8 has the following features:

1. **Unified compilation** with Jack, with efficient, bidirectional callouts— lisp can send and receive information from Jack, and Jack can execute lisp code, get or set the value of variables, etc...

2. Set of approximately 100 **basic primitives** for accessing and modifying low level Jack internals (documentation in Section F.6):
   - Direct access to most major Jack datatypes: Figures, Segments, Joints, Constraints, and Sites. Attributes, Textures, and Psurfs will be done in the 2nd Quarter 93.
   - Low level access to the motion system

3. A **JCL callout mechanism** that allows any JCL or Peabody expression to be evaluated. There are several functions for making construction of JCL strings easier. Basic documentation on this feature is in Section F.4.

4. **Lisp shells around all Jack motions** (including human, figure, and timed motions) that allow higher-level construction. Each motion call takes only a few arguments and uses defaults for the rest which can be overridden with keyword arguments. There is online documentation for each of these which is also printed below in Section F.5.

5. **User input functions** from within lisp, like input-string or input-figure, that call the standard familiar Jack input procedures.

6. **Output functions** that print to the three Jack windows types.

7. **Callback functions** that are executed on every Jack frame. This allows lisp code that is self-timingslicing to run in the background while Jack is running. This is described in Section F.3.3

8. **Motion callback functions** similar to the standard Callback functions, but executed only while the motion system is running. These can be used to modify the animation as it progresses, pasting new motions to be executed or killing motions in progress. This is described in Section F.3.3.

9. **Lisp motions** that are actual motions in the motion system. The apply, preaction, postaction, etc. are pieces of lisp code. New lisp motion types can be created and destroyed dynamically.

10. **Lisp menu functions**. There are functions that at run-time can add items to the Jack menus with lisp code attached to them. This will allow calling user-defined code interactively without having to type at the lisp-prompt (completely insulating intended users of the user-defined code from the lisp system).

11. **Remote Lisp Control**— An xlisp process running either in Jack or stand-alone on any machine can have bi-directional communication with another xlisp.

### F.3 Using XLISP-STAT

XLISP can currently be accessed from within Jack at the user level from a lisp prompt, through menu items bound to lisp code, or from callback functions executed on every Jack frame or every animation frame. To use it you must have the JACKXLISP environment variable set as:

```
JACKXLISP=/pkg/jack/jacklib5/lisp/
```

You must include the final ‘/’ in the JACKXLISP directory.

On startup, the file `$(HOME)/.jack.lisp` is read and executed if it exists.

**NOTE:** In addition to the user-level interface to XLISP, there is also a code-level interface where lisp strings constructed in C or C++ can be executed, and there is a socket interface which allows bidirectional communication between a remote process and the lisp within a running Jack process. These are currently undocumented.
F.3.1 The XLISP prompt

The command xlisp will start an xlisp prompt in the shell where Jack was invoked. Control is not returned to Jack until the user explicitly leaves xlisp by typing CTRL-D at the xlisp prompt. While using the xlisp prompt, the Jack window is not updated unless a user interface function is called. The lisp function REDRAW can also be used to explicitly redraw all Jack windows.

The command-xlisp string asks for a lisp string in the status window. The string is then executed in lisp and the result is printed on the Jack overlay planes.

F.3.2 Menu Items

New menu items can be added to Jack that execute lisp code when invoked. The basic approach is to use:

1. (MENU-BEGIN) — this destroys the old Jack menus including any previous lisp menu items commands defined using MENU-ADD. It then rebuilds the standard Jack menus.
2. (MENU-END) — rebuilds the Jack command structure and rebuilds the displayed menus.
3. (MENU-ADD (MENU-GET) name-str code) — this adds a new command to the end of the top-level menu with the given name and bound lisp code. MENU-GET currently only returns the top level Jack menu, though later it will be extended to return a submenu of a menu. The code can access global variables or can have local persistent variables using lexical closures. An example without variables:

   (menu-begin)
   (menu-add (menu-get) "Fred"
     '(format t "Fred must have goat cheese or he will die.\"%\")
   )
   (menu-end)

This will add the command Fred to the menus and to jcl, that will print a string to the shell where Jack was invoked. An example with bound variables:

   (menu-begin)
   (menu-add (menu-get) "Sam"
     (let ((count 0))
       (list #\'(lambda ()
         (format t "Frame: \"a\"%" count)
         (setf count (1+ count)))))
   )
   (menu-end)

F.3.3 Callback Functions

Callback functions can be bound that are executed on every Jack frame (like a background simulation function), or just on every motion system frame (only while an animation is running).

Use:

(BIND-BG-CALLBACK namestr code)

for bind code executed on every frame, and use:

(BIND-MOTION-CALLBACK namestr code)

for bind code only executed during an animation. The namestr argument can be any lisp object that is unique for all callbacks defined (otherwise the previous one with the same object is overwritten). To unbind a callback function, use:

(UNBIND-BG-CALLBACK namestr), and
(UNBIND-MOTION-CALLBACK namestr)

to unbind individual callbacks, or:

(UNBIND-BG-CALLBACKS), and
(UNBIND-MOTION-CALLBACKS)

to delete all callbacks of the appropriate type. The code argument for callback functions is the same as for MENU-ADD described in Section F.3.2.
F.4 JCL calls

The lowest level access to JCL (and Peabody in general) is through the rawjcl function. This takes a string argument and executes it as peabody code. JCL commands must end with a semi-colon, and all embedded quotes must be preceded with a \" so the parser knows where the whole jcl string ends. For example:

(rawjcl "movefigure(\"cube\",trans(0,100,0));")

will execute supplied JCL command assuming there is a figure named cube.

The jcl function provides slightly higher level access. The first argument to jcl must be the name of the JCL function as a string or a symbol, and the rest of the arguments should be the arguments of the JCL call. The jcl function puts parentheses around the argument list, commas between each argument, and a semi-colon at the end. Any argument passed is converted to a string before being stuffed in the jcl call. The above call would then be:

(jcl 'movefigure "\"cube\\"", 'trans(0,100,0)")

Several functions are available for building strings for common jcl constructs:

(js str)
Embeds the argument in quotes and returns as a string. If str is a symbol, converts it to a string in lower-case. For example: (js 'cube) returns "\"cube\"".

(jtrans x y z)
Returns a jcl translation string from the three arguments, which can be strings or numbers. For example, (jtrans 0 100 "0") returns "trans(0,100,0)".

(jxyz x y z)
Returns a jcl rotation string from the three arguments, which can be strings or numbers. For example, (jxyz 0 90 "0") returns "xyz(0,90,0)".

(jmult a b)
Returns a jcl product string of a and b. For example, (jmult (jxyz 0 90 0) (jtrans 0 100 0)) returns "xyz(0,90,0)*trans(0,100,0)".

(junits n unitstr)
Appends n (which should be a string or a number) with unitstr, which should be a string or symbol specifying units. For example, (junits 2.0 "sec") returns "2.0sec".

The above example now becomes:

(jcl 'movefigure (js 'cube) (jtrans 0 100 0))

This becomes most useful when using variables, for example if the name of the figure is in the variable fig and the height is in the variable y, we can execute:

(jcl 'movefigure (js fig) (jtrans 0 y 0))

Other useful functions for constructing jcl strings are the builtin functions strcat and format (all of the above functions are defined in terms of these).

F.5 Motion Shells

All of these shells construct a JCL string that is executed in Jack to paste the appropriate motion in the timeline. Any argument to any of these functions can be the special symbol $, to have JCL call for user input. For example, instead of executing:

(move-figure 'cube 1.0 2.0 (jtrans 0 100 0))

which would call for a figure motion for cube from time 1sec to 2sec to the location (0,100,0), you could pick the location and the time interactively like:

(move-figure 'cube $ $ $)
F.5. MOTION SHELLS

---

ARM-MOTION

Syntax: (arm-motion hstr start end leftp mat
   &key (ref 'palm) (relativep nil) (weight 1.0)
   (wf 'const) (vc 'ease))

   hstr - name of human (string)
   start - start time in seconds
   end - ending time in seconds
   leftp - t for left, nil for right
   mat - a 4x4 matrix, a jcl matrix string

   ref - one of 'forearm 'palm or 'attached. 'palm by default.
   relativep - nil by default, pass as t for relative
   weight - weight of this motion (increase for higher priority).
   wf - weight function, see wf-to-str
   vc - velocity control, see vc-to-str

HEEL-MOTION

Syntax: (heel-motion hstr start end leftp height
   &key (weight 2.0) (wf 'decay) (vc 'ease))

   hstr - name of human (string)
   start - start time in seconds
   end - ending time in seconds
   leftp - t for left, nil for right
   height - height of the heel motion in cm

   weight - weight of this motion (increase for higher priority).
   wf - weight function, see wf-to-str
   vc - velocity control, see vc-to-str

PELVIS-MOTION

Syntax: (pelvis-motion hstr start end mat
   &key (weight 5.0) (wf 'decay) (vc 'ease))

   hstr - name of human (string)
   start - start time in seconds
   end - ending time in seconds
   mat - a 4x4 matrix, a jcl matrix string

   weight - weight of this motion (increase for higher priority).
   wf - weight function, see wf-to-str
   vc - velocity control, see vc-to-str

TORSO-MOTION

Syntax: (torso-motion hstr start end angles
   &key (tp 'bend) (vc 'const))

   hstr - name of human (string)
   start - start time in seconds
   end - ending time in seconds
   angles - list of angles as (f a I) for the default human:
   f = flexion in [-51.78,84.02]
   a = axial in [-42.63,42.27]
   I = lateral in [-38.94,39.46]
   tp - 'bend to bend from waist or 'curl to curl
   from neck.
   vc - velocity control, see vc-to-str

---

COM-MOTION

Syntax: (com-motion hstr start end mat
   &key (how 'location) (weight 2) (wf 'decay)
   (vc 'ease))

   hstr - name of human (string)
   start - start time in seconds
   end - ending time in seconds
   how - one of 'between, 'location for "between feet" or "location" motion types.

   weight - weight of this motion (increase for higher priority).
   wf - weight function, see wf-to-str
   vc - velocity control, see vc-to-str

FIGURE-MOTION

Syntax: (figure-motion figstr start end mat &key (vc 'ease))

   figstr - name of figure (string)
   start - start time in seconds
   end - ending time in seconds
   mat - a 4x4 matrix, a jcl matrix string

   vc - velocity control, see vc-to-str

F.5.1 Figure Motions

CAMERA-MOTION

Syntax: (camera-motion wstr start end mat &key (vc 'ease))

   wstr - name of camera (string)
   start - start time in seconds
   end - ending time in seconds
   mat - a 4x4 matrix, a jcl matrix string

   vc - velocity control, see vc-to-str

---

FOOT-MOTION

Syntax: (foot-motion hstr start end leftp mat
   &key (height 5.0) (pos-orient 0.5)
   (weight 2.0) (wf 'const) (vc 'ease))

   hstr - name of human (string)
   start - start time in seconds
   end - ending time in seconds
   leftp - t for left, nil for right
   mat - a 4x4 matrix, a jcl matrix string

   height - height of the foot motion in cm
   pos-orient - position/orientation weight in [0,1] (0 for position only, 1 for orientation only).
   weight - weight of this motion (increase for higher priority).
   wf - weight function, see wf-to-str
   vc - velocity control, see vc-to-str

"Figure Motions"
APPENDIX F. THE LISP INTERFACE

figstr - name of a figure (string)
start - start time in seconds
end - ending time in seconds
mat - a 4x4 matrix, a jcl matrix string
("xyz(...)*trans(...)") or $ to generate an
input-transform. See mat-to-str.
:vc - velocity control, see vc-to-str

JOINT-MOTION

Syntax: (joint-motion fstr start end displacements &key vc)

hstr - name of human (string)
start - start time in seconds
end - ending time in seconds
displacements - a list of angles, one for each
degree of freedom, or a $ for user input.
:vc - velocity control, see vc-to-str

F.5.2 Timed Controls

TIMED-BALANCE-CONTROL

Syntax: (timed-balance-control hstr start end &key (btype 'follow))

hstr - name of human (string)
start - start time in seconds
end - ending time in seconds :btype - one of:
"follow" "hold-pos" "hold-elev" "release-elev" "seated" "use upper body"

TIMED-FOOT-CONTROL

Syntax: (timed-foot-control hstr start end &key (which 'both) (foot-type 'hold-global))

hstr - name of human (string)
start - start time in seconds
end - ending time in seconds :which - 'left, 'right, or 'both
:foot-type - one of: "release" "hold-global location" "hold local location" "follow balance line"

TIMED-HAND-CONTROL

Syntax: (timed-hand-control hstr start end &key (which 'both) (hand-type 'hips)
(start-joint 'shoulder or 'waist)

:hand-type - one of:
hips knees hold-global hold-local site release
:start-joint - shoulder or waist

TIMED-HEAD-CONTROL

Syntax: (timed-head-control hstr start end &key (eye-type 'fixate) (goal-type ('hold)))

hstr - name of human (string)
start - start time in seconds
end - ending time in seconds :eye-type - 'fixate or 'release
:goal-type - list of arguments defining the goal.
Passes directly to goal-to-str.

TIMED-PELVIS-CONTROL

Syntax: (timed-pelvis-control hstr start end &key (pelvis-type 'hold))

hstr - name of human (string)
start - start time in seconds
end - ending time in seconds :pelvis-type - one of: 'hold or 'follow, for "hold orientation" or "follow feet".

TIMED-TORSO-CONTROL

Syntax: (timed-torso-control hstr start end &key (torso-type 'vertical) (bend-type 'bend))

hstr - name of human (string)
start - start time in seconds
end - ending time in seconds :torso-type - one of: 'vertical, 'hold, or 'none. For
"vertical", "hold orientation", or "none".
:bend-type - one of: "curl" or "bend for curl from neck" or "bend from waist".

F.6 Lisp Primitives

These functions are the lowest level interface to Jack from lisp. They provide direct access to Jack data
structures and functions. They are very fast, but relatively unprotected.

F.6.1 Utility Callouts

SUBSTR

Syntax: (substr string sub-str)
Notes: If sub-str is a sub-string of string, returns the rest of
the string at the point where sub-str first appears.
Otherwise returns nil.
F.6. LISP PRIMITIVES

F.6.2 JCL Calls

RAWJCL
Syntax: (rawjcl &rest strings)
Notes: raw jcl callout mechanism. Executes the strings, which must already end in semicolons. Returns TRUE if all of the jcl calls succeeded.

REDRAW
Syntax: (REDRAW)
Notes: no args, redraws all jack windows

A NOTE ON POINTERS:
In these low-level access functions, all jack objects are referred to by actual pointers. When you call (figure-create "cube.pss"), a fixnum (long integer) is returned that is the address of the created figure. These pointers are currently unprotected, and there are no checks to see if the object pointed to is of the correct type (which can cause core dumps very easily). Soon there will be a whole class hierarchy mechanism over these low-level functions that will offer both pointer protection and higher-level access.

F.6.3 Figure Access Functions

FIGURE-CREATE
Syntax: (figure-create filename &optional figname)
Notes: First arg is the name of the file to read, figname is optional and is the name the figure will have. Currently works only for reading psurf files. Read figure files using the jcl command readfile.

FIGURE-TRANSFORM
Syntax: (figure-transform figptr &optional matrix-4x4)
Notes: First arg must be a figure pointer. If there is no second arg it gets the figure transform and returns it as a 4x4 matrix. If there is a second arg, it must be a 4x4 matrix that is to be the new transform for the figure.

FIGURE-BBOX
Syntax: (figure-bbox figptr)
Notes: Returns the bounding box as two 3-vecs in a list (min, max).

FIGURE-XZRADIUS
Syntax: (figure-xzradius figptr)
Notes: Returns the radius of the objects bounding cylinder in the global xz plane.

FIGURE-LOCALBBOX
Syntax: (figure-localbbox figptr)
Notes: Returns the axis-aligned bbox of the figure in its local space as two 3-vecs in a list (min, max).

FIGURE-ROOTSITE
Syntax: (figure-rootsite figptr)
Notes: Returns a pointer to the rootsite for the figure.

FIGURE-COM
Syntax: (figure-com figptr &optional globalOrLocal)
Notes: If globalOrLocal is present, it must be 'global or 'local. Default is 'global.

FIGURE-SEGMENT
Syntax: (figure-segment figptr &optional segname)
Notes: If segname is given, it must be a string that is the name of the segment within figptr to return. Otherwise a list of all segments in figptr is returned.

FIGURE-SITE
Syntax: (figure-site figptr &optional sitename)
Notes: If sitename, it should be a string that is the name of the site within figptr to return. Otherwise a list of all sites is returned.

FIGURE-JOINT
Syntax: (figure-joint figptr &optional jointname)
Notes: If jointname, it should be a string that is the name of the joint within figptr to return. Otherwise returns a list of all joints. Returns nil if joint not found or no joints.

FIGURE-FIND
Syntax: (figure-find &optional figname)
Notes: If figname present, it should be a string that is the name of the figure within the env to return. Otherwise, returns a list of all figures in the environment.

FIGURE-NAME
Syntax: (figure-name figptr)
Notes: Returns the name of the figure as a string.

FIGURE-ON
Syntax: (figure-on figptr1 figptr2)
Notes: Takes two figures and describes the on relationship between them - returns:
'FIG1-ON-FIG2 - if fig1 on fig2
'FIG2-ON-FIG1 - if fig2 on fig1
'TOUCHING - if fig1 and fig2 intersect but there is no 'physically based' on relationship.
'NOT-TOUCHING - if no intersections at all. Does not work on humans.

FIGURE-TORQUE
Syntax: (figure-torque figptr &optional initp)
APPENDIX F. THE LISP INTERFACE

Notes: Re-evaluates torque for a figure. If initp true, initializes torque for a figure (which MUST be done once for a figure - it's done by the torque window so no need to do it if there's a torque window on the figure).

F.6.4 Segment Acess Functions

SEGMENT-TRANSFORM
Syntax: (segment-transform segptr)
Notes: Returns the segment transform as a 4x4 matrix. The first three entries is the global position vector, the upper 3x3 is the rotation matrix.

SEGMENT-BBOX
Syntax: (segment-bbox segptr)
Notes: Returns the segment's global bounding box as two 3-vectors in a list, ( #(xmin ymin zmin) #(xmax ymax zmax) ).

SEGMENT-XZRADIUS
Syntax: (segment-xzradius segptr)
Notes: Returns the radius of the segment in the xz plane.

SEGMENT-LOCALBBOX
Syntax: (segment-localbbox segptr)
Notes: Returns the segment's local bounding box as two 3-vectors in a list, ( #(xmin ymin zmin) #(xmax ymax zmax) ).

SEGMENT-ROOTSITE
Syntax: (segment-rootsite segptr)
Notes: Returns a pointer to the segment's root site.

SEGMENT-COM
Syntax: (segment-com segptr &optional space)
Notes: If space is present, it should be either 'global or 'local to say whether to get the center of mass in global or local coordinates (default is 'global). The center of mass is returned as a 3-vector.

SEGMENT-SITE
Syntax: (segment-site segptr &optional sitename)
Notes: Returns the site or nil if sitename present, otherwise returns a list of pointers to all sites in the segment.

SEGMENT-NAME
Syntax: (segment-name segptr)
Notes: Returns name of the segment as a string or nil if no name.

SEGMENT-FULLNAME
Syntax: (segment-fullname segptr)
Notes: Returns full name of the segment as a string.

SEGMENT-FILNAME
Syntax: (segment-filename segptr)
Notes: Returns the filename for a segment as a string, or nil if none.

SEGMENT-ATTRIBUTES
Syntax: (segment-attributes segptr &optional n newattr)
Notes: If n and newattr not present, returns a list of all attributes for segptr. If n present returns attribute n for segptr. If n and newattr, sets attribute n for segptr to newattr. Returns nil if the segment has no psurf. On an attribute set, returns t if the set was successful or nil if not (out of bounds). On an attribute get, returns the attribute on success, or nil on out of bounds.

SEGMENT-HASGEOM
Syntax: (segment-hasgeom segptr)
Notes: Returns t if the segment has a psurf, nil otherwise.

SEGMENT-NNODES
Syntax: (segment-nnodes segptr)
Notes: NOT IMPLEMENTED YET.

SEGMENT-NODES
Syntax: (segment-nodes segptr)
Notes: NOT IMPLEMENTED YET.

SEGMENT-NATRTRS
Syntax: (segment-nattrs segptr)
Notes: NOT IMPLEMENTED YET.

SEGMENT-ATTRS
Syntax: (segment-atrrs segptr)
Notes: NOT IMPLEMENTED YET.

SEGMENT-NEDGES
Syntax: (segment-nedges segptr)
Notes: NOT IMPLEMENTED YET.

SEGMENT-EDGES
Syntax: (segment-edges segptr)
Notes: NOT IMPLEMENTED YET.

SEGMENT-NFACES
Syntax: (segment-nfaces segptr)
Notes: NOT IMPLEMENTED YET.

SEGMENT-FACES
Syntax: (segment-faces segptr)
Notes: NOT IMPLEMENTED YET.

SEGMENT-ADDSITE
Syntax: (segment-addsite segptr name location)
F.6. LISP PRIMITIVES

F.6.5 Joint Access Functions

JOINT-NAME
Syntax: (joint-name jptr)
Notes: Returns the name of the joint as a string.

JOINT-FULLNAME
Syntax: (joint-fullname jptr)
Notes: Returns the full name of the joint as a string.

JOINT-FIGURE
Syntax: (joint-figure jptr)
Notes: Returns a pointer to a joint's figure.

JOINT-FROM
Syntax: (joint-from jptr)
Notes: Returns a pointer to the 'from' site of a joint.

JOINT-TO
Syntax: (joint-to jptr)
Notes: Returns a pointer to the 'to' site of a joint.

JOINT-ROOTSITE
Syntax: (joint-rootsite jptr)
Notes: Returns a pointer to the root site of a joint.

JOINT-NDOFS
Syntax: (joint-ndofs jptr)
Notes: Returns the number of degrees of freedom of a joint.

JOINT-AXIS
Syntax: (joint-axis jptr &optional n)
Notes: Returns the axis for degree of freedom n for a joint as a 3-vec, or nil if a bound error on n. Returns a list of all axes if n not present.

JOINT-TYPE
Syntax: (joint-type jptr &optional n)
Notes: Returns the lisp character 'r' or 't' for translational or rotational joint for degree of freedom n (nil if bound error). If n not present, returns a list of all types.

JOINT-ANGLE
Syntax: (joint-angle jptr &optional (n newangle) — newangles)
Notes: If no optional args returns list of angles for each dof in radians. If n present returns just the angle for that dof or nil on a bound error. If n and newangle sets the dof angle returning t on success or nil on failure. If the optional arg is a list, it is assumed to be a list of new angles for the dofs for the joint and returns t or nil on success or failure for the set.

JOINT-ULIMIT
Syntax: (joint-ulimit jptr &optional (n newulimit) — ulimits)
Notes: If no optional args, returns upper limits for each dof for joint jptr as a list. If n present as a fixnum, returns upper limit for dof n or nil on a bound error. If n and newulimit sets upper limit n to newulimit returning t on success, or nil on failure. If optional arg is a list, it is assumed to be a list of numbers that will be the new upper limits for each dof (the list must be of the correct size).

JOINT-LLIMIT
Syntax: (joint-llimit jptr &optional (n newllimit) — llimits)
Notes: See j-jointulimit().

JOINT-TORQUE
Syntax: (joint-torque jptr &optional n)
Notes: Gets the torque for degree of freedom n for jptr. Value is undefined if torque computation has not been setup (though probably 0.0). Returns a list of torques if n not present.

JOINT-PREVANGLE
Syntax: (joint-prevangle jptr &optional n)
Notes: Returns previous angle for dof n or a list of previous angles if n not present.

F.6.6 Site Access Functions

SITE-NAME
Syntax: (site-name sptr)
Notes: Returns the name of the site as a string.
APPENDIX F. THE LISP INTERFACE

SITE-FULLNAME
Syntax: (site-fullname sptr)
Notes: Returns the full name of the site as a string.

SITE-SEGMENT
Syntax: (site-segment sptr)
Notes: Returns a pointer to the segment that owns the site.

SITE-JOINTS
Syntax: (site-joints sptr)
Notes: Returns a list of pointers to joints attached to the site.

SITE-CONSTRAINTS
Syntax: (site-constraints sptr)
Notes: Returns a list of pointers to constraints attached to the site.

SITE-ROOTJOINT
Syntax: (site-rootjoint sptr)
Notes: Returns a pointer to the root joint for the site.

SITE-LOCATION
Syntax: (site-location sptr &optional space)
Notes: Returns global site location by default as a 3-vector. If space present, it must be 'global or 'local and the return value will be global or local to the segment.

F.6.7 Jack Input Functions

INPUT-BOOLEAN
Syntax: (input-boolean prompt &optional default)
Notes: Prompt must be a string. Default, if present, must be nil for default or a string. Extension, if present, must be a string that is the extension for the file. If the file already exists, asks the user if they want to overwrite. Returns 'cancel if the user "C"d, or a string on success.

INPUT-FLOAT
Syntax: (input-float prompt &optional default)
Notes: Prompt must be a string. Default, if present, should be a number (if not present it's set to 0.0). Returns a flonum on success, or 'cancel if the user "C"d.

INPUT-INT
Syntax: (input-int prompt &optional default)
Notes: Prompt must be a string. Default, if present, should be a number (if not present it's set to 0.0). Returns a flonum on success, or 'cancel if the user "C"d.

INPUT-VECTOR
Syntax: (input-vector prompt (default-size))
Notes: Prompt must be a string. The second arg must be a vector of floats for a default vec, or a size if there is no particular default. Returns a vector on success, or 'cancel if the user "C"d.

INPUT-TRANSFORM
Syntax: (input-transform prompt &optional default)
Notes: Prompt must be a string. Default, if present, must be a matrix. Returns the matrix on success, or 'cancel if the user "C"d.

INPUT-NAMEDTYPE
Syntax: (input-namedtype prompt choices &optional default)
Notes: Prompt must be a string. Choices must be a list of alternative strings. Default, if present, must be a string or nil for no default. Returns the chosen string on success, or 'cancel if the user "C"d.

INPUT-LISP
Syntax: (input-lisp prompt &optional default)
Notes: Gets a lisp expression and returns the unevaluated result. Prints the lisp expr in 'default' as the default value, if it's present. Returns 'cancel if user hit "C" and 'error on a read error.

INPUT-STRING
Syntax: (input-string prompt &optional default)
Notes: Prompt must be a string. Default, if present, must be nil for no default or a string. Returns 'cancel if the user "C"d or a string on success.

INPUT-INLINE
Syntax: (input-infile prompt &optional default)
Notes: Prompt must be a string. Default, if present, must be nil for no default or a string. Returns 'cancel if the user "C"d or a string on success.

INPUT-OUTFILE
Syntax: (input-outfile prompt &optional default extension)
Notes: Prompt must be a string. Asks the user to select a file and returns a pointer to the file.

INPUT-SEGMENT
Syntax: (input-segment prompt)
F.6. LISP PRIMITIVES

INPUT-SITE
Syntax: (input-site prompt)
Notes: Prompt must be a string. Asks the user to select a site and returns a pointer to the site.

INPUT-JOINT
Syntax: (input-joint prompt)
Notes: Prompt must be a string. Asks the user to select a joint and returns a pointer to the joint.

INPUT-ATTRIBUTE
Syntax: (input-attribute prompt)
Notes: Prompt must be a string. Asks the user to select an attribute and returns a pointer to the attribute.

INPUT-NODE
Syntax: (input-node prompt)
Notes: Prompt must be a string. Asks the user to select a node. Returns a list whose first elem is the segment ptr and whose second is the node index.

INPUT-EDGE
Syntax: (input-edge prompt)
Notes: Prompt must be a string. Asks the user to select an edge. Returns a list whose first elem is the segment ptr and whose second is the edge index.

INPUT-FACE
Syntax: (input-face prompt)
Notes: Prompt must be a string. Asks the user to select a face. Returns a list whose first elem is the segment ptr and whose second is the face index.

F.6.8 Output Messages To Jack

OUTPUT-LOGMSG
Syntax: (output-logmsg str type)
Notes: Calls LogMsg(). Prints the string to the log window. The String can have newlines. Type must be one of:
'normal
'hilite
'reverse
'underline
'hilite-reverse

OUTPUT-STATMSG
Syntax: (output-statmsg str)
Notes: Prints a status message to the status line.

OUTPUT-STATERROR
Syntax: (output-staterror str)
Notes: Prints an error message to the status line and beeps.

F.6.9 Menu Functions

MENU-BEGIN
Syntax: (menu-begin)
Notes: Initiates rebuilding of the menus. Any menu changes should take between (menu-begin) and (menu-end).

MENU-END
Syntax: (menu-end)
Notes: Called after changes to menus are finished. Rebuilds the menus and the command list.

MENU-GET
Syntax: (menu-get)
Notes: Currently this function takes no args and only returns a pointer to the main jack menu. Later it may be extended to find a menu inside another menu...

MENU-NEW
Syntax: (menu-new name)
Notes: Returns a new, empty menu with the given name.

MENU-ADD
Syntax: (menu-add into (menu - (name action)))
Notes: 'into' must be a pointer to a menu. If adding a menu to 'into', pass a pointer to the menu to add.
If adding a command to 'into', pass a string that will be the name of the command, and an action, which must be a lisp expression to evaluate when the command is selected.

F.6.10 Motion Functions

MOTION-LAST
Syntax: (motion-last)
Notes: Returns a pointer to the last motion created (by whatever means...lisp, jcl, interactive).
MOTION-KILL
Syntax: (motion-kill motionptr)
Notes: Frees space for a motion and removes it from the list of active motions. Executes the motion's postaction if it is currently running.

MOTION-STATUS
Syntax: (motion-status motionptr)
Notes: Returns the status of the motion as:
'active
'executed
'waiting

MOTION-TIME
Syntax: (motion-time &optional newtime)
Notes: Returns current time in seconds if no args. Sets current time to 'newtime', which must be a flonum if it is present.

MOTION-ON
Syntax: (motion-on &optional onP)
Notes: If no args, returns t if the motion system is advancing and nil if not. If there is an argument, if it is nil, the motion system is stopped; otherwise it is started if it is not already running.

MOTION-CURRENT
Syntax: (motion-current &optional motionptr)
Notes: If no arg, returns a pointer to the current motion. Otherwise, sets the current motion (the current motion in the animation windows - needed for changing a motion using jcl).

MOTION-START
Syntax: (motion-start motionptr &optional starttime)
Notes: If starttime present, sets start time of motion, otherwise returns start time (all in seconds).

MOTION-DUR
Syntax: (motion-dur motionptr &optional duration)
Notes: If starttime present, sets start time of motion, otherwise returns start time (all in seconds).

MOTION-LISP
Syntax: (motion-lisp name start dur apply &optional data preaction postaction)
Notes: Creates a new lisp motion with the given start time and duration. Name is a documentation string that will appear in the motion window. The preaction is called when the motion starts, postaction when it finishes, and apply is called on every frame. Pass nil to have no effect for that function. When apply, preaction, and postaction are executed, (MOTION-DATA) will return the data for the motion, and (MOTION-PTR) will return the motion ptr for the motion. Returns a pointer to the motion.

MOTION-DATA
Syntax: (motion-data)
Notes: Gets data for a lisp motion whose apply, pre, or post function is currently active. Returns nil if no data or no lisp motion currently active.

MOTION-PTR
Syntax: (motion-ptr)
Notes: Gets the motion pointer for the currently active lisp motion if one is running (otherwise returns nil).

MOTION-FIND
Syntax: (motion-find &optional namestr)
Notes: If namestr present, looks for a motion with the given full-name and returns a pointer to it, otherwise returns a list of pointers to all motions. Returns nil if name not found or no motions.

MOTION-NAME
Syntax: (motion-name mptr &optional local)
Notes: Returns the full name of the motion as a string as "motionsetname.motionname". If local present and non-nil, returns only the local name - "motion-name".

F.6.11 4x4 Matrix Primitives

MATRIX-TOJCL
Syntax: (matrix-tojcl matrix-4x4)
Notes: Returns jcl representation of the matrix as a string.

MATRIX-FROMJCL
Syntax: (matrix-fromjcl jclstr)
Notes: Returns lisp representation of a jcl matrix. (Given a jcl matrix as "xyz(90,0,90)*trans(0.100,0)", returns the lisp matrix #2A((...)(...)(...)(...))).

MATRIX-TRANS
Syntax: (matrix-trans (x y z) - matrix-4x4 &optional newtrans)
Notes: If first arg is a matrix:
Sets or gets translation component of the 4x4 matrix. If newtrans is present it must be a 3-vec, and the matrix's translation component will be set to it. If newtrans is not present, returns the current translation component of the matrix.
If first arg is a number:
Returns a new translation matrix using x y and z, which must all be numbers.

MATRIX-ROT
Syntax: (matrix-rot axis theta)
Notes: Returns a rotation matrix of angle 'theta' (float in radians) around 'axis' (a 3-vector).

MATRIX-XYZ
Syntax: (matrix-xyz rx ry rz)
Notes: Returns a rotation matrix of rx, ry, and rz (all floats in radians) around the x, y, and z axes.
F. 6. LISP PRIMITIVES

MATRIX-ID
Syntax: (matrix-id)
Notes: Returns a new copy of the 4x4 id matrix.

MATRIX-INV
Syntax: (matrix-inv matrix-4x4)
Notes: Returns the inverse of the given matrix, which must be homogeneous (no shear or scaling).

MATRIX-MULT
Syntax: (matrix-mult base delta &optional (localp t))
Notes: Transforms the 4x4 matrix 'base' by the 4x4 matrix delta. Default is a local transformation of base by delta. If localp set to nil then does a global transformation.

MATRIX-HOMOGENEOUS
Syntax: (matrix-homogeneous matrix-4x4)
Notes: Returns t if the matrix is homogeneous, otherwise nil.

MATRIX-INTERP
Syntax: (matrix-interp from to tval)
Notes: Returns a linearly interpolated matrix between 4x4 matrices 'from' and 'to' using tval. Tval=0 is 'from' and tval=1 is 'to'.

F.6.12 Motion Group Primitives

MGROUP-EXPANDED
Syntax: (mgroup-expanded mgptr expandedp)
Notes: If expandedp true, then sets the display status to expanded, otherwise to contracted.

MGROUP-NAME
Syntax: (mgroup-name mgptr)
Notes: Returns the name for the motion group.

MGROUP-START
Syntax: (mgroup-start mgptr &optional starttime)
Notes: If starttime present, sets the start time for the motion group to the given value, which must be in seconds. Otherwise returns the current value.

MGROUP-DUR
Syntax: (mgroup-dur mgptr &optional dur)
Notes: If duration present, sets the duration for the motion group to the given value, which must be in seconds. Otherwise returns the current value.

MGROUP-FIND
Syntax: (mgroup-find &optional name)
Notes: If name, looks for a motion group with the given name (which must be a string), and returns a pointer to it, or nil if not found. If name not present, returns list of pointers to motiongroups.
APPENDIX F. THE LISP INTERFACE

F.7 Changes in Revision 1.0

1. Changed:
   - FIGURE-TRANSLATION should have been FIGURE-TRANSFORM

2. New lisp primitives:
   - FIGURE-TORQUE
   - SEGMENT-MASS
   - SEGMENT-HIGHLIGHT
   - MOTION-FIND
   - MOTION-NAME
   - MATRIX-FROMJCL
   - MGROUP-*

F.8 Copyright

This is the copyright information for XLISP which is used in Jack:

XLISP-STAT 2.1 Copyright (c) 1990, by Luke Tierney
XLISP version 2.1, Copyright (c) 1989, by David Betz.

Permission to use, copy, modify, distribute, and sell this software and its
documentation for any purpose is hereby granted without fee, provided that
the above copyright notice appear in all copies and that both that
copyright notice and this permission notice appear in supporting
documentation, and that the name of Luke Tierney and David Betz not be
used in advertising or publicity pertaining to distribution of the software
without specific, written prior permission. Luke Tierney and David Betz
make no representations about the suitability of this software for any
purpose. It is provided "as is" without express or implied warranty.

LUKE TIERNEY AND DAVID BETZ DISCLAIM ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS,
IN NO EVENT SHALL LUKE TIERNEY NOR DAVID BETZ BE LIABLE FOR ANY SPECIAL,
INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE
OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.

XLISP-STAT AUTHOR:
Luke Tierney
School of Statistics
University of Minnesota
Minneapolis, MN 55455
(612) 625-7843

Email Address:
internet: luke@umnstat.stat.umn.edu

XLISP AUTHOR:
David Betz
P.O. Box 144
Peterborough, NH 03458
(603) 924-4145
Appendix G

JCL Reference Manual

This chapter is a reference manual for the JCL form of most of the Jack commands. The commands are organized in a mostly alphabetic listing. Some commands that are intimately related are listed together. If you can't find a command where you expect it, try finding it in the index first.
CQ_check_environment

This command adds a collision pair for every segment pair in the environment to the collision queue.

Example:
CQ_check_environment();

CQ_check_figure(figure)

figure : A figure name string.

This command adds a collision check between every segment in figure and every segment in the environment to the collision queue. It does not add intra-figure checks.

Example:
CQ_check_figure("figure");

See Also:
CQ_clear_intra_figure

CQ_check_figure_pair(figure1, figure2)

figure1 : A figure name string.
figure2 : A figure name string.

This command adds a collision check between every segment in figure1 and every segment in figure2 to the collision queue. It does not add intra-figure checks.

Example:
CQ_check_figure_pair("figure1", "figure2");

See Also:
CQ_clear_intra_figure

CQ_check_segment(segment)

segment : A segment name string.

This command adds a collision check between segment and every other segment in the environment to the collision queue.

Example:
CQ_check_segment("cube1.cube");

CQ_check_segment_pair(segment1, segment2)

segment1 : A segment name string.
segment2 : A segment name string.

This command adds a collision check between segment1 and segment2 to the collision queue.

Example:
CQ_check_segment_pair("cube1.cube", "cube2.cube");

CQ_clear_current_figure(figure)

figure : A figure name string.

This command removes every check between two segments of figure that are currently in collision from the collision queue. This is primarily a way to signal that the current intra-figure collisions are ok.

Example:
CQ_clear_current_figure("cube");

See Also:
CQ_clear_intra_figure, CQ_clear_segment_pair

CQ_clear_figure(figure)

figure : A figure name string.

This command removes all checks involving any segments of figure from the command queue.

Example:
CQ_clear_figure("cube");

See Also:
CQ_clear_intra_figure

CQ_clear_figure_pair(figure1, figure2)

figure1 : A figure name string.
figure2 : A figure name string.

This command removes all checks between segments of figure1 and segments of figure2 from the collision queue.

Example:
CQ_clear_figure_pair("cube1","cube2");

CQ_clear_intra_figure(figure)

figure : A figure name string.

This command removes all checks between pairs of segments in figure from the collision queue.

Example:
CQ_clear_intra_figure("robot");

See Also:
CQ_clear_current_figure

CQ_clear_queue() 

This command removes all checks from the collision queue.

Example:
CQ_clear_queue();

CQ_clear_segment(segment)

figure : A segment name string.

This command removes all checks involving segment from the command queue.

Example:
CQ_clear_segment("cube1.cube");

CQ_clear_segment_pair(segment1, segment2)

segment1 : A segment name string.
segment2 : A segment name string.

This command removes all checks between segment1 and segment2 from the collision queue.

Example:
CQ_clear_segment_pair("cube1.cube","cube2.cube");
window: A window name string.
This command will bring up a potentiometer window to allow the user to interactively adjust the clipping planes in window.

Example:
adjust.clipping.planes("jack.window");

See Also:
reset.clipping.planes

adjust.joint (joint, position)
joint: A joint name string.
position: A vector which specifies the relative position and orientation of the two segments of the joint. The number, type, and ordering of values in the vector must match the joint type.

This command will set the position and orientation of joint to position.

Example:
adjust.joint("human5.left elbow", (118.49deg));

See Also:
create.joint.between.segments, create.joint.between.sites

apply.deformation (segment)
segment: A segment name string.
This command applies an existing deformation to segment.

Example:
apply.deformation("cube.cube");

See Also:
create.segment.deformation

apropos (string)
string: An arbitrary string.
This command puts a list of all command names containing string into the active jack window.

Example:
apropos("figure");

See Also:
help.subject, help.command, browse

attach.figure (figure, site)
create.timed.attachment (figure, start, end, site)
figure: A figure name string.
site: A site name string.
start: A float indicating start time.
end: A float indicating end time.
The command create.attachment attaches figure to site. If site is subsequently moved, figure will move automatically to maintain a fixed position and orientation relative to site.
The command create.timed.attachment creates an attachment between figure and site during the indicated time period. The attachment will not have any effect before or after the period.

Example:
attach.figure("pyramid","cube.cube.base");

See Also:
create.joint.between.segments, unattach.figure

attach.view.to.site (window, site)
window : A window name string.
site : A site name string.
This command will move the view in window to the position and orientation of site. If site is subsequently moved, the view will move with it. This command does not affect the position of the camera.

Example:
attach.view.to.site("Jack window","cube.cube.base");

See Also:
change.view, reset.view.to.camera

behavior.info (human)
human : A name string for a human figure.
This command puts a list of the current behaviors assigned to human to the Jack Message Window.

Example:
behavior.info("human5.fig");

bend.torso (human, top, bottom, initiator, resistor, resistance, interpolate, values)
human : A name string for a human figure.
top : The upper limit of joints in the spine that are free to rotate. This must be an integer from 1 to 17.
bottom : The lower limit of joints in the spine that are free to rotate. This must be an integer from 1 to 17.
initiator : The joint that will move rotate most. This must be an integer from 1 to 17.
resistor : The joint that will rotate least. This must be an integer from 1 to 17.
resistance : The resistance to rotation. This must be a value 0.0
interpolate : A string "yes" or "no" which controls whether Jack will interpolate through the resting position.
values : An angle vector specifying the flexion, axial, and lateral bend to be applied to the torso.
This command will apply a bend to the torso of human. Arguments control how the bend will be distributed between the joints of the spine.

Example:
bend.torso("human5.fig",1,17,1,50.00,"yes", (19.70deg,-16.51deg,-11.95deg));

See Also:
set.torso.control

bevel.edge (edge, resolution, back, in)
edge : An edge name string.
resolution : The number of points to use to describe the resulting edge.
back : The distance from the original edge to the beginning of curvature of the resulting edge.
in : The distance from the original edge to the center of curvature of the resulting edge.
This command will round the edge of a segment.

Example:
bevel.edge("cube.cube.1",6.25.00,10.36);

See Also:
bevel.node

bevel.node (node, resolution, back, in)
node : A node name string.
resolution : The number of points to use to describe the resulting edges.
back : The distance from the original node to the beginning of curvature of the resulting edges.
in : The distance from the original node to the center of curvature of the resulting edges.
This command will round the edges that include node.

Example:
bevel.node("cube.cube.1",6.25.00,10.36);

See Also:
bevel-edge

bind.command.to.key (command, key)
command : A command name string.
key : A key indicator.
This command makes key a shortcut key for command. The binding lasts for the duration of the Jack session. For permanent key bindings, this command can be used in the user's .jack.jcl initialization file.

Example:
bind.command.to.key("delete figure",D);

browse () stop.browsing ()
The command browse puts Jack into browse mode. Commands chosen during browse mode will not be executed. Instead, the help window on the chosen command will be displayed.
The command stop.browsing returns Jack to normal mode.

Example:
browse();

See Also:
apropos, help.by.subject, help.by.command

center.view (window)
window : A window name string.
This command moves the view reference point in window to (0.00,100.00,0.00).

Example:
center.view("jack.window");
See Also:

clear.frames()

dol: A joint name string.
dof: A degrees of freedom expression string. This can only include one degree of freedom.
rate: The motion rate in frames/cycle.
type: The motion type string. Either "constant" or "reciprocal".

The command motorize-joint creates a motor on joint. The command change.motor changes the parameters of an existing motor on joint.

Example:

motorize-joint("chain.joint2","R(z)",30,"constant");

change.view (window,xform, [depth])

window: A window name string.
xform: A transform which places the camera site in global coordinates.

depth: A float which places the view reference point this far away from the camera site, down the negative z axis.

This command will move the camera in window to the specified position. The previous position is lost.

depth is a derived argument. It can be omitted unless there is a need to explicitly reposition the view reference point.

Example:

change.view("jack.window",xyz(-29deg,40deg,20deg) trans(259cm,247cm,260cm));

See Also:
push.view, pop.view, move.view

clear.frames() This command will clear all computed animation frames from memory.

Example:

clear.frames();

clear.trace (trace)
delete.trace (trace)

trace: A trace name string.

The command delete.trace deletes trace and un-traces the segment or site that made the trace. The command clear.trace deletes trace but does not un-trace the segment or site.

Example:

clear.trace("trace1");

See Also:
untrace.segment, untrace.site

close.hand (mode, human, side, segment)

mode: A mode string. One of "disk", "power", "precision", "small disk", "tripod".

human: A human figure name string.

side: "left" or "right".

segment: A segment name string.
This command will cause human to grasp segment with its side hand. The type of grasp is determined by mode.

Example:
close.hand("power","human5","right","chain.arm2");
collapse.all.motion.groups()

This command collapses all motion groups so that the constituent motions of each appear as a single item in the animation window.

Example:
collapse.all.motion.groups

See Also:
expand.all.motion.groups, collapse.motion.group

collapse.motion.group()

This command collapses the current motion group so that the constituent motions appear as a single item in the animation window.

The user must select the current motion group interactively before issuing this command.

Example:
collapse.motion.group();

See Also:
expand.motion.group, collapse.all.motion.groups

concave.off()
concave.on()

This command turns concave face rendering [off/on] in the current environment.

Example:
concave.off();

constraint.info( constraint)
constraint : A constraint name string.

This command shows the parameters of constraint in the Jack Message Window.

Example:
constraint.info("cube pyramid");

See Also:
change.constraint

create.animation.window()
This command brings up a new Jack Animation Window.

Example:
create.animation.window();

create.arm.motion( human, start, end, velocity, side, reference, end.xform, relative, [relref], weight, weight.funct)

human : A human figure name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.
side : A string "left" or "right", indicating which arm is to be moved.
reference : A string from "attached object", "forearm", "palm" indicating the part of the arm that will act as the end effector.
end.xform : A transform indicating the end position and orientation of reference.
relative : A string "yes" or "no" indicating whether end.xform is given relative to another segment or given globally.
relref : A segment name string indicating the segment that end.xform is given relative to. This argument is only included if relative is "yes".

weight : A float indicating the relative weight to be given to this motion if it conflicts with simultaneous motions.
weight.funct : A string from "accelerate", "decay", "constant", "ease in/ease out", indicating the change in weight over the course of the motion.

This command will create an arm motion. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create.arm.motion("human5",1sec,2sec,"constant","left", "forearm",xyz(134.47deg,-29.36deg,-124.1Odeg)
trans(30.22cm,159.27cm,219.49cm),"no",2.00,"constant");

See Also:
go, move.arm

cREATE.CAMERA.MOTION( window, start, end, velocity, end.xform, [depth])

window : A window name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decay", "ease in/ease out" indicating the change in velocity over the course of the motion.
end.xform : A transform indicating the end position and orientation of the camera in global coordinates.
depth : A float which places the view reference point this far away from the camera site, down the negative z axis.

This command creates a camera motion in window. When this command is executed, Jack will open an animation window if none is open.
depth is a derived argument. It can be omitted unless there is a need to explicitly reposition the view reference point.
The command go will execute the motion.

Example:
create.camera.motion("jack window",0sec,1sec, "ease in/ease out",xyz(-91.63deg,-1.10deg,-145.96deg)
trans(-89.15cm,135.24cm,-28.59cm));

See Also:
go, change.view

create.center.of.mass.motion( human, start, end, velocity, type, reference, end.xform, relative , [relref], weight, weight.funct)
human : A human figure name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from “accelerate”, “constant”, “ease in/ease out” indicating the change in velocity over the course of the motion.
type : A string “between feet” or “location”, indicating whether the end position will be specified relative to the feet or globally.
end_xform : A transform indicating the end position and orientation of the center of mass.
weight : A float indicating the relative weight to be given to this motion if it conflicts with simultaneous motions.
weight_funct : A string from “accelerate”, “constant”, “decay”, “ease in/ease out”, indicating the change in weight over the course of the motion.

This command will create center of mass motion. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create.center_of_mass_motion(“human”, 1sec, 2sec, “constant”, “location”, xyz(13.4deg, -29.6deg, -1.1deg), trans(30.22cm, 159.27cm, 219.49cm), 2.00, “constant”);

See Also:
go, move.center_of_mass

create.constraint (name, goal_type, [goal], end_eff_type, end_eff, orient_rel, [aim_vect], [pos_rel], [pos_orient_weight], [orient_offset], [pos_offset], [end_vector], [goal_vector], [line_dir], [normal], starting_joint, weight)

change.constraint (name, goal_type, [goal], end_eff_type, end_eff, orient_rel, [aim_vect], [pos_rel], [pos_orient_weight], [orient_offset], [pos_offset], [end_vector], [goal_vector], [line_dir], [normal], starting_joint, weight)

name : A constraint name string.
goal.type : A string from “closest node”, “face”, “hold”, “node”, “relative transform”, “site”, “transform”, indicating the type of goal.
goal : The goal of the constraint. The type is determined by goal.type. If goal.type = “closest.node”, then goal is a segment name string. If goal.type = “hold”, then this parameter is omitted.
end_eff.type : A string from “closest node”, “node”, “site”, indicating the type of end effector.
end_eff : A name string of type end_eff_type, indicating the end effector.
orient_rel : A string from “aim”, “align_direction”, “align.frame”, “none”, “planar.direction”, “view”, indicating the type of positional relationship to maintain between the end effector and the goal.
aim_vect : A unit vector indicating the vector from the end effector to aim at the goal. This parameter is only included when orient_rel = “aim”.
pos.rel : A string from “limit.spring”, “none”, “point_to_line”, “point_to_plane”, “point_to_point”, “rest.angle”, indicating the type of positional relationship to maintain between the end effector and the goal. This parameter is omitted when orient_rel = “aim” or “view”.
pos_orient_weight : A float between 0.00 and 1.00 indicating the relative weight to be given to the position and orientation relationships. 0.00 gives all weight to position; 1.00 gives all weight to orientation. This parameter is omitted when pos.rel or orient.rel = “none”.
orient_offset : A string “yes” or “no”, indicating whether or not to offset the goal by the current orientational offset. If so, then the actual goal will be rotated from goal as the end effector is currently rotated from goal. This parameter is only included when orient.rel = “align.frame” or “align_direction”, or “planar_direction”.
pos_offset : A string “yes” or “no”, indicating whether or not to offset the goal by the current positional offset. If so, then the actual goal will be offset from goal as the end effector is currently offset from goal. This parameter is only included when pos.rel = “point_to_point”, “limit.spring”, “point_to_line”, “point_to_plane”, or “rest.angle”. (However, it is omitted when pos.rel = “point_to_line” and goal.type = “face”.)
end_vector : A unit vector indicating the direction from the end effector to use in solving the constraint. This parameter is only included when orient.rel = “align_direction” or “align_plane”.
goal_vector : A unit vector indicating the direction from the goal to be used in solving the constraint. This parameter is only included when orient.rel = “align_direction” or “align_plane”.
line_dir : A unit vector indicating the direction from the goal to be used in solving the constraint. This parameter is only included when pos.rel = “point_to_line”.
normal : A unit vector indicating the normal to the plane to be used in solving the constraint. This parameter is only included when pos.rel = “point_to_line”.
starting_joint : A joint name string indicating the joint closest to the root of the figure containing end_eff to be adjusted in solving the constraint.
weight : The relative weight to give to the constraint when it conflicts with other constraints.

The command create.constraint creates a constraint named name between end_eff and goal. The command change.constraint changes the constraint named name.

Example:
create.constraint(“COO”, “site”, “cube.cube_base”, “site”, “pointer.cube.tip”, “none”, “point_to_point”, “no”, “pyramid.cube”, 1.00);

See Also:
create rooting.constraint, display.constraints[on]off, delete constraint

create.contour.figure ([figure])

figure : A figure name string.

This command creates a complex human figure as defined in /pkg/jack/jacklib5/contour/contour.fig. The figure is created with the name figure. If the name is omitted, Jack will assign a unique name.
Example:
create contour figure ("steve");

See Also:
create human figure

create eye motion (human, start, end, velocity, side, site)
human : A human figure name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.
site : A site name string towards which the eyes of human will aim after the motion.

This command will create an eye motion. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create eye motion ("human", 1 sec, 2 sec, "constant", "pyramid.top");

See Also:
go, move eyes

create face (nodelist)
nodelist : A series of node name strings separated by commas.

This command creates a face connecting the nodes in nodelist. The new face is added to the segment containing the nodes in nodelist. All nodes must belong to the same segment. To assure that the proper side of the face is on the outside, the nodes should be listed in the order of a counterclockwise traversal.

Example:
create face ("cube.1", "cube.2", "cube.5");

create figure joint motion (figure, start, end, velocity, joint vectors list)
figure : A figure name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.
joint vectors list : A list of joint position vectors separated by commas indicating the final position of each joint in the figure. The order of the vectors must match the order in which the joints are written in the peabody file defining the figure.

This command will create a figure joint motion. Over the course of the animation, the joints in figure will move to the positions specified in joint vectors list. This command is similar to create joint chain motion, except that command operates on a subset of the joints in the figure.

When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create joint figure motion ("robot", 1 sec, 2 sec, "constant", (34.7 deg), (12.2 deg, 33.6 deg, 5.3 cm));

See Also:
go, create joint motion, create figure motion, create joint chain motion

create figure motion (figure, start, end, velocity, end xform)
figure : A figure name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.
end xform : A transform indicating the end position and orientation of figure.

This command will create a figure motion. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create figure motion ("cube", 1 sec, 2 sec, "constant", xyz(134.47 deg, -29.36 deg, -12.41 deg));

See Also:
go, move figure, create figure joint motion, create figure path motion

Bugs:

create finger motion (human, side, start, end, velocity, vector list)
human : A human figure name string.
side : A string "left" or "right", indicating which hand is to be moved.

Example:
start: A float indicating start time.

end: A float indicating end time.

trans:xyz: A string from “accelerate”, “constant”, “decelerate”, “ease in/ease out” indicating the change in velocity over the course of the motion.

vectorlist: A series of joint position vectors separated by commas, indicating the end positions of the joints in the fingers. There are a total of fifteen joints in the fingers (including the thumb.) Each digit has three joints. The joint at the base of each digit has two rotational degrees of freedom. The other two joints in each digit have one DOF each. The vectors are ordered in the list starting at the base of the index finger, moving to the end of that finger, then to the base of the next finger. The three joints of the thumb are given last.

This command will create a finger motion. Over the course of the motion, the fingers will reposition to the positions indicated by the vectorlist. Note that it is necessary to include all final joint positions, whether they change or not.

When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:

create.finger.motion("human5","left",1sec,2sec,"constant", (22.50deg,48.73deg),(76.96deg),(76.64deg), (0.0deg,15.46deg),(27.64deg),(0.0deg,15.46deg), (27.64deg),(18.04deg),(0.0deg,9.63deg),(46.00deg), (21.66deg),(0.00deg,50.70deg),(0.00deg),(3.02deg), (0.00deg,45.00deg),(33.86deg),(31.25deg));

See Also:
go, grasp, reachandshape.hand, create.hand.site

create.finger.motion (human, start, end, velocity, side, xform, height, trans:vs:xyz, weight, weight:funct)

human: A human figure name string.

start: A float indicating start time.

end: A float indicating end time.

velocity: A string from “accelerate”, “constant”, “decelerate”, “ease in/ease out” indicating the change in velocity over the course of the motion.

side: A string “left” or “right”, indicating which foot is to be moved.

xform: A transform indicating the final position and orientation of the toes.

height: A float indicating the maximum height (relative to the ground) of the arc through which the toes will move over the course of the motion.

trans:vs:xyz: A float 0.00; 1.00 indicating the relative weight to be given to foot position vs. orientation if they are in conflict during the motion. 0.00 gives full weight to position; 1.00 gives full weight to orientation.

weight: A float indicating the relative weight to be given to this motion if it conflicts with simultaneous motions.

weight:funct: A string from “accelerate”, “constant”, “decay”, “ease in/ease out”, indicating the change in weight over the course of the motion.

This command will create a foot motion. Over the course of the motion, the foot will move to the position and orientation indicated by xform.

Note that unlike most motions, which follow a straight line, foot motions travel in an arc. The maximum height of the arc is controlled by height.

When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:

create.foot.motion("human5",1sec,2sec,"constant", "left",xyz(-0.39deg,0.30deg,-28deg) trans(-0.01cm,.07cm,54.19cm),10.00,.50,10.00, "constant");

See Also:
go, move.foot, create.heel.motion, create.timed.foot.control

create.fractal ()
fractal: (fractal)
fractal:Above.sea.power:law ()
fractal:Below.sea.power:law ()
fractal:Max:level ()
fractal:Re:seed ()
fractal:Sea:level ()
fractal:Standard:deviation ()
fractal:Dimension ()
write:fractals.psrf (fractal)
fractal: A segment name string indicating the fractal to be written.

The command create.fractal allows the user to interactively define a fractal.

The commands fractal:* allow the user to interactively modify a parameter of the current fractal.

The command fractal:seed automatically selects a new random seed for the current fractal.

The command write:fractals.psrf allow the user to specify a file, then writes fractal to that file in psurf format.

Example:

create.fractal();

See Also:
select.current.fractal

create.ghost (figure) delete.ghost ()

figure: A figure name string.

The command create.ghost creates a ghost of figure.

The command delete.ghost allows the user to select a ghost, then deletes the ghost.

Example:

create.ghost("cube.cube");

create.hand.site (human, side, segment, name)

human: A figure name string of a human figure.

side: A string “left” or “right”.

segment: A segment name string. The site will be placed on this segment.

name: A name string for the new site.
This command creates a site named name on segment. The new site will initially be located at the base site of the palm segment of the side hand of human.

This command is typically used to create a site on an object to be grasped. You move the hand into a position from which to grasp the object, then execute this command. The site can then be used as input to the command grasp.

Example:
```
create.hand.site("human5", "left", "hammer shaft", "hammer grasp position");
```

See Also:

grasp

```
create.heel.motion (human, start, end, velocity, side, height, weight, weight funct)
```

human : A human figure name string.

start : A float indicating start time.

end : A float indicating end time.

velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.

side : A string "left" or "right", indicating which foot is to be moved.

height : A float indicating the maximum height (relative to the ground) to which the heel will raise over the course of the motion.

weight : A float indicating the relative weight to be given to this motion if it conflicts with simultaneous motions.

weight funct : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in weight over the course of the motion.

This command will create a heel motion. Over the course of the motion, the heel will raise to the height indicated by height. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
```
create.heel.motion("human5", "left", "hammer shaft", "hammer grasp position", 1.0, 2.0, "constant");
```

See Also:

go, move foot, create.foot motion, create. delay.foot.control

```
create.human.figure ([name])
```

name : A figure name string.

This command will create a human figure as defined in the file human5.fig. If name is included, the figure will be created with that name. Otherwise Jack will create a unique name for the figure.

Example:
```
create.human.figure();
```

See Also:

create.contour.figure, create.stick.figure

```
create.jcl.motion (start, end, jcl1, jcl2, jcl3)
```

start : A float indicating the start time of the motion.

end : A float indicating the end time of the motion.

jcl1 : A jcl statement string.

jcl2 : A jcl statement string.

jcl3 : A jcl statement string.

This command sets up a series of jcl statements to be executed during the course of animation. jcl1 is executed at start. jcl2 is executed before each frame up to end is computed. jcl3 is executed at end.

Any quote characters that are part of the jcl statements must be preceded by backslashes.

When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation. Note that the commands are not executed when the computed animation is viewed using play frames.

Example:
```
create.jcl.motion("fig1.fig2", "" gaping segment.new attribute("sphere":sphere)", "");
```

See Also:

jcl, go, move foot, create.foot motion, create. delay.foot.control

```
create.joint.between.segments (segment1, segment2, xform, type)
```

segment1 : A segment name string.

segment2 : A segment name string.

xform : A transform placing the joint relative to the root of the figure containing segment1.

type : A degrees of freedom expression string.

This command creates a joint between segment1 and segment2 with degrees of freedom indicated by type. A new site is added to each of the segments. The new sites are located at the position indicated by xform. The new joint connects these two sites.

The name of the new joint is the name of the figure that contains segment1, followed by a period, followed by the name of the figure that contained segment2, e.g. "fig1.fig2". Creating a joint between segments causes the two figures containing the segments to be combined into a single figure. The root of the new figure is the root of the figure that contained segment1. The name of the new figure is the name of the figure that contained segment1.

Example:
```
create.joint.between.segments("pyramid":pyramid", "cube":cube", trans(20.00cm,20.00cm,0.00cm)
xyz(27.4deg,0.00deg,0.00deg),"R(z)"T(z)");
```

See Also:

create.joint.between.sites

```
create.joint.between.sites (site1, site2, type)
```

site1 : A site name string.

site2 : A site name string.

type : A degrees of freedom expression string.

This command creates a joint between the segment containing site1 and the segment containing site2.

The name of the new joint is the name of the figure that contains site1, followed by a period, followed by the name of the figure that contains site2, e.g. "fig1.fig2". The figure containing site2 will be moved so that site1 and site2 initially have the same position and orientation.
Creating a joint between segments causes the two figures containing the segments to be combined into a single figure. The root of the new figure is the root of the figure that contained sitel. The name of the new figure is the name of the figure that contained sitel.

Example:
create.joint.between.sites("redfig.redseg.hingesite", "greenfig.greenseg.hingesite","\$(z)=x(z)"};

See Also:
create.site.between.segments

create.joint.chain.motion (segmentfirst, segmentlast, start, end, velocity, vectorlist)

segmentfirst : A segment name string indicating the first segment of the joint chain.
segmentlast : A segment name string indicating the last segment of the joint chain.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.
vectorlist : A list of joint position vectors separated by commas indicating the final position of each joint in the joint chain. These are ordered starting with the joint attached to segmentfirst and following down the joint chain towards segmentlast.

This command creates a motion across a joint chain. A joint chain is a progression of joints connecting two segments in a figure. Over the course of the motion, the joints move towards the positions indicated by vectorlist. The command is similar to create.figure.joint.motion, except that command creates a motion across all the joints of a figure. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create.joint.chain.motion("redfigseg2", "redfigseg7", 0sec, 1sec, "constant", (270.50deg));

See Also:
go, create.figure.motion

create.joint.motion (joint, start, end, velocity, vector)

joint : A joint name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.
vector : A joint position vector indicating the final position of joint.

This command creates a joint motion. Over the course of the motion, joint moves towards the position indicated by vector. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create.joint.motion("redfig.bluefig", 0sec, 1sec, "constant", (270.50deg));

See Also:
go

create.light ()

This command creates a light source at the world root. The light source segment name is set to "lightsource". The lightsource figure name is also set to "lightsource". If there is already a figure with this name, it is named "lightsourceNN" where NN is the lowest integer that will make the name unique.

Example:
create.light();

create.light.motion (segment, start, end, velocity, start.color, end.color)

segment : A segment name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decay", "ease in/ease out" indicating the change in velocity over the course of the motion.
start.color : A color triple indicating the starting color for the light.
end.color : A color triple indicating the end color for the light.

This command creates a "motion" of the color of segment over the indicated time period. I.e. the color of segment will change from start.color to end.color over time. To create an actual motion (i.e. a position change) of a light, use the command create.figure.motion. When this command is executed, Jack will open an animation window if none is open. The command go will execute the motion.

Example:
create.light.motion("lightsource.lightsource", 0sec, 1sec, "ease in/ease out", (0,0,0),(1,1,1));

See Also:
go, create.figure.motion

create.motion.group (name)

name : A motion group name string.

This command creates a motion group named name.

Example:
create.motion.group("group1");

create.node (segment)

segment : A segment name string.

This command allows the user to interactively add a node to segment. There is not a way to programmatically add a node to a segment.

Example:
create.node("cube.cube");
create ordinary window (windowrect)
create shaded window (windowrect)
create wireframe window (windowrect)

windowrect : A screen rectangle.

These commands create a Jack window of the appropriate type in the rectangle windowrect.

Example:
create ordinary window((0,0,200,400));

create path (name, xform list)

name : A figure name string.

xform list : A list of transforms. Each transform indicates the position and orientation of a point on the path relative to the first point of the path. Each transform must be followed by a string "yes" or "no", indicating whether more transforms will follow, i.e. only the last transform will be followed by "no".

This command creates a path figure named name. The segment name is set to name".paths".
The user must interactively place the first point of the path. The positions of the points on the path relative to the first point of the path are indicated by xform list.

Example:
create path("mypath", trans(119.20cm, 0.00cm, 0.00cm), "yes", trans(250.00cm, 0.00cm, 0.00cm), "no");

create path motion (path, start, end, direction)

path : A segment name string.

start : A float indicating start time.

end : A float indicating end time.

direction : A string "forward" or "backward" indicating the direction of travel along the path.

This command creates a motion of the site "point" along path. This is typically useful for creating constraints that change over time.
The command create figure path motion can be used to create a motion of a figure along a path.

Example:
create path motion("mypath", 0sec, 1sec, "forward");

See Also:
create path motion, create path

create pelvis motion (human, start, end, velocity, xform, weight, weight funct)

human : A human figure name string.

start : A float indicating start time.

end : A float indicating end time.

velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.

xform : A transform indicating the end position and orientation of the pelvis.

weight : A float indicating the relative weight to be given to this motion if it conflicts with simultaneous motions.

weight funct : A string from "accelerate", "constant", "decelerate", "ease in/ease out", indicating the change in weight over the course of the motion.

This command will create a pelvis motion of the figure human. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create pelvis motion("human", 1sec, 2sec, "constant", xyz(134.47deg, -29.36deg, -124.10deg)
trans(30.22cm, 159.27cm, 219.49cm), 2.00, "constant");

See Also:
go, rotate pelvis

create preview movie (start, end, name, xdim, ydim)

start : Integer indicating first frame of movie.

end : Integer indicating last frame of movie.

name : Name of movie to be produced.

xdim : Integer indicating x dimension of movie.

ydim : Integer indicating y dimension of movie.

This command creates a movie based on the current computed frames. The movie is written to a file called name in the current movie directory.
The current movie directory can be set using set movie file directory.
This command will also open a movie viewing window.

Example:
create preview movie(0, 20, "jack.movie", 180, 120);

See Also:
set movie file directory

create rooting constraint (name, goal type, [goal], end eff type, end eff, orient rel, [aim vect], [pos rel], [pos orient weight], [orient offset], [pos offset], [end vect], [goal vect], [line dir], [normal], weight)

name : A constraint name string.

goal type : A string from "closest node", "face", "hold", "node", "relative transform", "site", "transform", indicating the type of goal.

goal : The goal of the constraint. The type is determined by goal type. If goal type = "closest node", then goal is a segment name string. If goal type = "hold", then this parameter is omitted.

end eff type : A string from "closest node", "node", "site", indicating the type of end effector.

end eff : A name string of type end eff type, indicating the end effector.

orient rel : A string from "aim", "align direction", "align frame", "none", "planar direction", "view", indicating the type of orientation relationship to maintain between the end effector and the goal.

aim vect : A unit vector indicating the vector from the end effector to aim at the goal. This parameter is only included when orient rel = "aim".
This command creates a rooting constraint named name between end-eff and goal. This parameter is omitted when \texttt{orient.rel} = \textit{aim} or \textit{view}.

\texttt{pos.orient.weight} : A float between 0.00 and 1.00 indicating the relative weight to be given to the position and orientation relationships. 0.00 gives all weight to position; 1.00 gives all weight to orientation. This parameter is omitted when \texttt{pos.rel} or \texttt{orient.rel} = \textit{none}.

\texttt{orient.offset} : A string \textit{"yes"} or \textit{"no"}, indicating whether or not to offset the goal by the current orientational offset. If so, then the actual goal will be rotated from goal as the end effector is currently rotated from goal. This parameter is only included when \texttt{orient.rel} = \textit{"align.frame"} or \textit{"align.direction"}, or \textit{"planar.direction"}.

\texttt{pos.offset} : A string \textit{"yes"} or \textit{"no"}, indicating whether or not to offset the goal by the current positional offset. If so, then the actual goal will be offset from goal as the end effector is currently offset from goal. This parameter is only included when \texttt{pos.rel} = \textit{"point.to.point"}, \textit{"limit.spring"}, \textit{"point.to.line"}, \textit{"point.to-plane"}, or \textit{"rest.angle"}. (However, it is omitted when \texttt{pos.rel} = \textit{"point.to-plane"} and \texttt{goal.type} = \textit{"face"}.)

\texttt{end.vector} : A unit vector indicating the direction from the end effector to use in solving the constraint. This parameter is only included when \texttt{orient.rel} = \textit{"align.direction"} or \textit{"align.plane"}.

\texttt{goal.vector} : A unit vector indicating the direction from the goal to be used in solving the constraint. This parameter is only included when \texttt{orient.rel} = \textit{"align.direction"} or \textit{"align.plane"}.

\texttt{line.dir} : A unit vector indicating the direction from the goal to be used in solving the constraint. This parameter is only included when \texttt{pos.rel} = \textit{"point.to.line"}.

\texttt{normal} : A unit vector indicating the normal to the plane to be used in solving the constraint. This parameter is only included when \texttt{pos.rel} = \textit{"point.to-plane"}.

\texttt{weight} : The relative weight to give to the constraint when it conflicts with other constraints.

This command creates a rooting constraint named name between \texttt{end.eff} and \texttt{goal}.

\textbf{Example:}
\begin{verbatim}
create.rooting.constraint("CO", "site", "cube1.cube.base", 
"site", "cube2.cube.base", "none", "point.to.point", 
"no", 1.00);
\end{verbatim}

\textbf{See Also:}
\begin{verbatim}
[create/change].constraint, display.constraints.[on/off] 
\end{verbatim}

\texttt{create.ruler (segment1, segment2)}

\texttt{segment1} : A segment name string.

\texttt{segment2} : A segment name string.

This command will create a ruler between two segments. Note that the user must interactively place a reference point on each segment when this command is executed. Use \texttt{create.ruler.between.sites} if this is a problem.

\textbf{Example:}
\begin{verbatim}
create.ruler("cube1.cube", "cube2.cube");
\end{verbatim}
end : A float indicating end time.

name : A constraint name string.

goal_type : A string from "closest node", "face", "hold", "node", "relative transform", "site", "transform", indicating the type of goal.

goal : The goal of the constraint. The type is determined by goal_type. If goal_type = "closest node", then goal is a segment name string. If goal_type = "hold", then this parameter is omitted.

end_eff_type : A string from "closest node", "node", "site", indicating the type of end effector.

eff : A name string of the end effector.

orient_rel : A string from "aim", "align direction", "align frame", "none", "planar direction", "view", indicating the type of orientation relationship to maintain between the end effector and the goal.

aim_vec : A unit vector indicating the vector from the end effector to aim at the goal. This parameter is only included when orient_rel = "aim" or "view".

pos_rel : A string from "limit.spring", "none", "point.to.line", "point.to.plane", "point.to.point", "rest.angle", indicating the type of positional relationship to maintain between the end effector and the goal. This parameter is omitted when orient_rel = "aim" or "view".

pos_orient_weight : A float between 0.0 and 1.0 indicating the relative weight to give to position and orientation relationships. 0.0 gives all weight to position; 1.00 gives all weight to orientation. This parameter is omitted when pos_rel or orient_rel = "none".

orient_offset : A string "yes" or "no", indicating whether or not to offset the goal by the current orientational offset. If so, then the actual goal will be rotated from goal as the end effector is currently rotated from goal. This parameter is only included when orient_rel = "align frame" or "align direction", or "planar direction".

pos_offset : A string "yes" or "no", indicating whether or not to offset the goal by the current positional offset. If so, then the actual goal will be offset from goal as the end effector is currently offset from goal. This parameter is only included when pos_rel = "point.to.line", "limit.spring", "point.to.plane", "point.to.point", or "rest.angle". (However, it is omitted when pos_rel = "point.to.line" and goal_type = "face".)

dep : A unit vector indicating the direction from the end effector to use in solving the constraint. This parameter is only included when orient_rel = "align direction" or "align plane".

goal_vec : A unit vector indicating the direction from the goal to be used in solving the constraint. This parameter is only included when orient_rel = "align direction" or "align plane".

line_dir : A unit vector indicating the direction from the goal to be used in solving the constraint. This parameter is only included when pos_rel = "point.to.line".

normal : A unit vector indicating the normal to the plane to be used in solving the constraint. This parameter is only included when pos_rel = "point.to.plane".

starting joint : A joint name string indicating the joint closest to the root of the figure containing end_eff to be adjusted in solving the constraint.

weight : The relative weight to give to the constraint when it conflicts with other constraints.

This command creates a timed constraint named name between end_eff and goal.

Example:
create.timed.constraint(0.00, 1.00, "site", "cube.cube.base", "site", "pointer.cube.tip", "none", "point.to.point", "no", "pyramid.cube", 1.00);

See Also:
[create[change]constraint, display.constraints.[on|off]

delete.constraint

create.torsomotion (human, start, end, velocity, top, bottom, initiator, resistor, resistance, interpolate, values)

human : A name string for a human figure.

start : A float indicating start time.

end : A float indicating end time.

velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.

top : The upper limit of joints in the spine that are free to rotate. This must be an integer from 1 - 17.

bottom : The lower limit of joints in the spine that are free to rotate. This must be an integer from 1 - 17.

initiator : The joint that will rotate most. This must be an integer from 1 - 17.

resistor : The joint that will rotate least. This must be an integer from 1 - 17.

resistance : The resistance to rotation. This must be a value 0.0

interpolate : A string "yes" or "no" which controls whether Jack will interpolate through the resting position.

values : An angle vector specifying the final flexion, axial, and lateral bend to be applied to the torso.

This command will apply a bend to the torso of human over the specified time period. Arguments control how the bend will be distributed between the joints of the spine. When this command is executed, Jack will open an animation window if there is none open. The command go will execute the animation.

Example:
create.torsomotion("human5.fig", 0.00, 1.00, "constant", 1.17, 17, 1.50, 0.0, "yes", (19.70deg, -16.51deg, -11.95deg));

See Also:
goto, move.torso

create.view.cones (human, length, angle)

human : A human figure name string.

length : A float indicating the length of the view cones.

angle : A float indicating the angle of the view cones.

This command creates view cones on human.

Example:
create.view.cones("human5", 200cm, 40deg);
create_x_window (windowrect)
create_y_window (windowrect)
create_z_window (windowrect)

windowrect : A screen rectangle.

This command creates a new Jack window in windowrect. The new window presents an orthographic projection along the \([xlylz]\) axis. The new window is initially an ordinary window (as opposed to shaded or wireframe).

Example:
create_x_window((345,1132,132,838));

delete_all_constraints ()
This command deletes all constraints from the environment.

Example:
delete_all_constraints();

See Also:
delete_constraint , turn_constraint_off

delete_all_motors ()
This command deletes all motors from the current environment.

Example:
delete_all_motors();

See Also:
turn_motor_on , turn_motor_off , turn_all_motors_on , turn_all_motors_off

delete_constraint (constraint)
constraint : A constraint name string.

This command will delete constraint.

Example:
delete_constraint("CO");

See Also:
delete_all_constraints , turn_constraint_off

delete_edge (edge)
edge : An edge name string.

This command deletes edge and the faces that include it.

Example:
delete_edge("cube_cube.8");

delete_environment ([really])
really : A string "yes". If this parameter is omitted, Jack will query the user as to whether or not to delete the environment.

This command deletes the current environment.

Example:
delete_environment("yes");

delete_face (face)
face : A face name string.

This command deletes a face.

Example:
delete_face("cube_cube.3");

delete_figure (figure);
figure : A figure name string.

This command deletes a figure.

Example:
delete_figure("cube");

delete_ghost ()
This command allows the user to interactively choose a ghost to delete.

Example:
delete_ghost();

delete_intersecting_faces (face)
face : A face name string.

This command deletes all faces that intersect face.

Example:
delete_intersecting_faces("cube_cube.3");

delete_joint (joint)
joint : A joint name string.

This command deletes a joint.

Example:
delete_joint("cube0_cube");

delete_motion ()
This command will delete the current motion.

Example:
delete_motion();

delete_motion_group ()
This command deletes the current motion group. Note that this does not delete the constituent motions. The user must interactively select the current motion group before this command is issued.

Example:
delete_motion_group();

delete_motions ()
This command deletes all motions from the current environment.

Example:
delete_motions();

delete_motor (joint)
joint : A joint name string.

This command deletes the motor on joint.

Example:
delete_motor("cube_cube0");
See Also:
delete.allmotors

delete.node (node)
delete.vertex (node)

node : A node name string.
The command delete.node deletes node and any faces that include it.
The command delete.vertex will delete a node without deleting the faces that include it.

Example:
delete.node("cube.cube.3");

delete.nonrooting.constraints ()
This command deletes all constraints that are not rooting constraints from the current environment.

Example:
delete.nonrooting.constraints();

See Also:
delete.constraint

delete.path (path)
path : A path segment name string.
This command deletes a path segment.

Example:
delete.path("mycurve.paths");

delete.path.point (point)
point : A path point node string.
This command deletes point.

Example:
delete.path.point("mycurve.paths.pnt2");

delete.ruler ()
This command allows the user to interactively choose a ruler to delete.

Example:
delete.ruler();

delete.segment (segment)
segment : A segment name string.
This command deletes segment.

Example:
delete.segment("cube.cube");

delete.segment.deformation (segment)
segment : A segment name string.
This command removes the deformation lattice from segment.
It does not reset any deformations that have been applied to segment.

Example:
delete.segment.deformation("cube.cube");

delete.view.cones (human)

human : A human figure name string.
This command will delete the view cones from human.

Example:
delete.view.cones("human5");

disable.graphics ()
enable.graphics ()
These commands turn graphics drawing on and off.

Example:
disable.graphics();

disable.user.clipping.plane (plane)
enable.user.clipping.plane (plane)

plane : An integer 1 to 6 indicating the plane to enable or disable.

These commands disable and enable one of the user-defined clipping planes.

Example:
disable.user.clipping.plane(2);

display.constraints.off
display.constraints.on

Turns on or off display of constraints.

Example:
display.constraints.off();

divide.face.at.edges (face)
divide.face.at.vertices (face)

face : A face name string.

These commands divide face into several faces.

The command divide.face.at.edges divides the face by connecting the midpoints of the edges to the center of the face. New nodes are added at the midpoints of the edges and at the center of the face.

The command divide.face.at.vertices divides the face by connecting the vertices to the center of the face. New nodes are added at the midpoints of the edges and at the center of the face.

Example:
divide.face.at.edges("cube.cube.3");

draw.channels.off ()
draw.channels.on ()

These commands turn on and off the drawing of channels during animation. When drawing is on, Jack draws marks showing the subgoals pursued during the animation.

Example:
draw.channels.off();

See Also: enable.channels

draw.windows ()

This command causes a refresh of all windows. When it executes, it freezes Jack until the user types a key.

Example:
draw.windows();

draw.window()

draw.window()

draw.window()

edge.info (edge);

edge : An edge name string.

Puts info on edge to the Jack message window.

Example:
edge.info("cube.cube.3");

enable.all.channels ()
disable.all.channels ()
These commands enable and disable animation for all channels.

Example:

enable.all.channels();

See Also:
disable.channel

enable.channel (channel)
disable.channel (channel)

channel : A channel name string.

These commands enable or disable channel.

Example:
enable.channel("figure.cube");

See Also:
disable.all.channels

evaluate.constraints ()

This command forces an evaluation of all constraints, even if they have been turned off.

Example:
evaluate.constraints();

See Also:
turn.constraints.on/off

expand.all.motion.groups ()

This command expands all motion groups so that the constituent motions are displayed individually in the animation window.

Example:

expand.all.motion.groups();

See Also:
collapse.all.motion.groups , expand.motion.group

expand.motion.group ()

This command expands the current motion group so that constituent motions are displayed individually in the animation window.
The user must interactively select the current motion group before this command is issued.

Example:

expand.motion.group();

See Also:
collapse.motion.group , expand.all.motion.groups

extrude.edge (part , size)
extrude.face (part , size)
extrude.node (part , size)

part : An [edge|face|node] name string.
size : A float indicating length of extrusion base.
These commands allow the user to interactively extrude an edge, face, or node.

Example:
extrude.edge("cube.cube.3");

face.info(face)
face : A face name string.
This command puts info on face to the Jack Message Window.

Example:
face.info("cube.cube.3");

figure.center.of.mass(site)
site : A site name string.
This command designates site as the center of mass for the figure that contains it.

Example:
figure.center.of.mass("cube.cube.center");

figure.info(figure)
figure : A figure name string.
This command puts info on figure to the Jack Message Window.

Example:
figure.info("cube");

four.panel.screen()
This command tiles the screen with four windows: A standard Jack Window, and x,y, and z windows.

Example:
four.panel.screen();

freeze.joint(joint)
thaw.joint(joint)
joint : A joint name string.
These commands disallow and allow adjustments of joint.

Example:
freeze.joint("cube1.cube2");

freeze.window(window)
thaw.window(window)
window : A window name string.
These commands turn on and off updating for window.

Example:
freeze.window("jack.window");

give.csurf.new.attribute(face)
give.csurf.old.attribute(attribute, face)
face : A face name string.
attribute : An attribute name string.
The command give.csurf.new.attribute creates a new attribute and assigns it to face.
The command give.csurf.old.attribute assigns attribute to face.

Example:
give.csurf.old.attribute("cherryred","cube.cube.1");

Bugs:
The first time one of these commands is used on a face of a particular segment, the entire segment is given the new attribute.

give.face.new.attribute(face)
give.face.old.attribute(attribute, face)
face : A face name string.
attribute : An attribute name string.
The command give.face.new.attribute creates a new attribute and assigns it to face.
The command give.face.old.attribute assigns attribute to face.

Example:
give.face.old.attribute("cherryred","cube.cube.1");

give.segment.new.attribute(segment)
give.segment.old.attribute(attribute, segment)
segment : A segment name string.
attribute : An attribute name string.
The command give.segment.new.attribute creates a new attribute and assigns it to segment.
The command give.segment.old.attribute assigns attribute to segment.

Example:
give.segment.old.attribute("cherryred","cube.cube");

go()
This command animates the current set of motions. It clears any existing frames, sets time to zero, then advances time while computing, displaying, and storing each frame, continuing until the end of the last motion. The command start.time can be used to begin animation from a specific time. The frames are stored in memory, and can subsequently be replayed using the command play.frames.

Example:
go();

See Also:
play.frames, go.without.save, start.time, stop.time

go.without.save()
This command is identical to the command go, except that frames are not stored in memory as they are generated, so they cannot be replayed using the command play.frames.

Example:
go.without.save();

See Also:
go, start.time, stop.time
goto.time(time)
time: An integer indicating the time to go to. It may be in units of seconds("sec") or frames(default).

This command moves time to time. It does not cause any animation.

Example:
goto.time(15);

grasp (type, human, side, segment, locationtype, [site])

type: A string from "disk", "power", "precision", "small disk", "tripod", indicating the type of grip to use.

human: A human figure name string.

side: A string from "left" or "right", indicating which hand is to grasp.

segment: A segment name string.

locationtype: "calculated transform" or "predefined site"

site: A site name string. This argument is omitted when locationtype = "calculated transform".

This command moves the side hand of human such that it is grasping segment. The arguments locationtype and site determine the position to which the hand is moved before the fingers begin to close around segment.

Example:
grasp("power","human5","left","cube.cube","calculated transform");

See Also:
reach_and_shape_hand

help_by_subject (subject)

subject: A string.

This command brings up a help window on subject. If there is no help window for subject, the command brings up the first help window.

Example:
help_by_subject("joints");

See Also:
help_for_command

help_for_command (command)

command: A command name string.

This command brings up a help window on the command command. If there is no help window on command, this command brings up the first help window.

Example:
help_for_command("move figure");

See Also:
help_by_subject, apropos

initialize_figure_for_collision

This command has been replaced by the command CQ.clear.current.figure.

Example:
initialize_figure_for_collision

See Also:
CQ.clear.current.figure

install_file (filename)

filename: A filename string.

This command adds the file filename to the user's Jack installation file.

Example:
install_file("usr/jack/shared.env");

See Also:
show_installed_files, uninstall_file

interactive_reach (end_effector_type, end_effector, starting_joint)

end_effector_type: A string from "closest node", "node", "site".

end_effector: The type of this argument depends upon the value of end_effector.site.

If end_effector_type = "closest node", this argument is a segment name string, indicating the segment to move. The closest node to the goal is used as the end effector.

If end_effector_type = "node", this argument is a node name string.

If end_effector_type = "site", this argument is a site name string.

starting_joint: A joint name string, indicating the joint closest to the root that is to be adjusted to attain the goal.

This command allows the user to interactively adjust a figure by a goal site.

Example:
interactive_reach("site","chain.arm6.top","chain.joint1");

intersect_segments (segment1, segment2)

segment1: A segment name string.

segment2: A segment name string.

This command creates a segment that is the intersection between segment1 and segment2. The new segment is called "intersection.intersection". If there is already a figure "intersection", the name will be made unique by adding an integer to the end.

Example:
intersect_segments("cube1.cube","cube2.cube");

See Also:
union_segments, difference_segments

joint_info (joint)

joint: A joint name string.

This command puts info on joint to the Jack Message Window.

Example:
joint_info("cube.pyramid");
keyboard.command ()
This command prompts the user to enter a keyboard command.

Example:
keyboard.command();

make.background.shaded (window)
make.background.wireframe (window)

window : A window name string.
These commands make the background in window shaded or wireframe.

Example:
make.background.shaded("jack window");
make.background.wireframe("jack window");

Example:
makenindowardinary (window)
makesindowshaded (window)
makeainindowireframe (window)

window : A window name string.
These commands make window ordinary, shaded, or wireframe.

Example:
makenindowardinary("jack window");

Example:
make.everything.shaded (window)
make.everything.wireframe (window)

window : A window name string.
These commands make all figures in window shaded or wireframe.

Example:
make.everything.shaded("jack window");

Example:
make.segment.shaded (segment)
make.segment.wireframe (segment)

segment : A segment name string.
These commands make segment shaded or wireframe.

Example:
makesegmentshaded("cube .cube .27");

Example:
make.segment.transparent (segment)

segment : A segment name string.
This command makes segment transparent.

Example:
makesegment.transparent("cube .cube .27");

Example:
make.light.infinite (light)
make.light.local (light)

light : A light source name string.
These commands make a light source act as local to the scene or as located at an infinite distance. A local light source makes shading more accurate but less efficient.

Example:
make.light.infinite("lightsource .lightsource");

make.lights.invisible ()
make.lights.visible ()

These commands control whether lightsources are visible in the scene.

Example:
make.lights.invisible ()

make.segment.flat (segment)
make.segment.smooth (segment)

segment : A segment name string.
These commands turn phong shading off ("flat") and on ("smooth") for segment.

Example:
makesegment.flat("sphere");

See Also:
make.figure.[flat|solid]

make.segment.shaded (segment)
make.segment.wireframe (segment)

segment : A segment name string.
These commands make segment shaded or wireframe.

Example:
makesegment.shaded("cube .cube .27");

Example:
makesegment.transparent (segment)

segment : A segment name string.
This command makes segment transparent.

Example:
makesegment.transparent("cube .cube .27");

Example:
make.viewer.infinite (window)
make.viewer.local (window)

window : A window name string.
These commands make the viewpoint in window local to the scene or located at an infinite distance. A local viewpoint makes shading more accurate but less efficient.

Example:
makeviewer.infinite("jack window");

make.window.ordinary (window)
make.window.shaded (window)
make.window.wireframe (window)

window : A window name string.
These commands make window ordinary, shaded, or wireframe.

Example:
make.window.ordinary("jack window");
measure.anterior.arm (human)
measure.arm.span (human)
measure.biceps (human)
measure.buttock.knee (human)
measure.buttock.popliteal (human)
measure calf (human)
measure.chest (human)
measure.chest.depth (human)
measure.crotch (human)
measure.distance (human)
measure.eye.sitting (human)
measure.eye.standing (human)
measure.foot.breadth (human)
measure.foot.length (human)
measure.forearm.hand (human)
measure.functional (human)
measure.hand.breadth (human)
measure.hand.circum (human)
measure.hand.length (human)
measure.head.circum (human)
measure.head.length (human)
measure.hip (human)
measure.hip.sitting (human)
measure.hip.standing (human)
measure.interpupilary (human)
measure.interocular (human)
measure.knee.cap (human)
measure.neck (human)
measure.overhead (human)
measure.palm.length (human)
measure.popliteal (human)
measure.shoulder.breadth (human)
measure.shoulder.circum (human)
measure.shoulder.elbow (human)
measure.shoulder.length (human)
measure.shoulder.sitting (human)
measure.shoulder.standing (human)
measure.stature.sitting (human)
measure.stature.standing (human)
measure.thigh.clearance (human)
measure.vertical.trunk (human)
measure.waist (human)

side : A string "left" or "right", indicating which arm is to be moved.
reference : A string from "attached object", "forearm", "palm" indicating the part of the arm that will act as the end effector.
start.joint : A string from "shoulder" or "waist" indicating the joint at which to initiate the move.
end.xform : A transform indicating the end position and orientation of reference in global coordinates.

This command repositions human to put reference at end.xform.

Example:
move.arm( "human5", "left", "forearm", "shoulder", "xyz(144.67deg,-63.29deg,-64.03deg)"

move.center.of.mass (human, end.xform)

human : A human figure name string.
end.xform : A transform indicating the end position and orientation of the center of mass in global coordinates.

This command repositions human to put the center of mass at end.xform.

Example:
move.center.of.mass("human5", "xyz(144.67deg,-63.29deg,-64.03deg)"

move.deformation.lattice.edge ()
move.deformation.lattice.face ()
move.deformation.lattice.node ()

These commands allow the user to interactively choose a segment and deform it by moving an edge, face, or node of the existing deformation lattice around the segment. There is no way to programmatically deform a segment using JCL.

Example:
move.deformation.lattice.edge();

See Also:
create.segment.deformation.lattice

move.edge (object)
move.face (object)
move.node (object)
move.site (object)

object : An [edge|face|node|site] name string.

This command allows the user to interactively modify the psurf containing object.

Example:
moves.edge( "cube.cube.4" );

move.eyes (human, end.xform)

human : A human figure name string.
end.xform : A transform indicating the end position and orientation of the eye target of mass in global coordinates.

This command repositions human to put the eye target at end.xform.

Example:
**movef**

*figure*, *(endxform)*

*human*: A figure name string.

*endxform*: A transform indicating the end position and orientation of the root of figure in global coordinates.

This command repositions figure.

**Example:**

```plaintext
time = 1.39129
movef(figure, "cube", "xyz(144.67deg,-63.29deg,-64.03deg)
trans(34.49cm,96.38cm,-40.06cm));
```

**movef**

*figure*, *(window, endxform)*

*human*: A figure name string.

>window*: A window name string.

*endxform*: A transform indicating the end position and orientation of the root of figure in global coordinates.

This command repositions figure.

If *endxform* is omitted, the user has to reposition the view interactively. In this case, the viewpoint in *window* is temporarily moved to the root of *figure*, which is pretty fun.

**Example:**

```plaintext
movef(figure, "jack_window", "cube", "xyz(144.67deg,-63.29deg,142.67deg)
trans(-380.09cm,-363.70cm,-336.24cm);
```

**See Also:**

*movef*

**movefoot**

*human*, *(side, endxform)*

*human*: A human figure name string.

*side*: A string from "left" or "right", indicating which foot is to be moved.

*endxform*: A transform indicating the end position and orientation of the foot in global coordinates.

This command repositions human to put the foot at *endxform*.

**Example:**

```plaintext
time = 1.39129
movefoot("human5", "left", "xyz(144.67deg,-63.29deg,-64.03deg)
trans(34.49cm,96.38cm,-40.06cm));
```

**moveheel**

*human*, *(side, endxform)*

*human*: A human figure name string.

*side*: A string from "left" or "right", indicating which heel is to be moved.

*endxform*: A transform indicating the end position and orientation of the heel in global coordinates.

This command repositions human to put the heel at *endxform*.

**Example:**

```plaintext
time = 1.39129
moveheel("human5", "left", "xyz(144.67deg,-63.29deg,-64.03deg)
trans(34.49cm,96.38cm,-40.06cm));
```

**moveholdconstraint**

*constraint*

*constraint*: A constraint name string. The constraint must have a goal of type "hold".

This command allows the user to interactively move the point to which the end effector is constrained.

**Example:**

```plaintext
time = 1.39129
moveholdconstraint("human5", "left", "xyz(144.67deg,-63.29deg,-64.03deg)
trans(34.49cm,96.38cm,-40.06cm));
```

**moveholdconstraint**

*constraint*

*constraint*: A constraint name string. The constraint must have a goal of type "hold".

This command allows the user to interactively move the point to which the end effector is constrained.

**Example:**

```plaintext
time = 1.39129
moveholdconstraint("human5", "left", "xyz(144.67deg,-63.29deg,-64.03deg)
trans(34.49cm,96.38cm,-40.06cm));
```

**moveview**

*window*, *(xform, [depth])*

*window*: A window name string.

*xform*: A transform which places the camera site in global coordinates.

*depth*: A float which places the view reference point this far away from the camera site, down the negative z axis.

This command is identical to the command change-view, except that command moves the view instantly, while this command slowly pans to the new view.

This command will move the camera in *window* gradually to the specified position. The previous position is lost.

*depth* is a derived argument. It can be omitted unless there is a need to explicitly reposition the view reference point.

**Example:**

```plaintext
time = 1.39129
moveview("jack_window", xyz(-29deg,40deg,20deg)
trans(259cm,247cm,260cm));
```

**See Also:**

*push_view, pop_view, change_view*

**nodeinfo**

*node*

*node*: A node name string.

This command puts info on *node* to the Jack Message Window.

**Example:**

```plaintext
nodeinfo("cube.cube.2");
```

**pixelinfo**

*direction*

This command allows the user to interactively choose pixels in a Jack Window. Info on the chosen pixels is put to the Jack Message Window.

The info comprises location of the pixel in window coordinates, location in screen coordinates, and color in several formats.

**Example:**

```plaintext
pixelinfo();
```

**playframes**

*direction*

*direction*: A string from "forward", "backward", or "swing".

This command plays the computed frames that are currently in memory.

**Example:**

```plaintext
playframes("forward");
```
See Also:
  go, play.frames.background, stop.playback

pop.view (window)

  window : A window name string.

This command moves the view in window to the top view on the view stack.

Example:
pop.view("jack window");

See Also:
push.view, move.view

print.psrf (segment)

  segment : A segment name string.

This command sends info on segment to the printer.

Example:
print.psrf("cube.cube");

push.view (window, xform, depth)

  window : A window name string.
  xform : A transform which places the camera site in global coordinates.
  depth : A float which places the view reference point this far away from the camera site, down the negative z axis.

This command is identical to the command change.view, except that the previous view is pushed onto the view stack. depth is a derived argument. It can be omitted unless there is a need to explicitly reposition the view reference point.

Example:
push.view("jack.window", xyz(-29deg, 40deg, 20deg) trans(259cm, 247cm, 260cm));

See Also:
push.view, pop.view, change.view

reach_and_shape_hand (type, human, side, segment, location_type, site)

  type : A string from "disk", "power", "precision", "small disk", "tripod", indicating the type of grip to use.
  human : A human figure name string.
  side : A string from "left" or "right", indicating which hand is to grasp.
  segment : A segment name string.
  location_type : "calculated transform" or "predefined site".
  site : A site name string. This argument is omitted when location_type = "calculated transform".

This command moves the side hand of human such that it is grasping segment. The arguments location_type and site determine the position to which the hand is moved before the fingers begin to close around segment.

Example:
reach_and_shape_hand("power","human5","left", "cube.cube","calculated transform");

See Also:
grasp

reach.for.site (human, side, site, duration)

  human : A human figure name string.
  side : A string "left" or "right" indicating the hand to move.
  site : A site name string.
  duration : An integer indicating the duration in frames.

This command moves the side hand of human towards site while moving time forward duration frames. Note that this does not create a motion - the hand is moved when the command is executed, but the motion is not saved.

Example:
reach.for.site("human5","left","cube.cube.base",30);

See Also:
grasp, reach_and_shape_hand, move.arm, create_arm_motion, reach.for.site

read.environment.file (filename)

  filename : A file name string.

This command takes information from filename and applies it to the current environment.

Example:
read.environment.file("myobjects");

See Also:
read.file

read.figure.file (filename)

  filename : A file name string.

This command adds the figure from filename to the current environment.

Example:
read.figure.file("/usr/objects/human5.fig");

See Also:
read.file

read.file (filename)

  filename : A file name string.

This command takes the information from filename and applies it to the current environment.

Example:
read.file("myobjects.env");

read.peabody.string (string)

  string : A string of peabody code.

This command reads string and puts the results to the screen.

Example:
read.peabody.string("0.0 0.0 0.0 100.0 0.0 0.0 .0. 100.0 0.0 ; 1 2 3 [attribute 0];");
This command changes the root site of the figure containing site to site.

Example:
```
reroot.figure("cube .cube .newroot");
```

---

reset_clipping.planes (window);

window : A window name string.

This command resets the clipping planes in window to the default values.

Example:
```
reset_clipping.planes("jack window");
```

See Also:

adjust_clipping.planes

reset.figure (figure)

figure : A figure name string.

This command sets the displacement of all joints in figure to 0 for all degrees of freedom.

Example:
```
reset.figure("robot");
```

See Also:

adjust.figure, reset.joint

reset.joint (joint)

joint : A joint name string.

This command sets the displacement of joint to 0 for all degrees of freedom.

Example:
```
reset.joint("cube .propeller");
```

See Also:

go, start.time

reset.view.to.camera (window)

window : A window name string.

This command attaches the view in window to the camera in that window.

Example:
```
reset.view.to.camera("jack window");
```

See Also:

attach.view.to.site

reverse.face (face)

face : A face name string.

This command changes the direction of traversal of face so that its implied orientation will be reversed. A face will only be rendered properly on its "outside". The "outside" is determined by the direction of traversal of its constituent nodes, using the right-hand rule.

Example:
```
reverse.face("cube .cube .3");
```
See Also:
reverse segment faces

reverse segment faces (segment)
segment : A segment name string.

This command changes the direction of traversal of the faces of segment so that their implied orientation will be reversed. A face will only be rendered properly on its “outside”. The “outside” is determined by the direction of traversal of its constituent nodes, using the right-hand rule.

Example:
reverse segment faces("cube.cube");

See Also:
reverse face

rock frames (number)
number : An integer.

This command plays back and forth through the frames starting at number before the current frame and going up to number past the current frame. The frames must have already been generated using the command go or start time.

Example:
rock frames(5);

See Also:
play frames, go, start time

rotate pelvis (human, xform);
human : A human figure name string.
xform : A transform indicating the final position of the pelvis of human.

This command will adjust the goal of the constraint on the pelvis of human to xform.

Example:
rotate pelvis("human5",xyz(-87.42deg,14.57deg,168.48deg)*trans(-12.98cm,85.87cm,-11.25cm));

See Also:
create pelvis motion

save cad object (segment, filename)
segment : A segment name string.
filename : A file name string.

This command writes segment to a psurf file named filename. This command is different from write.psurf in that it will only operate on cad objects, and it adds some comments to the psurf file.

Example:
save cad object("sphere.sphere","sphere.pss");

See Also:
write.psurf

save figure dimensions (human, filename)
human : A human figure name string.

filename : A file name string.

This command saves the anthropometric information for human to a jcl file called filename. The information is saved as a series of jcl commands such as set head dimensions. The measurements can be re-applied to a human figure using the command read file.

Example:
save figure dimensions("human5", "/usr/measures.jcl");

See Also:
set-[foot/hand/lower arm/lower leg/neck/pelvis]/torso/upper arm/upper leg].dimensions, read file

save screen (filename)
filename : A file name string.

This command saves the screen image to an rgb file named filename. If filename does not have the .rgb suffix, it will be added.

Example:
save screen("screen.rgb");

see Also:
save window setup (filename)
filename : A file name string. This file name must have a .jcl suffix.

This command saves the current window setup to a jcl file named filename. The setup is saved in the form of jcl commands. The setup can be re-applied by reading the file with the command read file.

Example:
save window setup("windows.jcl");

See Also:
read file

scale environment (vector)
vector : A scaling vector aligned with the global coordinate system.

This command scales all figures in the current environment by the scale factors in vector.

Example:
scale environment((1,3.5,1));

See Also:
scale figure, scale environment globally

scale environment globally (vector)
vector : A scaling vector aligned with

This command scales all figures in the current environment by the scale factors in vector.

Example:
scale environment globally((1,3.5,1));

See Also:
scale figure, scale environment

scale figure (figure, vector)
figure : A figure name string.
vector : A scaling vector aligned with the coordinate system of figure.

This command scales figure by the scale factors in vector.

Example:
scalefigure("cube", (1, 2.5, 1));

See Also:
scale.environment, scale.figure.globally

scale.figure.globally (figure, vector)
figure : A figure name string.
vector : A scaling vector aligned with the global coordinate system.

This command scales figure by the scale factors in vector.

Example:
scalefigure.globally("cube", (1, 2.5, 1));

See Also:
scale.environment, scale.figure

segmentinfo (segment)
segment : A segment name string.

This command puts info on segment to the Jack Message Window.

Example:
segmentinfo("cube.cube");

select.currentfractal (figure)
figure : A fractal figure name string.

This command makes figure the current fractal. Subsequent fractal operations will act on this fractal.

Example:
select.currentfractal("myfractal");

See Also:
fractals

set.angle.precision (digits)
digits : An integer indicating the number of decimal places to use in displaying, storing and calculating angles.

This command sets the precision used in displaying, storing and calculating angles.

Example:
set.angle.precision(2);

set.angle.units (units)
units : A string "deg - degrees" or "rad - radians" indicating the default angle units.

This command sets the default angle units, i.e. the type of units assumed for angle arguments that are given without units. This also controls the units used when angle values are stored and displayed.

Example:

**set.angle.units("rad - radians");**

set.attribute.ambient (attribute, color)
set.attribute.color (attribute, color)
set.attribute.diffuse (attribute, color)
set.attribute.specular (attribute, color)

attribute : An attribute name string.
color : A color triple.

This command will change the [ambient|color|diffuse|specular] color of attribute to color.
Jack will query the user interactively for a new name for the attribute.

Example:
set.attribute.ambient("green", (.3, .5, 1));

See Also:
set.attribute.glossiness

set.attribute.glossiness (attribute, glossiness)

attribute : An attribute name string.
glossiness : A integer indicating glossiness.

This command will change the glossiness of attribute to glossiness.
Jack will query the user interactively for a new name for the attribute.

Example:
set.attribute.glossiness("green", 50);

See Also:
set.attribute.{ambient|color|diffuse|specular}

set.balance.control (human, balance)
create.timed.balance.control (human, start, end, velocity, balance)

human : A human figure name string.
start : A float indicating start time.
end : A float indicating end time.
velocity : A string from "accelerate", "constant", "decelerate", "ease in/ease out" indicating the change in velocity over the course of the motion.
balance : A string from "follow feet", "hold current position", "seated", "hold current elevation", "release elevation", "use upper body" indicating the type of balance control.

The command set.balance.control sets the type of balance control for human.
The command create.timed.balance.control each set a type of control for human during the indicated period. Before and after this period they have no effect. This commands will cause Jack to open an animation window if none is open.

Example:
set.balance.control("human5", "follow feet");

set.balance.control("human5", [balance.it, balance.eps, balance.point.eps, COM.eps])
balance.it : An integer indicating balance iterations.
balance.eps : A float indicating the balance epsilon.
balance_point.ep: A float indicating the balance point epsilon.
COM.ep: A float indicating the center of motion epsilon.

This command sets the balance parameters.

Example:
set.behavior.parameters(5,1.00,1.00,1.00);

set.collision.detect.type (method)

method: A string from "Gilber & Johnson", "Moore & Wilhelm", "The Unknown Method", "bounding box only", "bounding sphere only", indicating the algorithm to use for collision detection.

This command sets the algorithm to use for collision detection. See the Jack user's manual for descriptions of the algorithms.

Example:
set.collision.detect.type("The Unknown Method");

See Also:
set.collision.mode, set.collision.threshold, set.collision.display

set.collision.display (type)
type: A string from "off" or "highlight", indicating whether or not collisions should be highlighted.

This command sets whether or not collisions should be highlighted.

Example:
set.collision.display("off");

See Also:
set.collision.mode

Bugs:
Textured segments may disappear during highlighting.

set.collision.mode (mode)
mode: A string from "on demand", "continuous", or "prevention".

This command sets the collision mode, i.e. when Jack will check for collisions: "on demand": only updates CQ when a CQ.check.collision command is issued. "continuous": updates CQ with every interactive step. "prevention": updates CQ with every interactive step, plus backs up one frame when a collision occurs.

Example:
set.collision.mode("on demand");

See Also:
set.detection.type

Bugs:
"prevention" mode is not yet implemented.

set.collision.threshold (number)
number: A float indicating the maximum distance at which a collision is reported.

This command sets the collision threshold for detection. This is only used with the Gilbert & Johnson detection method.

Example:
set.collision.threshold(0cm);

See Also:
set.collision.detect.type

set.constraint.step.factor (distance)
distance: A float indicating step distance.

This command controls the distance between subgoals that Jack pursues in evaluating constraints. When this number is set low, constraints will be solved more smoothly but more slowly. The default value is 10cm.

Example:
set.constraint.step.factor(10cm);

See Also:
set.constraint.time.limit

set.constraint.time.limit (time)
time: A float indicating time limit.

This command sets the constraint time limit. This is the number of seconds Jack will spend per iteration in trying to evaluate constraints. When this number is high, the end effector will track the goal more closely, but interactive actions will be more sluggish. The default value is .10sec.

Example:
set.constraint.time.limit(.10sec);

See Also:
set.constraint.step.factor

set.constraint.weight (constraint, weight)
constraint: A constraint name string.
weight: A float indicating the constraint weight.

This command sets the weight of constraint to weight.

Example:
set.constraint.weight("CO",3.00);

See Also:
change.constraint

set.deformation.mode (segment, mode)
segment: A segment name string.
mode: A mode type string. This must be: "interactive", "off", "on demand", or "semi-interactive".

This command sets the frequency for re-drawing segment when its deformation lattice is modified.

Example:
set.deformation.mode("cube.cube","interactive");

set.distance.precision (digits)
digits: An integer indicating the number of decimal places to use in displaying, storing and calculating distances.
This command sets the precision used in displaying, storing and calculating distances.

Example:
```c
set.distance.precision(2);
set.distance.units (units)
units : A string from "cm - centimeters", "in - inches", "mi - miles", "yd - yards", "ft - feet", "m - meters", "mm - millimeters" indicating the default distance units.
```

This command sets the default distance units, i.e. the type of units assumed for distance arguments that are given without units. This also controls the units used when distance values are stored and displayed.

Example:
```c
set.distance.units("cm - centimeters");
```

This command sets the eye coordinates for human.

Example:
```c
set.eye.coordinates (human, values)
human : A human figure string.
values : A vector of floats indicating body center, interpupillary and chin distances.
```

Example:
```c
set.eye.coordinates(9.20,8.12,12.88);
```

This command sets the field of view in window.

Example:
```c
set.field.of.view (window, FOV)
window : A window name string.
FOV : A float indicating field of view.
```

Example:
```c
set.field.of.view("jack window",40.00);
```

See Also:
```c
change.view
```

See Also:
```c
set.foot.control (human, side, foot)
```

```
create.timed.foot.control(human, start, end, side, foot)
human : A human figure name string.
start : A float indicating start time.
end : A float indicating end time.
side : A string from "left", "right", "both", indicating which side of the body is affected.
foot : A string from "follow balance line", "hold global location", "hold local location", "pivot", "release", indicating the type of foot control.
```

The command `set.foot.control` sets the foot control for human. The command `create.timed.foot.control` sets a type of control for humans during the indicated period. Before and after this period it has no effect. This command will cause Jack to open an animation window if none is open.

Example:
```c
set.foot.control("human", "left", "hold global position");
```

Bugs:
```c
set.foot.control to "follow balance line" has not been implemented yet.
```

```
set.foot.dimensions (human, values)
set.head.dimensions (human, values)
set.leg.dimensions (human, values)
```

These commands set the dimensions for the [foot|hand] head[lower.arm|lower.leg][neck|pelvis|torso][upper.arm|upper.leg] of human.

Example:
```c
set.foot.dimensions(20.0,32.5,13.05);
```

```
set.frame (frame)
frame : An integer indicating the frame to set.
```

This command sets frame number frame to the image in the current window.

Example:
```c
set.frame(0);
```

```
set.grid.blocks (window, major, minor)
window : A window name string.
major : An integer indicating the number of major blocks.
minor : An integer indicating the number of minor blocks per major block.
```

This command sets the floor grid in window.

Example:
```c
set.grid.blocks("jack window",2,6);
```

```
set.hand.control (human, side, hand, [handgoal], startjoint)
create.timed.hand.control (human, start, end, side, hand, [handgoal], startjoint)
human : A human figure name string.
start : A float indicating start time.
end : A float indicating end time.
side : A string from "left", "right", "both", indicating which side of the body is affected.
hand : A string from "hips", "hold global location", "hold local location", "knees", "release", "site", indicating the type of hand control.
handgoal : This is a conditional variable. When hand = "hold global location" or "hold local location", handgoal must be set to a transform. When hand = "site", handgoal must be set to a site name string. In other cases, handgoal is omitted.
```
**startjoint** : A string from "shoulder" or "waist" indicating the beginning joint that should be moved to achieve the hand behavior.

The command **set_hand.control** sets the hand control for human.

The command **create_timed_hand.control** sets the hand control for human during the indicated period. Before and after this period it has no effect. This command will cause Jack to open an animation window if none is open.

**Example:**

```
set_hand.control("human5","left","hips","waist");
```

---

**set_head.control** (human, head, [headgoaltypetype], [headgoal])

**create_timed_head.control** (human, start, end, head, [headgoaltypetype], [headgoal])

**human** : A human figure name string.

**start** : A float indicating start time.

**end** : A float indicating end time.

**head** : A string from "fixate" or "release" indicating the desired type of head control.

**headgoaltypetype** : A string from "closest node", "face", "hold", "node", "relative transform", "site", "transform" indicating the type of goal that the head is fixated on. This is a conditional variable. It is omitted when head = "release".

**headgoal** : The type of this variable is determined by the value of headgoaltypetype. This is a conditional variable. It is omitted when head = "release".

The command **set_head.control** sets the head control for human.

The command **create_timed_head.control** sets the head control for human during the indicated period. Before and after this period it has no effect. This command will cause Jack to open an animation window if none is open.

**Example:**

```
set_head.control("human5","fixate","closest node", "cube", "cube");
```

---

**set_horizontal.view.gain** (gain)

**set_vertical.view.gain** (gain)

**set_zoom.view.gain** (gain)

**gain** : An integer indicating the view gain.

This command sets the [horizontal-vertical-zoom] view gain to gain. This controls Jack's sensitivity to mouse movements when the view is being moved.

**Example:**

```
set_horizontal.view.gain(1.5);
```

---

**set.*OBJECT*.color** (color)

**color** : A color triple.

These commands set the colors of various incidental objects in the jack environment. "OBJECT" can be replaced by:

background

ceiling_node

floor_node

---

**set_inside.wheel**

**major_grid**

**major_highlight**

**minor_grid**

**minor_highlight**

**node**

**outside.wheel**

**site**

**star**

**status_background.color**

**status_highlight.color**

**status_text_color**

**trace_color**

**Example:**

```
set_inside.wheel.color((.5,.5,.3));
```

---

**set_joint.limits** (joint, lower, upper)

**joint** : A joint name string.

**lower** : A joint position vector indicating the lower limits of the joint in the various degrees of freedom.

**upper** : A joint position vector indicating the upper limits of the joint in the various degrees of freedom.

This command sets the joint limits of joint to upper and lower.

**Example:**

```
set_joint.limits("fig1.fig2",(10.5cm,0.0deg), (100cm,90.5deg));
```

---

**set_joint.type** (joint, type)

**joint** : A joint name string.

**type** : A degrees of freedom expression.

This command sets the type of joint to type. This sets the degrees of freedom in which the joint can move.

**Example:**

```
set_joint.type("fig1.fig2","T(z)*R(x)");
```

---

**set_knee.control** (human, side, knee)

**human** : A human figure name string.

**side** : A string from "left", "right", "both", indicating which side of the body is affected.

**knee** : A string from "hold current orientation", "keep vertical", "release", indicating the type of knee control.

The command **set_knee.control** sets the knee control for human.

**Example:**

```
set_knee.control("human5","left","keep vertical");
```

---

**set_light.ambient** (light, color)

---

**Bugs:**

There is no command **set_timed_knee.control**.
light: A light source segment name string.
color: A color triple.

This command sets the ambient color of light to color.

Example:
```
set.light.ambient("lightsources.lightsource", (.56, .33, 0.00));
```

See Also:
```
set.light.ambient
```

**set.light.color**

light: A light source segment name string.
color: A color triple.

This command sets the color of light to color.

Example:
```
set.light.color("lightsources.lightsource", (.56, .33, 0.00));
```

See Also:
```
set.light.ambient
```

**set.light.concentration**

light: A light source segment name string.
concentration: An integer indicating the concentration of light.
radius: A float indicating the radius of light.

This command sets the concentration of the light source light.

Example:
```
set.light.concentration("lightsources.lightsource", 1, 30.00);
```

**set.linspace**

width: An integer indicating linewidth in pixels.

This command sets the line width of incidental lines in the environment to width.

Example:
```
set.linspace(1);
```

**set.motion.group**

name: A motion group name string.

This command makes the current motion part of the motion group name. The current motion must be selected interactively by the user before this command is executed.

Example:
```
set.motion.group("group1");
```

**set.move.glide**

value: A float indicating the amount of glide in moving objects.

This command sets the amount of glide there will be when the user moves objects. A setting of 1 means no glide, i.e., when the user stops moving the mouse, the object will stop moving. If the glide is set higher than one, the object will keep moving after the mouse has stopped, and its speed will depend on the speed with which the mouse was moved.

Example:
```
set.move.glide(2);
```

**set.movie.directory**

directory: A directory name string.

This command sets the default directory in which movie files are saved.

Example:
```
set.movie.directory("/user/movies");
```

**set.path.parameters**

path: A path segment name string.
samples: An integer indicating the number of points to calculate to interpolate path.

This command sets the smoothness of the path calculated for path. The more points that are used, the smoother the path will be. In future versions, this command will be used to set other path parameters if necessary.

Example:
```
set.path.parameters("mycurves.paths", 30);
```

See Also:
```
create.path
```

**set.pelvis.control**

human: A human figure name string.
start: A float indicating start time.
end: A float indicating end time.
pelvis: A string "follow feet" or "hold orientation" indicating the type of pelvis control.

The command set.pelvis.control sets the pelvis control for human. The command create.timed.pelvis.control sets the pelvis control for human during the indicated period. Before and after this period it has no effect. This command will cause Jack to open an animation window if none is open.

Example:
```
set.pelvis.control("human5", "follow feet");
```

**set.preferred.angle**

joint: A joint name string.
weight: A float indicating the relative importance of this constraint when it conflicts with other constraints.

This command sets a preferred angle constraint on joint. The joint will seek a goal of zero displacement.

Example:
See Also:
create.constraint

set.record.mode (mode)

mode : A string from "field" or "frame".
This command sets the mode for recording motions to video disk.
In frame mode, an entire frame is calculated and written to disk at one time. In field mode, the animation is advanced one half frame between writing the odd lines and the even lines to disk. Frame mode produces cleaner still images. Field mode writes more slowly, but provides smoother animation.

Example:
set.record.mode("field");

gset.render.parameters (samples)
samples : A string of the form "nn samples", where nn is an integer indicating the number of sample per point to use for anti-aliasing.
This command sets the anti-aliasing resolution. In the future, this command will be used to set other rendering parameters, if necessary.

Example:
set.render.parameters("4 samples");

set.rotation.type (type)
type : A string from "xyz - angles" or "quat - quaternion", indicating the default format for rotational transforms.
This command sets the default format for rotational transforms. This controls the format that is used for display and storage of transforms.

Example:
set.rotation.type("quat - quaternion");

set.ruler.parameters (resolution)
resolution : An integer indicating the number of pixels per tick to be used in displaying rulers.
This command sets the ruler tick resolution to resolution. In the future, this command will be used to set other ruler parameters, if necessary.

Example:
set.ruler.parameters(10);

set.scene.aux.scale (factor)
factor : A float indicating aux scale factor.
This command sets the scaling factor for sizing auxiliary objects in the Jack environment, including sites, nodes, and the rotation wheel. The scaling factor is the ratio of the height of a site frame to the height of a "standard" object, as defined by the scene scale.

Example:
set.scene.aux.scale(.25);

See Also:
set.scene.scale

set.scene.scale (scale)
scale : A float indicating the height of a "standard" object. This command sets the scaling factor used in rendering Jack windows. The windows are scaled such that a "standard" object will fit comfortably on the screen.

Example:
set.scene.scale(200.00cm);

set.segment.density (segment, density)
ssegment : A segment name string.
density : A float indicating density in grams per cubic centimeter.
This command sets the density of segment to density.

Example:
set.segment.density("cube.cube",2.00);

See Also:
set.segment.mass

set.segment.mass (segment, mass)
ssegment : A segment name string.
mass : A float indicating mass.
This command sets the mass of segment to mass.

Example:
set.segment.mass("cube.cube",20.00);

See Also:
set.segment.density

set_stepping_behavior (human, behavior)
human : A human figure name string.
behavior : A string from "off", "take step when pelvis twisted", or "take step before losing balance", indicating the stepping behavior for human.
This command sets the stepping behavior for human. This controls when the figure will automatically take a step to adjust its position.

Example:
set_stepping_behavior("human5","off");

set.torso.control (human, torso, [torso.top], [torso.bottom], [torso.initiator], [torso.resistor], [torso.resistance], [torso.interpolate])
create.timed.torso.control (human, start, end, torso, [torso.top], [torso.bottom], [torso.initiator], [torso.resistor], [torso.resistance], [torso.interpolate])
human : A human figure name string.
start : A float indicating start time.
end : A float indicating end time.
torso : A string from "keep vertical", "hold orientation", "none", "set parameters" indicating the type of torso control.
torso.top : The upper limit of joints in the spine that are free to rotate in maintaining the torso position. This must be an integer from 1 to 17. This is a conditional variable. It is omitted when torso = "none".
torso.bottom : The lower limit of joints in the spine that are free to rotate in maintaining the torso position. This must be an integer from 1 to 17. This is a conditional variable. It is omitted when torso = "none".
torso.initiator : The joint that will move rotate most in maintaining the torso position. This must be an integer from 1 to 17. This is a conditional variable. It is omitted when torso = "none".
torso.resistor : The joint that will rotate least in maintaining the torso position. This must be an integer from 1 to 17. This is a conditional variable. It is omitted when torso = "none".
torso.resistance : The resistance to rotation in maintaining torso position. This must be a value 0.0torso = "none".
torso.interpolate : A string "yes" or "no" which controls whether Jack will interpolate through the resting position in maintaining torso position. This is a conditional variable. It is omitted when torso = "none".

The command set.torso.control sets the torso control for human. The command create.timed.torso.control sets the torso control for human during the indicated period. Before and after this period it has no effect. This command will cause Jack to open an animation window if none is open.

Example:
set.torso.control("human5","keep vertical",1,17,17,50,"yes");

set.view.glide (value)
value : A float indicating the amount of glide in moving the view.

This command sets the amount of glide there will be when the user moves the view.
A setting of 1 means no glide, i.e. when the user stops moving the mouse, the view will stop moving.
If the glide is set higher than one, the view will keep moving after the mouse has stopped, and its speed will depend on the speed with which the mouse was moved.

Example:
set.view.glide(2);

set.window.location (window, location)
window : A window name string.
location : A screen rectangle.
This command moves window to location.

Example:
set.window.location("jack window",(8,778,277,992));

shade.windows ()
This command temporarily makes all windows shaded. Jack will wait for a keystroke before returning windows to normal and continuing.

Example:
site_info (site)
site : A site name string.
This command puts info on site into the Jack Message window.
Example:
site_info("world.base");

snap_view (window, site)
window : A window name string.
site : A site name string.
This command moves the view in window such that site is
centered in the window.
Example:
snap_view("jack window", "world.base");

split_edge (edge)
edge : An edge name string.
This command adds a node to the segment containing edge.
The user must place the node interactively, then two edges are
created connecting the endpoints of the original edge to this
node.
Example:
split_edge("cube.cube.6");

split_face (face, node1, node2)
face : A face name string.
node1 : A node name string.
node2 : A node name string.
This command splits face into two faces along the line between
node1 and node2. The nodes must belong to face.
Example:
split_face("cube.cube.3","cube.cube.6","cube.cube.4");

start_time()
This command calculates and displays the current motions,
starting at the current time. It works like the command go,
except that go always starts at time = 0.
Example:
start_time();
See Also:
go, goto_time, stop_time

step_through_frames()
This command allows the user to step through the current
frames interactively. Note that the frames must have already
been generated using the command go or start_time.
Example:
step_through_frames();

See Also:
play_frames, go, start_time

step_time()
This command allows the user to advance time interactively,
starting at time 0. The frames of the animation are generated
and displayed one by one.
Example:
step_time();

stop_playback()
When Jack is playing frames, this command will stop it.
Example:
stop_playback();
See Also:
play_frames, stop_time

stop_time()
When Jack is moving time, e.g. as a result of the go command,
this command will stop it.
Example:
stop_time();

switch_figure (human)
human : A human contour or stick figure name.
This command replaces a contour figure, human, with a stick
figure and vice versa.
Example:
switch_figure("contour");

toggle_psurf.bounding_boxes()
This command toggles the setting for show/hide psurf bounding
boxes. When bounding boxes are off, the psurfs do not
show up at all.
Example:
toggle_psurf.bounding_boxes();
See Also:
toggle_psurf.nodes_only

toggle_psurf.nodes_only()
This command toggles the setting for show only psurf nodes.
Example:
toggle_psurf.bounding_boxes();
See Also:
toggle_psurf.bounding_boxes

trace_segment (segment) untrace_segment (segment)
segment : A segment name string.
These commands turn on and off a trace on segment. When
a trace is on a segment, a trail is left behind whenever the
segment is moved.
Example:
tracesegment("cube .cube1");

See Also:
clear.trace , tracesite , delete.trace

trace.site(site) untrace.site(site)
site : A site name string.
These commands turn on and off a trace on site. When a trace is on a site, a trail is left behind whenever the site is moved.

Example:
trace.site("cube .cube .base");

See Also:
clear.trace , tracesite , delete.trace

turn.background.on (window)
turn.background.off (window)
window : A window name string.
These commands turn the background (i.e. grid, stars, background color) on and off in window.

Example:
turn.background.off("jack window");

turn.behaviors.on ()
turn.behaviors.off ()
These commands enable and disable all jack behaviors in the current environment.

Example:
turn.behaviors.off();

turn.camera.on (window)
turn.camera.off (window)
window : A window name string.
These commands control whether the camera in window is visible as an object.

Example:
turn.camera.off("jack window");

turn.constraint.on (constraint)
turn.constraint.off (constraint)
turn.constraints.on ()
turn.constraints.off ()
constraint : A constraint name string.
These commands enable and disable constraints. Note that they do not delete the constraints permanently. This can be done with the commands delete.constraint[all.constraints]. The "constraint" commands act only on constraint. The "constraints" commands act on all constraints in the environment.

Example:
turn.constraint.off("Cl");

See Also:
delete.constraint , delete.all.constraints

turn.edges.off (segment)
turn.edges.on (segment)

segment : A segment name string.
These commands show and hide the edges of segment.

Example:
turn.edges.off("cube .cube");

turn.face.enumeration.on (segment)
turn.face.enumeration.off (segment)
segment : A segment name string.
These commands show and hide face numbers on segment.

Example:
turn.face.enumeration.off("cube .cube");

turn.face.normals.on (segment)
turn.face.normals.off (segment)
segment : A segment name string.
These commands show and hide normal vector arrows on the faces of segment.

Example:
turn.face.normals.off("cube .cube");

turn.figure.on (figure)
turn.figure.off (figure)
figure : A figure name string.
These commands show and hide figure.

Example:
turn.figure.off("cube");

turn.figure.projections.on (figure)
turn.figure.projections.off (figure)
figure : A figure name string.
These commands show and hide the projections of figure.

Example:
turn.figure.projections.off("cube");

turn.figure.sites.on (figure)
turn.figure.sites.off (figure)
figure : A figure name string.
These commands show and hide the sites on figure.

Example:
turn.figure.sites.off("cube");

turn.framerate.on ()
turn.framerate.off ()
These commands show and hide the framerate display in the corner of jack windows.

Example:
turn.framerate.off();

turn.global.axis.on ()
turn.global.axis.off ()
These commands show and hide the global coordinate frame.

Example:
These commands show and hide the nodes of segment.

Example:

```plaintext
turn.nodes.off("cube.cube");
```

These commands show and hide projections in window.

Example:

```plaintext
turn.projections.off("jack window");
```

These commands enable and disable messages in jack windows.

Example:

```plaintext
turn.screen.messages.off();
```

These commands show and hide projections of segment.

Example:

```plaintext
turn.segment.projections.off("cube.cube");
```

These commands show and hide the sites on segment.

Example:

```plaintext
turn.segment.sites.off("cube.cube");
```

These commands turn show and hide node numbers on segment.

Example:

```plaintext
turn.node.enumeration.off("cube.cube");
```

These commands turn show and hide grid (aka "floor");

Example:

```plaintext
turn.grid.on();
```

These commands enable and disable the current motion group. The user must interactively select the current motion group before this command is issued.

Example:

```plaintext
turn.motion.group.on();
```

These commands turn off or on the motor on a specified joint or all motors in the current environment.

Example:

```plaintext
turn.all.motors.on();
```

These commands temporarily turn on and off the current motion. When a motion is off, it is ignored by the animation system. Motions can be deleted permanently with the command delete.motion.

Example:

```plaintext
turn.motion.off();
```

These commands show and hide site labels throughout the environment.

Example:

```plaintext
turn.sitelabels.off();
```

These commands turn on and off the grid.

Example:

```plaintext
turn.grid.on();
```

These commands show and hide the grid (aka "floor");

Example:

```plaintext
turn.grid.off();
```

These commands turn off or on the motor on a specified joint.

Example:

```plaintext
turn.motor.on("joint");
```

These commands turn on or off the motor on a specified joint.

Example:

```plaintext
turn.motor.off("joint");
```

These commands show and hide the motor on a specified joint.

Example:

```plaintext
turn.all.motors.off();
```

These commands turn off or on all motors in the current environment.

Example:

```plaintext
turn.all.motors.on();
```

These commands enable and disable messages in jack windows.

Example:

```plaintext
turn.screen.messages.on();
```

These commands turn on and off the current motor group.

Example:

```plaintext
turn.motion-group.off();
```

These commands temporarily turn on and off the current motion.

Example:

```plaintext
turn.motion.off();
```

These commands show and hide all motors in the current environment.

Example:

```plaintext
turn.all.motors.on();
```

These commands turn on or off all motors.

Example:

```plaintext
delete.motor, delete.all.motors
```

These commands show and hide site labels throughout the environment.

Example:

```plaintext
turn.sitelabels.on();
```

These commands turn on and off the site.

Example:

```plaintext
turn.site.on("site");
```

These commands turn on and off the site.

Example:

```plaintext
turn.site.off("site");
```

These commands turn on and off the joint.

Example:

```plaintext
turn.motor.on("joint");
```

These commands enable and disable the current joint.

Example:

```plaintext
turn.motor.off("joint");
```

These commands turn off or on the joint.

Example:

```plaintext
turn.all.motors.off();
```

These commands turn show and hide node numbers on segment.

Example:

```plaintext
turn.node.enumeration.off("cube.cube");
```

These commands turn off or on the joint.

Example:

```plaintext
turn.motor.on("joint");
```

These commands turn off or on the joint.

Example:

```plaintext
turn.motor.off("joint");
```

These commands turn off or on the joint.

Example:

```plaintext
turn.all.motors.off();
```

These commands turn on or off all motors.

Example:

```plaintext
delete.motor, delete.all.motors
```

These commands turn on and off the joint.

Example:

```plaintext
turn.motor.on("joint");
```

These commands enable and disable the current joint.

Example:

```plaintext
turn.motor.off("joint");
```

These commands turn off or on the joint.

Example:

```plaintext
turn.all.motors.off();
```

These commands turn on or off all motors.

Example:

```plaintext
delete.motor, delete.all.motors
```

These commands show and hide the motor.

Example:

```plaintext
turn.motor.on("joint");
```

These commands enable and disable the current joint.

Example:

```plaintext
turn.motor.off("joint");
```

These commands turn on or off the joint.

Example:

```plaintext
turn.all.motors.on();
```

These commands turn on and off all motors.

Example:

```plaintext
delete.motor, delete.all.motors
```

These commands show and hide the motor.

Example:

```plaintext
turn.motor.on("joint");
```

These commands enable and disable the current joint.

Example:

```plaintext
turn.motor.off("joint");
```

These commands turn on or off the joint.

Example:

```plaintext
turn.all.motors.on();
```

These commands turn on and off all motors.

Example:

```plaintext
delete.motor, delete.all.motors
```

These commands show and hide the motor.

Example:

```plaintext
turn.motor.on("joint");
```

These commands enable and disable the current joint.

Example:

```plaintext
turn.motor.off("joint");
```
APPENDIX G. JCL REFERENCE MANUAL

This command creates a segment that is the union between segment1 and segment2. The new segment is called “union.union”. If there is already a figure “union”, the name will be made unique by adding an integer to the end.

Example:
union.segments("cube1.cube","cube2.cube");

See Also:
intersect.segments, difference.segments

writeJCLlog (filename)

filename : A filename string. The suffix must be “.jcl”.

This command writes all jcl commands used in the current session to filename.

Example:
writeJCLlog("actions.jcl");

See Also:
write.environment

write.complete.environment (filename)

filename : A filename string. The suffix must be “.env”.

This command saves the current environment, including jack state variables, to filename.

Example:
write.complete.environment("world.env");

See Also:
write.environmet

write.environment (filename)

filename : A file name string. The suffix must be “.env”.

This command writes the current environment, but not jack state variables, to filename. The user will be queried interactively as to whether or not to save the camera position.

Example:
write.environment("world.env");

write.environment.to.psurf (filename)

filename : A file name string. The suffix must be “.psf”.

This command writes all figures in the environment to a psurf file. The figures are combined into one psurf, so when the file is read, they become a single segment.

Example:
write.environment.to.psurf("blob.psf");

write.figure.definition (figure, filename, attributes)

figure : A figure name string.
filename : A file name string. The suffix must be “.fig”.
attributes : A string “yes” or “no”, indicating whether attributes should be included in the file.

This command writes a description of figure to a figure file named filename.

Example:
write.figure.definition("robot", "/usr/robot.fig", "yes");

---

turn_stars_off (window)
turn_stars_on (window)

window : A window name string.

These commands show and hide the background stars in window.

Example:

turn_stars_off("jack window");

turn_support_polygon_off (human)
turn_support_polygon_on (human)

human : A human figure name string.

These commands show and hide the support polygon of human.

Example:

turn_support_polygon_off("human5");

turn_x_projection_off (window)
turn_y_projection_off (window)
turn_z_projection_off (window)

window : A window name string.

These commands turn on and off projections along the [x|y|z] axes in window.

Example:

turn_x_projection_off("jack window");

two_sided_lighting_off ()
two_sided_lighting_on ()

These commands turn the two sided lighting model on and off.

Example:

two_sided_lighting_off();

unattach.figure (figure)

figure : A figure name string.

This command unattaches figure.

Example:

unattach.figure("cube");

See Also:
attach.figure

uninstall_file (filename)

filename : A file name string.

This command removes filename from the jack installation file.

Example:

uninstall_file("/user/model");

union.segments (segment1, segment2)

segment1 : A segment name string.
segment2 : A segment name string.
write.figure.position (figure, filename)
figure : A figure name string.
filename : A file name string. The suffix must be ".env".
This command writes the position of figure to an environment file named filename.

Example:
write.figure.position("robot","/usr/robot.env");

write.figure.psrefs (figure)
figure : A figure name string.
This command writes the segments of figure to separate psurf files. Jack will query the user interactively for each of the filenames.

Example:
write.figure.psrefs("robot");

write.fractal.as.psurf (fractal)
fractal : A fractal segment name string.
This command writes the fractal segment to a psurf file. Jack will query the user interactively for the filename.

Example:
write.fractal.as.psurf("fractal.fractal");

write.frames (filename)
filename : A file name string.
This command will write the current frames to filename.

Example:
write.frames("/usr/animation");

write.frames.to.env.files (base, startindex, all)
base : A base file name string.
startindex : An integer.
all : A string "yes" or "no", indicating whether to write the entire environment, i.e. attribute and figure descriptions and file header comments. If all = "no", only figure locations will be written.
This command will write the current frames to disk as a series of env files. Note that you must generate the frames using the command go or start.time before writing the frames to disk.
The frames will be written one per file. The files will be named baseNNN.env, where NNN is an integer index beginning at startindex.

Example:
write.frames.to.env.files("/usr/movie",0,"no");

See Also:
go, start.time, record.frames.to.vdisk
write.motions (filename1, filename2)
filename1 : A file name string. The suffix must be ".env".
filename2 : A file name string. The suffix must be ".env".
This command writes the current environment to filename1 and writes the motions in the current environment to filename2.

Example:
write.motions("/usr/wld.env","/usr/wld.motions.env");

write.positions (filename)
filename : A file name string. The suffix must be ".env".
This command writes a list of figure and joint positions to a peabody file named filename.
If this file is read back, the objects in the environment will return to those positions. Note that reading this file will not recreate the objects if they are no longer present.

Example:
write.positions("positions.env");

See Also:
write.environment

write.ps[.globally] (segment, filename)
segment : A segment name string.
filename : A file name string.
These command writes the psurf of segment to filename. The command write.psurf will write the node positions relative to the root site, so that when the psurf is reloaded it will be positioned at the global origin.
To preserve the global position of the segment, use the command write.psurf.globally, which writes the node positions relative to the global origin.

Example:
write.psurf("cube.cube","/usr/cube.ps")

write.to.vdisk (area, startframe)
area : A string from "window" or "whole screen".
startframe : An integer indicating at which frame on the video disk to begin recording.
This command records the current frames to video disk. Note that before the frames can be recorded, they must be generated using the command go or start.time.

Example:
write.to.vdisk("window",234);

See Also:
go, start.time, write.frames.to.vdisk

write.window.image (window, filename)
window : A window name string.
filename : A file name string. The prefix must be ".rle".
This command writes an image of window to filename.

Example:
write.window.image("jack window","picture.rle");

See Also:
Appendix H

Utility Programs

The following is a list of programs which accompany Jack to perform various functions. These are all provided with Jack but they are separate programs, not callable within Jack.

H.1 Psurf Programs

H.1.1 bps — Binary Psurf Generator

The program bps generates a binary psurf file. A binary psurf is a pre-process form of the psurf which Jack can read much more efficiently. You should run bps on any psurfs which you use frequently. Binary psurf files are rather large, though, so you need to be careful not to generate them indiscriminately.

bps takes a single argument, giving the name of the psurf file, ending in .pss. It generates a binary psurf file with the same base name but with the suffix .bps.

% bps thing.pss

If bps prints messages saying that something is wrong with faces, do not worry to much. This means that the topology of the object is not what it should be, but this seldom makes a big difference. In any case, the appearance of an object that comes from a bps file which generated such messages will be the same as if it had been read from a .pss file.

H.1.2 pnr — Psurf Node Reducer

pnr eliminates redundant nodes, or merges close nodes.

% pnr [-f file] [-r range]

pnr reads from standard input and writes to standard output. The "-f" option reads from "file" instead of standard input.

Nodes within "range" are merged into one node and the faces are updated accordingly.

"range" is 0 by default. Other values effectively produce a simple psurf face decimator.

Elimination is done on a first-seen, first-kept basis. Comments are not permitted in input file.

H.1.3 pfr — Psurf Face Reducer

pfr eliminates or corrects bad faces in a psurf file.

% pfr [-f file] [-1]

pfr reads from standard input and writes to standard output. The "-f" option reads from "file" instead of standard input.

Faces with bad edges are reduced. Zero length edges as well as 0 area cycles are eliminated. Single-edge faces are eliminated by default.

The "-1" option prevents elimination of single-edge faces i.e. 2 vertex faces.
Elimination of cycles is done on only 3 vertex cycles. Longer cycles may not be correctly eliminated. Comments are not permitted in input file.

**H.1.4 psp — Psurf Splitter**

psp splits large psurfs into csurf or figures.

```
% psp [-n numnodes] [-f infile] [-o outbase] [-b]
```

psp reads from standard input and writes to standard output. The "-f" option reads from "infile" instead of standard input.

The input psurf is split into separate csurfs and written to standard output as one psurf.

The psurf is split at approximately every 2000 nodes. The "-n" option allows changing this default.

If the "-o" option is specified, separate pspurf files are generated for each csurf created. A figure file containing all of the created pspurf segments is also created with the name "outbase.fig". The pspurf files are named "outbase"*x*.fig, where "x" is the number of the pspurf created starting at 0. No output to standard output is produced.

The "-b" option writes the [big] option at the top of each created pspurf file. Normally [big] is only written if present in the original file.

Splitting is done semi-randomly. Each split takes place after faces containing "numnodes" distinct nodes have been read in. Vertices are renumbered so that vertex numbers increase sequentially as new vertices are encountered.

Because splitting is done on a purely sequential basis, splits may not occur on geometric discontinuities. An enhancement would be to divide across disjoint geometry.

Also, greater than "numnodes" may be present in any particular csurf. This is due to the last face in the csurf having more distinct nodes than necessary to reach the desired total. This should be taken into account in pspurf containing faces with a large number of vertices. Comments are not permitted in input file.

**H.2 Image Programs**

**H.2.1 display**

The program display displays an .rle image on the IRIS screen. To run it, type "display" and the name of the image file.

```
% display image.rle
```

display can automatically read compressed files. To read a compressed file, you can type its name with or without the .Z suffix.

display can also display multiple images by just entering more than one name on the command line. It will display the first one and then allow you to choose from the others with a pop-up menu.

display can draw the alpha channel of an image by using the -a option. This will display the alpha channel as a grayscale image.

```
% display -a image.rle
```

**H.2.2 rletovdisk**

rletovdisk sends an Utah Raster Toolkit rle image to the Abekas video disk. It takes two arguments: the first argument is the name of an rle file; the second is the frame number of the video disk.

```
% rletovdisk image.rle 97
```

To use this program at places other than the Graphics Lab at Penn, you must obviously have an Abekas video disk. If you do, you must set the Unix shell variable VDISK to be the host name for the video disk.

```
% setenv VDISK vdisk.cis.upenn.edu
```

Please set it to your own machine, though, so you don’t write to our video disk, at least not without asking.
H.2.3 scrtovdisk

The program scrtovdisk sends an image of the entire IRIS screen to the video disk. It reads directly from the framebuffer and automatically shrinks the image to the proper size. It actually omits the bottom few lines of the screen, so it conveniently misses the Jack status window. The program takes a single numerical argument giving the frame number on the video disk.

```
% scrtovdisk 97
```

Just like printrle, to use this program at places other than the Graphics Lab at Penn, you must obviously have an Abekas video disk. If you do, you must set the Unix shell variable VDISK to be the host name for the video disk.

```
% setenv VDISK vdisk.cis.upenn.edu
```

H.2.4 printrle

The program printrle prints an image on the Textronix Color printer in the Graphics Lab at Penn. It takes a single argument giving the name of the the rle file.

```
% printrle image.rle
```

This command is not available at places other than Penn, unless you have a Tektronix printer.

H.3 Utah Raster Toolkit (RLE) Programs

The programs listed here are part of the Utah Raster Toolkit. Jacks the toolkit for creating image files (file.rle files). This is a subset of the utilities provided by the toolkit. These are the man pages provided by the University of Utah. The URT is available (source code and this documentation) free of charge via anonymous ftp to cs.utah.edu.
**APPLYMAP**

**NAME**

`applymap` - Apply the color map in an RLE file to the pixel data

**SYNOPSIS**

`applymap [-1] [-o outfile] [infile]`

**DESCRIPTION**

This program takes the color map in an RLE(5) file and modifies the pixel values by applying the color map to them. If there is more than one color channel in the input file, the color map in the input file should have the same number of channels. If the input file has a single color channel, the output file will have the same number of color channels as the color map.

Each pixel in the input file is mapped as follows: For a multi-channel input file, a pixel in channel $i$ is mapped as $\text{map}[i][\text{pixel}] >\!> 8$, producing a pixel in output channel $i$. The right shift takes the 16 bit color map value to an 8 bit pixel value. For a single channel input file, to produce a pixel in output channel $i$ is produced from the corresponding input pixel value as $\text{map}[i][\text{pixel}] >\!> 8$.

**OPTIONS**

- `-1` This option will cause a linear (identity) color map to be loaded into the output file. Otherwise, the output file will have no color map.

- `infile` The input will be read from this file, otherwise, input will be taken from stdin.

- `-o outfile` If specified, output will be written to this file, otherwise it will go to stdout.

**SEE ALSO**

`rleldmap(1), urt(1), RLE(5)`.  

**AUTHOR**

Spencer W. Thomas, University of Utah

**BUGS**

If the image data and color map channels in the input file do not conform to the restriction stated above (NN or IN) the program will most likely core dump.

**AVG4**

**NAME**

`avg4` - Downfilter an image by simple averaging.

**SYNOPSIS**

`avg4 [-o outfile] [infile]`
DESCRIPTION

Avg4 downfilters an RLE image into a resulting image of 1/4th the size, by simply averaging four pixel values in the input image to produce a single pixel in the output. If the original image does not contain an alpha channel, avg4 creates one by counting the number of non-zero pixels in each group of four input pixels and using the count to produce a coverage value. While the alpha channel produced this way is crude (only four levels of coverage) it is enough to make a noticeable improvement in the edges of composited images.

OPTIONS

 infile The input will be read from this file, otherwise, input will be taken from stdin.
-o outfile If specified, output will be written to this file, otherwise it will go to stdout.

SEE ALSO

fant(1), rlecomp(1), smush(1), urt(1), RLE(5).

AUTHOR

Rod Bogart, John W. Peterson

BUGS

Very simple minded – more elaborate filters could be implemented.

CROP

NAME

crop – Change the size of an RLE image

SYNOPSIS

crop [-b ] [ xmin ymin xmax ymax ] [-o outfile ] [ infile ]

DESCRIPTION

Crop changes the size of an RLE image. The command line numbers xmin ymin xmax ymax specify the bounds of the resulting image. If the resulting image is larger than the original, crop supplies blank pixels, otherwise pixels are thrown away.

OPTIONS

-b The input image is cropped to the enclosing box. Extra rows and columns of black pixels are removed. The infile must be a file; no piped input is allowed for this option.

-o outfile If specified, output will be written to this file, otherwise it will go to stdout.

infile The input will be read from this file, otherwise, input will be taken from stdin.

SEE ALSO

repos(1), urt(1), RLE(5).

AUTHOR

Rod Bogart
BUGS

Could be combined with repos. Does not check to see if the input and output regions are disjoint.

FANT

NAME

fant - perform simple spatial transforms on an image

SYNOPSIS

fant [-a angle] [-b] [-o outfile] [-p xoff yoff] [-s xscale yscale] [-v] [infile]

DESCRIPTION

Fant rotates or scales an image by an arbitrary amount. It does this by using pixel integration (if the image size is reduced) or pixel interpolation if the image size is increased. Because it works with subpixel precision, aliasing artifacts are not introduced (hah! see BUGS). Fant uses a two-pass sampling technique to perform the transformation. If infile is "-" or absent, input is read from the standard input.

OPTIONS

-a angle Amount to rotate image by, a real number from 0 to 45 degrees (positive numbers rotate clockwise). Use rleflip(1) first to rotate an image by larger amounts.

-b Blur the resulting image. Always interpolate between pixels of the input image. This results in fewer artifacts but a slightly blurry resulting image. Recommended for pathological cases such as single pixel wide lines in the input image.

-o outfile Specifies where to place the resulting image. The default is to write to stdout. If outfile is "-", the output will be written to the standard output stream.

-p xoff yoff Specifies where the origin of the image is - the image is rotated or scaled about this point. If no origin is specified, the center of the image is used.

-s xscale yscale The amount (in real numbers) to scale an image by. This is often useful for correcting the aspect of an image for display on a frame buffer with non square pixels. For this use, the origin should be specified as 0, 0 (see -p above). If an image is only scaled in Y and no rotation is performed, fant only uses one sampling pass over the image, cutting the computation time in half.

-v Verbose output. Primarily for debugging.

SEE ALSO

avg4(1), rleflip(1), urt(1), RLE(5),


AUTHORS

John W. Peterson, James S. Painter
BUGS

*Font* uses a rather poor anti-aliasing filter (a box filter). This is usually good enough but will exhibit noticeable aliasing artifacts on nasty input images.

**GET4D**

**NAME**

`get4d` - get RLE images to a Silicon Graphics Iris/4D display

**SYNOPSIS**

```
get4d [ -D ] [ -f ] [ -{GS} ] [ -g disp.gamma ] [ -{Ii} in.age.gamma ] [ -n ] [ -p xpos ypos ] [ -s xsize ysize ] [ -w ] [ infile ]
```

**DESCRIPTION**

This program displays an RLE(5) file on a Silicon Graphics Iris/4D display or IBM RS6000 with the GL library.

The default behavior is to display the image in RGB color. An option is provided to force black and white display. There is currently no support in `get4d` for non-24-bit color (lookup table modes), but the `getmex` program should work on 8-bit 4D's which cannot do RGB display.

The GT graphics fast pixel access routines are used by default on 4D/GT and GTX machines, and Personal Iris. The `-G` option is provided to force this mode, if the string returned by the `gversion(3g)` function changes, or is different on future 4D's.

The penalty of GT mode is not being able to resize or pan the window, but redisplay is so fast that there is no need to do so. You can also go into "slow mode" on GT machines by giving the `-S` flag. Slow mode allows resizing the window and panning with the mouse.

**OPTIONS**

- `-p xpos ypos` Position of the lower left corner of the window.
- `-s xsize ysize` Initial size of the window (slow mode only.)
- `-f` Normally, `get4d` will fork itself after putting the image on the screen, so that the parent process may return the shell, leaving an "invisible" child to keep the image refreshed. If `-f` is specified, `get4d` will remain attached to the shell, whence it may be killed with an interrupt signal. In either case the window manager "quit" menu button can be used to kill `get4d`.

- `-g display.gamma` Specify the gamma of the display monitor. If this flag is not specified, `get4d` looks in the user's home directory for a `.gamma` file. This file is produced by the `gamma(1g)` SGI command (This is not done on the IBM R6000). The value in the .gamma file is used to determine the gamma of the display by calculating \((2.4 / \text{gamma-value})\) and using that as the `disp.gamma`.

- `-i image.gamma` Specify the gamma (contrast) of the image. A low contrast image, suited for direct display without compensation on a high contrast monitor (as most monitors are) will have a gamma of less than one. The default image gamma is 1.0. Image gamma may also be specified by a picture comment in the RLE (5) file of the form `image.gamma=gamma`. The command line argument will override the value in the file if specified.

- `-I image.gamma` An alternate method of specifying the image gamma, the number following `-I` is the gamma of the display for which the image was originally computed (and is therefore 1.0 divided by the actual gamma of the image). Image display gamma
may also be specified by a picture comment in the RLE (5) file of the form display:gamma=gamma. The command line argument will override the value in the file if specified.

-n Do not draw a window border.

-w This flag forces get4d to produce a gray scale dithered image instead of a color image. Color input will be transformed to black and white via the NTSC Y transform.

-D "Debug mode". The operations in the input RLE(5) file will be printed as they are read.

file Name of the RLE(5) file to display. If not specified, the image will be read from the standard input.

In "slow mode" you can "pan" a small window around in an image by clicking the left mouse button in the image. The position in the image under the cursor will jump to the center of the window. The F9 key or Alt keys reset the view to position the center of the image in the center of the window. Furthermore, control-F9 (or control-Alt) saves the current view, and shift-F9 (or shift-Alt) restores it.

NOTE

If you have a shaded image that looks "too dark", it is probably because the gamma is not set on the display. (The default gamma is 1, which assumes that gamma compensation will be done once and for all by programs producing images.) gamma 2 is better when the image producing program does not do the gamma correction. You may want to put a gamma command in your .login file.

SEE ALSO
gemex(1), urt(1), gversion(3g), gamma(1g), RLE(5).

AUTHOR

Russ Fish, University of Utah. Based on getX, by Spencer W. Thomas.

GIFTORLE

NAME
giftorle - Convert GIF images to RLE format

SYNOPSIS
giftorle [ -c ] [ -o outfile.rle ] [ infile.gif ... ]

DESCRIPTION

Giftorle converts a file from Graphics Interchange Format (GIF) format into RLE format. Images converted with giftorle will need to be flipped with rleflip -v for correct presentation. Multiple input images may be converted, these will be written sequentially to the output RLE file.
OPTIONS

- Preserve the colormap that the GIF image contains, otherwise the colormap is applied to input image.

-o outfile.rle If specified, the output will be written to this file. If outfile.rle is "-", or if it is not specified, the output will be written to the standard output stream.

infile.gif ... The input will be read from these files. If infile.gif is "-" or is not specified, the input will be read from the standard input stream.

MISC

GIF and Graphics Interchange Format are both trademarks of CompuServe Incorporated.

SEE ALSO

rletogzf(1), rlefzip(1), urt(1), RLE(5).

AUTHOR

David Koblas (koblas@mips.com or koblas@cs.uoregon.edu)

BUGS

Should probably flip the image itself (or at least have an option).

GRAYTORLE

NAME

graytorle - Merges gray scale images into an RLE format file.

SYNOPSIS

graytorle [ -a ] [ -h hdrsize ] [ -o outfile ] zsize ysize infiles

DESCRIPTION

Graytorle reads a list of 8-bit gray scale images in unencoded binary format and converts them to an RLE(5) image with the number of channels corresponding to the number of input files. A command line option allows specifying one of the files as an alpha channel.

OPTIONS

-a Designates the first file in the input list as being information for the alpha channel of the image.

-h hdrsize Often gray scale image files include some sort of header information. This option allows specification of a count of bytes to discard at the beginning of each input file.

zsize Specifies the horizontal resolution of the input files.

y whole Specifies the vertical resolution of the input files.

-o outfile This option is used to name the output file. Otherwise, output is written to stdout.

infiles List of input files.
APPENDIX H. UTILITY PROGRAMS

SEE ALSO

rletogray(1), url(1), RLE(5).

AUTHOR

Michael J. Banks, University of Utah.

INTO

NAME

into - copy into a file without destroying it

SYNOPSIS

into [-f] outfile

DESCRIPTION

Into copies its standard input into the specified outfile, but doesn't actually modify the file until it gets EOF. This is useful in a pipeline for putting stuff back in the "same place." The outfile is not overwritten if that would make it zero length, unless the -f option is given. That option also forces overwriting of the outfile even if it is not directly writable (as long as the directory is writable).

SEE ALSO

pipe(2)

BUGS

For efficiency reasons, the directory containing the outfile must be writable by the invoker. The original outfile's owner is not preserved.

MCUT

NAME

mcut - Quantize colors in an image using the median cut algorithm

SYNOPSIS

mcut [-n colors] [-d] [-o outfile] infile

DESCRIPTION

Mcut reads an RLE file and tries to choose the "best" subset of colors to represent the colors present in the original image. A common use for this is to display a 24 bit image on a frame buffer with only eight bits per pixel using a 24 bit color map. Mcut first quantizes intensity values from eight bits to five bits, and then chooses the colors from this space. Mcut runs in two passes; the first pass scans the image to find the color distributions, and the second pass maps the original colors into color map indices. The output file has a color map containing the colors mcut has chosen. Mcut also sets the picture comment "color_map_length" equal to the number of colors it has chosen. The getxil program (among others) will use this color map instead of dithering.
OPTIONS

-\( n \) \( n\)colors Limit the number of colors chosen to \( n\)colors. The default is 200.

-\( d \) Uses Floyd/Steinberg dither to hide contouring. Greatly improves images that have a small number of colors.

\( \text{infile} \) The input will be read from this file. If it is a multi-image file, each image will be quantized to its own colormap. Piped input is not allowed.

-\( o \) \( \text{outfile} \) If specified, output will be written to this file, otherwise it will go to stdout.

SEE ALSO


AUTHOR

Robert Mecklenburg, John W. Peterson, University of Utah.

BUGS

The initial quantization is hardwired to five bits. This should be an option.

MERGECHAN

NAME

mergechan - merge channels from several RLE files into a single output stream

SYNOPSIS

mergechan [-a] [-o \( \text{outfile} \)] \( \text{infiles} \) ...

DESCRIPTION

Mergechan takes input from several RLE files and combines them into a single output stream. Each channel in the output stream comes from the respective filename specified on the input (i.e., channel zero is taken from the first file, channel one from the next, etc). The same file can be specified more than once. If the \(-a\) flag is given, the channels are numbered from -1 (the alpha channel) instead of zero. All of the input channels must have exactly the same dimensions (use \( \text{crop}(1) \) to adjust files to fit each other).

Mergechan is typically used to introduce an alpha mask from another source into an image, or combine color channels digitized independently.

If \(-o\) is specified, the output will be written to \( \text{outfile} \).

SEE ALSO

crop(1), rleswap(1), urt(1), RLE(5).

AUTHOR

John W. Peterson, University of Utah.
BUGS

Mergechan is totally ignorant of the color maps of the input files.
The restriction that all input files must be the same size could probably be removed.

PAINTTORLE

NAME

painttorle - Convert MacPaint images to RLE format.

SYNOPSIS

```
painttorle [ -c [ red ] [ green ] [ blue ] [ alpha ] ] [ -r ] [ -o outfile.rle ] [ infile.paint ]
```

DESCRIPTION

Painttorle converts a file from MacPaint format into RLE format. Because MacPaint and RLE disagree on which end is up, the output should be sent through rlefip to preserve orientation.

OPTIONS

- `c [ red ] [ green ] [ blue ] [ alpha ]` Allows the color values to be specified (the default is 255).
- `r` Invert the color of the MacPaint pixels (reverse video).
- `infile.paint` The input paint data will be read from this file, otherwise, input will be taken from stdin.
- `o outfile.rle` If specified, output will be written to this file, otherwise it will go to stdout.

SEE ALSO

rietopaint(1), urt(1), RLE(5).

AUTHOR

John W. Peterson

PGMTORLE

NAME

pgmtorle - convert a pbmplus/pgm image file into an RLE image file.

SYNOPSIS

```
pgmtorle [ -h ] [ -v ] [ -a ] [ -o outfile ] [ filename ]
```

DESCRIPTION

This program converts PBMPLUS grayscale (pgm) image files into Utah RLE(5) image files. PBMPLUS/pgm image files contain the image dimensions and 8-bit pixels with no matte or alpha data. When converting to an RLE file, the alpha channel may optionally be computed. The RLE file will contain a "grayscale" image (8 bits) with no colormap. The origins of PBMPLUS and Utah RLE files are in the upper left and lower left corners respectively. Therefore, it is common practice to send the output of this program through the "rlefip" utility - see examples below. These RLE files may then be viewed using any RLE image viewer.
OPTIONS

-\texttt{-v} This option will cause pgmtorle to operate in verbose mode. The header information is written to "stderr". Actually, there is not much header information stored in a PBMPLUS file so this information is minimal.

-\texttt{-h} This option allows the header of the PBMPLUS image to be dumped to "stderr" without converting the file. It is equivalent to using the -\texttt{-v} option except that no file conversion takes place.

-\texttt{-a} This option will cause pgmtorle to use the grayscale data to compute an alpha channel in the resulting RLE file. For any non-zero grayscale data, the alpha channel will contain a value of 255. The resulting RLE image file will contain one color channel and one alpha channel.

-\texttt{-o outfile} If specified, the output will be written to this file. If outfile is "-", or if it is not specified, the output will be written to the standard output stream.

infile.pgm The name of the PBMPLUS image data file to be converted. This file must end in "\texttt{.pgm}". However, it is not necessary to supply the "\texttt{.pgm}" extension as it will be added to the supplied name if not already there.

EXAMPLES

pgmtorle -v test.pgm -o test.rle While running in verbose mode, convert test.pgm to RLE format and store resulting data in test.rle.

pgmtorle test | rleflip -v > test.rle Convert test.pgm to RLE format and flip its contents so that it may be correctly displayed.

pgmtorle -h test Dump the header information of the PBMPLUS file called test.pgm.

SEE ALSO

ppmtorle(1), rleoppm(1), urt(1), RLE(5).

AUTHOR

Wesley C. Barris
Army High Performance Computing Research Center (AHPCRC)
Minnesota Supercomputer Center, Inc.

PPMTORLE

NAME

ppmtorle – convert a PBMPLUS/ppm image file into an RLE image file.

SYNOPSIS

ppmtorle [ -h ] [ -v ] [ -a ] [ -o outfile ] [ infile.ppm ]
APPENDIX H. UTILITY PROGRAMS

DESCRIPTION

This program converts PBMPPLUS full-color (ppm) image files into Utah RLE(5) image files. PBMPPLUS/ppm image files contain the image dimensions and image data in the form of RGB triplets. When converting to an RLE file, the alpha channel may be optionally computed. The origins of PBMPPLUS and Utah RLE files are in the upper left and lower left corners respectively. Therefore, it is common practice to send the output of this program through the "rleflip" utility — see examples below.

The RLE file will contain a "true color" image (24 bits). These RLE files may then be viewed using any RLE image viewer. When they are displayed on an 8 bit display, the image must be dithered. In order to produce a better looking image (on 8 bit displays), it is recommended that the image be quantizing (to 8 bit mapped color) prior to its display. This may be done by piping the output of this program into the Utah mcut(1) or rlequant(1) utilities. An example of this is shown later.

OPTIONS

- v This option will cause ppmtorle to operate in verbose mode. The header information is written to "stderr". Actually, there is not much header information stored in a PBMPPLUS file, so this information is minimal.

- h This option allows the header of the PBMPPLUS image to be dumped to "stderr" without converting the file. It is equivalent to using the -v option except that no file conversion takes place.

- m This option will cause ppmtorle to use the RGB data to compute an alpha channel in the resulting RLE file. For any non-zero RGB data, the alpha channel will contain a value of 255. The resulting RLE image file will contain three color channels and an alpha channel.

- o outfile If specified, the output will be written to this file. If outfile is "-", or if it is not specified, the output will be written to the standard output stream.

infile.ppm The name of the PBMPPLUS image data file to be converted. This file must end in ".ppmX". However, it is not necessary to supply the ".ppm" extension as it will be added to the supplied name if not already there.

EXAMPLES

ppmtorle -v test.ppm -o test.rle While running in verbose mode, convert test.ppm to RLE format and store resulting data in test.rle.

ppmtorle test | rleflip -v | mcut >test.rle Convert test.ppm to RLE format, flip if vertically, and convert to 8 bit mapped color before storing data in test.rle.

ppmtorle -h test Dump the header information of the PBMPPLUS file called test.ppm.

SEE ALSO

mcut(1), ppmtorle(1), rlequant(1), rletoppm(1), urt(1), RLE(5).

AUTHOR

Wesley C. Barris
Army High Performance Computing Research Center (AHPCRC)
Minnesota Supercomputer Center, Inc.
**PYRMASK**

**NAME**

`pyrmask` – Blend two images together using Gaussian pyramids.

**SYNOPSIS**

`pyrmask [-l levels] [-o outfile] inmask outmask maskfile`

**DESCRIPTION**

`Pyrmask` blends two images together by first breaking the images down into separate bandpass images, combining these separate images, and then adding the new bandpass images back into a single output image. This can produce very seamless blends of digital images. The two images are combined on the basis of a third "mask" image. The resulting image will contain the `inmask` image where the mask contains a maximum value (255) and the `outmask` image where the mask contains zeros. This is done on a channel by channel basis, i.e. the maskfile should have data in each channel describing how to combine each channel of the `inmask` and `outmask` images. All three images must have exactly the same dimensions (both image size and number of channels). For best results, it’s often useful to filter the mask image a little with `smush(1)` first.

**OPTIONS**

- `[-l levels]` How many pyramid levels to use (maximum is log(2) of image size).
- `[-o outfile]` If specified, output will be written to this file, otherwise it will go to stdout.

**SEE ALSO**


**AUTHOR**

Rod Bogart

**BUGS**

The current implementation has very strict requirements for image sizes and dimensions. The extensive use of floating point computation makes it very slow for normal sized images. It also keeps all of the bandpass images in core at once, which requires considerable amounts of memory.

Pyrmask is built on top of a library of functions for working with Gaussian pyramids. This library has yet to be documented or fully tested.

**RASTORLE**

**NAME**

`rastorle` – Convert sun raster file image to an RLE format file.

**SYNOPSIS**

`rastorle [-o outfile] [infile.ras]`
DESCRIPTION

\textit{Rastorle} converts a sun raster file to a file in the Utah Raster Toolkit RLE format.

OPTIONS

\texttt{infil.e.ras} \quad The input sun raster will be read from this file, otherwise, input will be taken from stdin.

\texttt{-o outfile} \quad If specified, output will be written to this file, otherwise it will go to stdout. The output should be run through \textit{rleflip(1)}, (-v) since RLE and Sun raster files disagree on where the origin is.

SEE ALSO

\textit{rleflip(1)}, \textit{rletorast(1)}, \textit{urt(1)}, \textit{RLE(5)}.

AUTHOR

Berry Kercheval

BUGS

It always produces a 24 bit output file, even from a pseudo-colored 8 bit input file. Thus, the original color set is "lost".

\textbf{RAWTORLE}

NAME

\texttt{rawtorle} \quad Convert raw image data to RLE.

SYNOPSIS

\texttt{rawtorle [ -N ] [ -s ] [ -r ] [ -w width ] [ -h height ] [ -f header-size ] [ -t trailer-size ] [ -n nchannels ] [ -a [ alpha-value ] ] [ -p scanline-pad ] [ -l left-scanline-pad ] [ -o outfile ] [ infil.e ]}

DESCRIPTION

This program is used to convert image data in any of a number of "raw" forms to the \textit{RLE(5)} format. The expected input size is computed from the arguments, so that several images may be concatenated together and will be processed in sequence. In this case, the output file will contain several RLE images.

OPTIONS

\texttt{-N} \quad The input is in non-interleaved order, as might be generated by the command cat \texttt{pic.r pic.g pic.b}

\texttt{-s} \quad The input is in scanline interleaved order.

\texttt{-r} \quad Reverse the channel order. (E.g., data will be interpreted as ABGR instead of RGBA.)

\texttt{-wwidth} \quad Specify the width of the input image.

\texttt{-hheight} \quad Specify the height of the input image.

\texttt{-fheader-size} \quad This many bytes will be skipped before starting to read image data.

\texttt{-ttrailer-size} \quad This many bytes will be skipped at the end of the image data.
H.3. UTAH RASTER TOOLKIT (RLE) PROGRAMS

-m This option will cause `rlatorle` to ignore the RGB data and use the matte channel information to produce a monochrome image. The resulting RLE image file will contain only one color channel instead of the usual four (RGB + alpha).

-0 outfile If specified, the output will be written to this file. If `outfile` is "-", or if it is not specified, the output will be written to the standard output stream.

infile.rla The name of the Wavefront image data file to be converted. It is not necessary to supply the "rla" extension as it will be added to the supplied name if not already there.

EXAMPLES

```
rlatorle -v test.0001.rla -o test.rle
While running in verbose mode, convert test.0001.rla to RLE format and store resulting data in test.rle.

rlatorle test.0001.rla | mcut >test.rle
Convert test.0001.rla to RLE format and convert to 8 bit mapped color before storing data in test.rle

rlatorle -h test.0001.rla
Dump the header information of the Wavefront file called test.0001.rla.
```

SEE ALSO

`mcut(1), rlequant(1), rletorla(1), urt(1), RLE(5)`

AUTHOR

Wesley C. Barris
Army High Performance Computing Research Center (AHPCRC)
Minnesota Supercomputer Center, Inc.

RLEADDCOM

NAME

`rleaddcom` - add picture comments to an RLE file.

SYNOPSIS

```
rleaddcom [ -d ] [ -i ] [ -o outfile ] infile comments
```

DESCRIPTION

The `rleaddcom` program will add one or more comments to an RLE(5) file. If `infile` is "-", it will read from the standard input. The modified RLE(5) file is written to the standard output if the `-o outfile` option is not given. All remaining arguments on the command line are taken as comments. Comments are nominally of the form `name=value` or `name`. Any comment already in the file with the same `name` will be replaced.

OPTIONS

-`d` Will cause matching comments to be deleted, no comments will be added in this case.
"In place." The input file will be rewritten with the added comments. This argument requires write permission to the directory containing infile, but does not require write permission for infile. Of the special file name cases described in urt(1), only compressed files may be updated in place. (It doesn’t make sense to update the output of a pipe "in place", does it?)

If -o outfile is specified together with -i, then outfile will not be modified until rleaddcom has finished (this is similar to the way that into(1) works).

SEE ALSO
into(1), rlehdr(1), urt(1), RLE(5).

AUTHOR
Spencer W. Thomas, University of Utah

RLEBG

NAME
rlebg - generate simple backgrounds

SYNOPSIS

DESCRIPTION
rlebg generates a simple background. These are typically used for compositing below other images. The values red green blue specify the pixel values (between 0 and 255) the background will have. If alpha is not specified, it defaults to 255 (full coverage). rlebg generates both constant backgrounds and backgrounds with continuous ramps.

OPTIONS
-s xsize ysize This is the size of the background image. The default is 512480.
-l Generate a linear ramp of pixel values. If no ramp flag is given, rlebg generates a constant background.
-v top bottom Generate a variable ramp, using a quadratic function (this looks best with gamma corrected images). top and bottom are the fractions of the full color values at the top and bottom of the image. The defaults are 1.0 0.1, respectively. If both -v and -l are given, then a linear ramp function is used instead of a quadratic ramp.
-o outfile If specified, the output will be written to this file. If outfile is "-", or if it is not specified, the output will be written to the standard output stream.

SEE ALSO
rlesetbg(1), urt(1), RLE(5).
H.3. **UTAH RASTER TOOLKIT (RLE) PROGRAMS**

**AUTHOR**

Rod Bogart

---

**RLEBOX**

**NAME**

rlebox – print bounding box for image in an RLE file.

**SYNOPSIS**

rlebox [ -c ] [ -m margin | [ -v ] [ infile ]

**DESCRIPTION**

This program prints the bounding box for the image portion of an RLE(5) file. This is distinct from the bounds in the file header, since it is computed solely on the basis of the actual image. All background pixels are ignored.

**OPTIONS**

- **-c** Print the numbers in the order that crop wants them on its command line. The default order is xmin xmax ymin ymax. If this option is specified, the bounds are printed in the order xmin ymin xmax ymax. Thus, a file foo.rle could be trimmed to the smallest possible image by the command

  crop 'rlebox -c foo.rle' foo.rle

- **-m** margin  Pads the output values by the margin given.

- **-v** Verbose mode: label the numbers for human consumption.

- **infile** Name of the RLE file (defaults to standard input).

**SEE ALSO**

crop(1), urt(1), RLE(5).

**AUTHOR**

Spencer W. Thomas, University of Utah

---

**RLECOMP**

**NAME**

rlecomp – Digital image compositor

**SYNOPSIS**

rlecomp [ -o outfile ] Afile operator Bfile
DESCRIPTION

`rlecomp` implements an image compositor based on presence of an alpha, or matte channel the image. This extra channel usually defines a mask which represents a sort of a cookie-cutter for the image. This is the case when alpha is 255 (full coverage) for pixels inside the shape, zero outside, and between zero and 255 on the boundary. If `Afile` or `Bfile` is just a single -, then `rlecomp` reads that file from the standard input.

The operations behave as follows (assuming the operation is "A operator B"):

- **over**: The result will be the union of the two image shapes, with A obscuring B in the region of overlap.
- **in**: The result is simply the image A cut by the shape of B. None of the image data of B will be in the result.
- **atop**: The result is the same shape as image B, with A obscuring B where the image shapes overlap. Note this differs from over because the portion of A outside B's shape does not appear in the result.
- **out**: The result image is image A with the shape of B cut out.
- **xor**: The result is the image data from both images that is outside the overlap region. The overlap region will be blank.
- **plus**: The result is just the sum of the image data. Output values are clipped to 255 (no overflow). This operation is actually independent of the alpha channels.
- **minus**: The result of A − B, with underflow clipped to zero. The alpha channel is ignored (set to 255, full coverage).
- **diff**: The result of A − B, with underflow wrapping around. This is useful for comparing two very similar images.

SEE ALSO


AUTHORS

Rod Bogart and John W. Peterson

BUGS

The other operations could be optimized as much as `over` is.

Rlecomp assumes both input files have the same number of channels.

RLEFLIP

NAME

`rleflip` - Invert, reflect or rotate an image.

SYNOPSIS

`rleflip -{rlhv} [ -o outfile ] [ infile ]`
DESCRIPTION

*Rlefip* inverts, reflects an image; or rotates left or right by 90 degrees. The picture’s origin remains the same. If no input file is specified, the image is read from standard input. For rotations of other than 90 degrees, use *fant*(1).

OPTIONS

Exactly one of the following flags must be given:

- *r* Rotate the image 90 degrees to the right
- *l* Rotate the image 90 degrees to the left
- *h* Reflect the image horizontally
- *v* Flip the image vertically

- *o outfile* If specified, the output will be written to this file. If *outfile* is "-", or if it is not specified, the output will be written to the standard output stream.

SEE ALSO

*fant*(1), *urt*(1), *RLE*(5).

AUTHOR

John W. Peterson

**RLEHDR**

NAME

*rlehdr* – Prints the header of an RLE file

SYNOPSIS

*rlehdr* [-d] [-m] [infile]

DESCRIPTION

This program prints the header of an *RLE*(5) file in a human readable form. If the optional *infile* argument is not supplied, input is read from standard input.

OPTIONS

- *d* Dump a very verbose version of the image contents as text to the standard error output stream.
- *m* Print out the color map information.

SEE ALSO

*urt*(1), *RLE*(5).

AUTHOR

Spencer W. Thomas, University of Utah
**RLEHISTO**

**NAME**

rlehisto – generate histogram of RLE image.

**SYNOPSIS**

```
rlehisto [-b] [-c] [-t] [-h height] [-o outfile] [infile]
```

**DESCRIPTION**

*Rlehisto* counts the pixel values in an RLE file, producing an RLE file graphing frequency of occurrence. The horizontal axis runs from pixel value 0 on the left to pixel value 255 on the right. The height indicates the number of pixels seen for each pixel value. Histograms are computed independently for each channel, scaled identically, and then overlaid.

The following options are available:

- `-b` Don’t count the background pixel values when scaling the histogram. This is useful if most pixels are colored the background color, so that the interesting part of the histogram would be too small. This option is ignored if `-c` is specified.

- `-c` Output cumulative values instead of discrete values.

- `-t` Print the totals instead of generating the histogram as an RLE file.

- `-h` *height* Scale the output image to the specified height. The default is 256.

- `-o` *outfile* Direct the output to *outfile*.

**SEE ALSO**

`urt(1), RLE(5)`.

**AUTHORS**

Gregg Townsend, University of Arizona; Rod Bogart, University of Utah.

---

**RLELDINDEX**

**NAME**

rleldmap – Load a new color map into an RLE file.

**SYNOPSIS**

```
rleldmap [ -{ab} ] [ -n nchan length ] [ -s bits ] [ -l factor ] [ -g gamma ] [ -{tf} file ] [ -m files ... ] [ -r rlefile ] [ -o outfile ] [ infile ]
```

**DESCRIPTION**

The program will load a specified color map into an RLE(5) file. The color map may be computed by *rleldmap* or loaded from a file in one of several formats. The input is read from *infile* or stdin if no file is given, and the result is written to *outfile* or stdout.

The following terms are used in the description of the program and its options:

- **input map**: A color map already in the input RLE file.

- **applied map**: The color map specified by the arguments to *rleldmap*. This map will be applied to or will replace the input map to produce the output map.
output map: Unless \(-a\) or \(-b\) is specified, this is equal to the applied map. Otherwise it will be the composition of the input and applied maps.

map composition: If the applied map is composed after the input map, then the output map will be applied map[input map]. Composing the applied map before the input map produces an output map equal to input map[applied map]. The maps being composed must either have the same number of channels, or one of them must have only one channel. If an entry in the map being used as a subscript is larger than the length of the map being subscripted, the output value is equal to the subscript value. The output map will be the same length as the subscript map and will have the number of channels that is the larger of the two. If the input map is used as a subscript, it will be downshifted the correct number of bits to serve as a subscript for the applied map (since the color map in an RLE(5) file is always stored left justified in 16 bit words). This also applies to the applied map if it is taken from an RLE(5) file (\(-r\) option below). Note that if there is no input map, that the result of composition will be exactly the applied map.

nchan: The number of separate lookup tables (channels) making up the color map. This defaults to 3.

length: The number of entries in each channel of the color map. The default is 256.

bits: The size of each color map entry in bits. The default value is the log base 2 of the length.

range: The maximum value of a color map entry, equal to \(2^{**}\text{bits} - 1\).

OPTIONS

\(-a\) Compose the applied map after the input map.

\(-b\) Compose the applied map before the input map. Only one of \(-a\) or \(-b\) may be specified.

\(-n\ nchan\ length\) Specify the size of the applied map if it is not 3x256. The length should be a power of two, and will be rounded up if necessary. If applying the map nchan must be either 1 or equal to the number of channels in the input map. It may have any value if the input map has one channel or is not present.

\(-s\ bits\) Specify the size in bits of the color map entries. I.e., only the top bits bits of each color map entry will be set. Exactly one of the options \(-l, -g, -t, -f, -m,\) or \(-r\), must be specified.

\(-l\ factor\) Generate a linear applied map with the nth entry equal to \(range * \min(1.0, factor*(n/(length-1)))\). Factor defaults to 1.0 if not specified. Negative values of factor will generate a map with values equal to \(range * \max(0.0, 1.0 - factor*(n/(length-1)))\).

\(-g\ gamma\) Generate an applied map to compensate for a display with the given gamma. The nth entry is equal to \(range * (n/(length-1))**((1/gamma))\).

\(-t\ file\) Read color map entries from a table in a text file. The values for each channel of a particular entry follow each other in the file. Thus, for an RGB color map, the file would look like:
red0 green0 blue0
red1 green1 blue1
... ...
Line breaks in the input file are irrelevant.
-f file  Reads the applied map from a text file, with all the entries for each channel following each other. Thus, the input file above would appear as
red0 red1 red2 ... (length values)
green0 green1 green2 ... (length values)
blue0 blue1 blue2 ... (length values)
As above, line breaks are irrelevant.

-m files ... Read the color map for each channel from a separate file. The number of files specified must equal the number of channels in the applied map. (Note: the list of files must be followed by another flag argument or by the null flag — to separate it from the infile specification.

-o outfile  The output will be written to the file outfile if this option is specified. Otherwise the output will go to stdout.

infile  The input will be taken from this file if specified. Otherwise, the input will be read from stdin.

SEE ALSO
applymap(1), urt(1), RLE(5).

AUTHOR
Spencer W. Thomas, University of Utah

RLENOISE

NAME
rlenoise – Add random noise to an image

SYNOPSIS
rlenoise [ -n amount ] [ -o outfile ] [ infile ]

DESCRIPTION
Rlenoise adds uniform random noise to an image. The peak-to-peak amplitude of the noise can be specified with the -n flag, the default value is 4. This program may be useful for trying to deal with quantization in an output device, if you are able to trade spatial resolution for color resolution, and you don’t have a good characterization of the quantization function.

OPTIONS
infile  The input will be read from this file, otherwise, input will be taken from stdin.

-o outfile  If specified, output will be written to this file, otherwise it will go to stdout.

SEE ALSO
urt(1), RLE(5).

AUTHOR
Spencer W. Thomas, University of Michigan.

BUGS
Of limited utility.
H.3. UTAH RASTER TOOLKIT (RLE) PROGRAMS

RLEPATCH

NAME

rlepatch - patch smaller RLE files over a larger image.

SYNOPSIS

rlepatch [-o outfile] infile patchfiles...

DESCRIPTION

Rlepatch puts smaller RLE files on top of a larger RLE image. One use for rlepatch is to place small "fix" images on top of a larger image that took a long time to compute. Along with repos(1), rlepatch can also be used as a simple way to build image mosaics. Unlike rlecomp(1), rlepatch does not perform any arithmetic on the pixels. If the patch images overlap, the patches specified last cover those specified first.

If the input files each contain multiple images, they are treated as streams of images merging to form a stream of output images. I.e., the nth image in each input file becomes part of the nth image in the output file. The process ceases as soon as any input file reaches its end.

OPTIONS

infile The background image will be read from this file. If input is to be taken from stdin, "-" must be specified here.

-o outfile If specified, output will be written to this file, otherwise it will go to stdout.

SEE ALSO

rlecomp(1), crop(1), repos(1), urt(1), RLE(5).

AUTHOR

John W. Peterson, University of Utah.

BUGS

Rlepatch uses the "row" interface to the RLE library. It would run much faster using the "raw" interface, particularly for placing small patches over a large image. Even fixing it to work like rlecomp (which uses "raw" mode only for non-overlapping images) would make a major improvement.

RLEQUANT

NAME

rlequant - variance based color quantization for RLE images

SYNOPSIS

rlequant [-b bits] [-c colors] [-d] [-f] [-o outfile] [infile]
DESCRIPTION

This program quantizes the colors in an RLE image using a variance-based method. See colorquant(3) for more details on the method.

-b bits The colors in the input image will be "prequantized" to this many bits before applying the variance-based method. Two internal tables of size $2^*(3*bits)$ are allocated, so values of bits greater than 6 are likely to cause thrashing or may prevent the program from running at all. The default value of bits is 5. It must be less than or equal to 8 and greater than 0.

-c colors The output image will be quantized to at most colors colors. It might have fewer if the input image has only a few colors itself. The default value of colors is 256. It must be less than or equal to 256.

-d Floyd Steinberg dithering is performed on the output. This is very helpful for images being quantized to a small number of colors.

-f If this option is specified, a faster approximation will be used. In most cases, the error so introduced will be barely noticeable.

-o outfile The output will be written to the file outfile. If not specified, or if outfile is "-", the output will be written to the standard output stream.

infile This file contains one or more concatenated RLE images. Each will be processed in turn. A separate quantization map will be constructed for each image. If not specified, or if infile is "-", the image(s) will be read from the standard input stream.

SEE ALSO

mcut(1), rledither(1), urt(1), colorquant(3), RLE(5).

AUTHOR

Spencer W. Thomas
Craig Kolb (Yale University) wrote the color quantization code.
Rod Bogart wrote the dithering code.

BUGS

If you specify -d, it needs to compute a complete inverse color map. This takes a long time (especially if you don’t specify -f). That’s life.

RLESCALE

NAME

rlescale - produce gray scale images.

SYNOPSIS

rlescale [-c] [-n nsteps] [-o outfile] [zsize] [ysize]

DESCRIPTION

Rlescale produces an RLE image containing a (more-or-less) standard gray scale image. Along the bottom are 8 colored patches (in the standard primary and secondary colors). Above these are a sequences of logarithmically scaled gray patches. By default, a 16 step scale is produced. The size of the output file (default 512 by 480) can be set with the zsize and ysize arguments.
OPTIONS

- `c` Produce red, green, blue, and gray scales.
- `n nsteps` Specify the number of steps to be produced.

SEE ALSO

`ur1(1), RLE(5)`

AUTHOR

Spencer W. Thomas, University of Michigan.

BUGS

Can't make an image narrower than 3 * nsteps pixels wide.

RLESELECT

NAME

`rleselect` – Select images from an RLE file.

SYNOPSIS

```
rleselect [-i infile] [-o outfile] [-v] [image-numbers ...]
```

DESCRIPTION

This program selects images from an `RLE(5)` file containing multiple concatenated images. The selected images are specified by number; the first image in the file is number 1. A negative number in the `image-numbers` list means that all images from the previous number in the list to the absolute value of this number should be included. A zero in the list is taken as '-infinity', so that all images from the previous number to the last image in the file will be included. To try to clarify this, some examples are included below.

OPTIONS

- `-i infile` The input will be read from this file. If `infile` is "-" or is not specified, the input will be read from the standard input stream.
- `-o outfile` If specified, the output will be written to this file. If `outfile` is "-", or if it is not specified, the output will be written to the standard output stream.
- `-v` Verbose output.

EXAMPLES

```
rleselect 1 4 5  Selects image 1, 4, and 5.
rleselect 4 1 5  Also selects image 1, 4, and 5.
rleselect 1 -4 5  Selects images 1 through 4 and 5 (i.e., 1 through 5).
rleselect 3 0  Selects images 3 through the last.
rleselect -4  Selects images 1 through 4.
```
APPENDIX H. UTILITY PROGRAMS

SEE ALSO

rlesplit(1), urt(1), RLE(5).

AUTHOR

Spencer W. Thomas

RLESETBG

NAME

rlesetbg - Set the background value in the RLE header.

SYNOPSIS

rlesetbg [ -D ] [ -c bgcolor ... ] [ -o outfile ] [ infile ]

DESCRIPTION

rlesetbg sets the background color field in the image header of an RLE(5) image (none of the actual pixels are changed). If infile isn't specified, the image is read from stdin.

The background color in the header is used to save space in the run-length encoded file. Runs of background-colored pixels longer than 2 pixels are simply not saved. (Doing this for runs of 1 or 2 background pixels can make the saved image larger than if no encoding were done.) Therefore, changing the background color with rlesetbg may still leave some pixels saved in the original background color. The -D option will delete the background color altogether from the header; this can be useful in certain circumstances, but can also lead to very strange results.

OPTIONS

-D Delete any background specification that might be present.

-O Specifies that the image has no background, it overlays existing images.

-c bgcolor ... Specifies the color values to set the background to. There should be at least as many values as there are color channels in the image. Use — or another option to separate the list of colors from infile.

-o outfile If specified, the output will be written to this file. If outfile is "-", or if it is not specified, the output will be written to the standard output stream.

AUTHORS

John W. Peterson and Rod Bogart

SEE ALSO

repos(1), urt(1), RLE(5).

BUGS

This should really be part of a single program that does all header munging...

RLESPIFF

NAME

rlespiff - Use simple contrast enhancement to "spiff up" an image.
**H.3. UTAH RASTER TOOLKIT (RLE) PROGRAMS**

**SYNOPSIS**

```
rlespiff [-b blacklevel] [-s] [-t threshold] [-w whitelevel] [-o outfile] [ infile ]
```

**DESCRIPTION**

`rlespiff` "spiffs up" an image by stretching the contrast range so that the darkest pixel maps to black and the lightest to white. If the `-s` flag is given, the color channels will be treated separately. This will likely cause some drastic color shifts.

**OPTIONS**

- `-b blacklevel`
  
The darkest input pixel will map to this pixel value in the output image. The default is 0.

- `-s`
  
  If specified, each color channel will be mapped separately.

- `-t threshold`
  
  This argument controls the number of samples of a pixel value that should be considered insignificant (and will therefore be ignored). It is specified in pixels/million. A threshold of 4 applied to a 512x512 image would mean that any value that existed at only one pixel would be ignored. The default value is 10.

- `-w whitelevel`
  
  The lightest input pixel will map to this pixel value in the output image. The default is 255.

- `-o outfile`
  
  If specified, the output will be written to this file. If `outfile` is ",", or if it is not specified, the output will be written to the standard output stream.

- `infile`
  
  The input will be read from this file. If `infile` is "," or is not specified, the input will be read from the standard input stream.

**SEE ALSO**

`ur7(1), RLE(5)`

**AUTHOR**

Spencer W. Thomas

**RLESPLICE**

**NAME**

`rlesplice` - Splice two RLE files together horizontally or vertically.

**SYNOPSIS**

```
rlesplice -{hv} [-c] [-o outfile] infile1 infile2
```

**DESCRIPTION**

`rlesplice` splices two RLE images together either vertically or horizontally. If one image is smaller, then its background value or black is used to pad that image to equal the larger dimension in the other image. The `-c` flag is used to specify whether the smaller image should be centered when put next to the larger. Presently the two images must have the same number of color channels, the same presence of an alpha channel, and the same colormap size and length. The colormap from the first image is used for the resultant image.
SEE ALSO

riecomp(1), rlepatch(1), unslice(1), urt(1), RLE(5).

AUTHOR

Martin R. Friedmann

RLESPLIT

NAME

rlesplit — split a file of concatenated RLE images into separate image files

SYNOPSIS

rlesplit [-n number [ digits ]] [-o prefix ] [ infile ]

DESCRIPTION

This program will split a file containing a concatenated sequence of RLE(5) images into separate files, each containing a single image. The output file names will be constructed from the input file name or a specified prefix, and a sequence number. If an input infile is specified, then the output file names will be in the form "rlefileroot.#.rle", where rlefileroot is infile with any "rle" suffix stripped off. If the option -o prefix is specified, then the output file names will be of the form "prefix.#.rle". If neither option is given, then the output file names will be in the form "#.rle". Input will be read from infile if specified, from standard input, otherwise. File names will be printed on the standard error output as they are generated.

The option -n allows specification of an initial sequence number, and optionally the number of digits used for the sequence number. By default, numbering starts at 1, and numbers are printed with 3 digits (and leading zeros).

SEE ALSO

rieselect(1), urt(1), RLE(5).

AUTHOR

Spencer W. Thomas

RLESWAP

NAME

rleswap — swap the channels in an RLE file.

SYNOPSIS

rleswap [ -v ] [ -f from-channels,... ] [ -t to-channels,... ] [ -d delete-channels,... ] [ -p channel-pairs,... ] [ -o outfile ] [ infile ]

DESCRIPTION

This program can be used to select or swap the color channels in a RLE(5) file. The major options provide four different ways of specifying a mapping between the channels in the input file and the output file. Only one of the options -f, -t, -d, or -p may be specified. If the optional infile is not given, input will be read from standard input. A new RLE(5) file will be written to the standard
output or to outfile, if specified. The output image will be similar to the input, except for the specified channel remappings.

OPTIONS

-v  Print the channel mappings that will be performed on the standard error output.

-f  Following this option is a comma separated list of numbers indicating the input channel that maps to each output channel in sequence. I.e., the first number indicates the input channel mapping to output channel 0. The alpha channel will be passed through unchanged if present. Any input channels not mentioned in the list will not appear in the output.

-t  Following this option is a comma separated list of numbers indicating the output channel to which each input channel, in sequence, will map. I.e., the first number gives the output channel to which the first input channel will map. No number may be repeated in this list. The alpha channel will be passed through unchanged if present. Any output channel not mentioned in the list will not receive image data. If there are fewer numbers in the list than there are input channels, the excess input channels will be ignored. If there are more numbers than input channels, it is an error.

-d  Following this option is a comma separated list of numbers indicating channels to be deleted from the input file. All other channels will be passed through unchanged. The alpha channel may be specified as -1.

-p  Following this option is a comma separated list of pairs of channel numbers. The first channel of each pair indicates a channel in the input file that will be mapped to the channel in the output file indicated by the second number in the pair. No output channel number may appear more than once. Any input channel not mentioned will not appear in the output file. Any output channel not mentioned will not receive image data. The alpha channel may be specified as -1.

SEE ALSO

mergechan(1), url(1), RLE(5)

AUTHOR

Spencer W. Thomas, University of Utah

RLETOABA62

NAME

rletoabA62 - Convert from RLE Format to Abekas A62 Dump Format

SYNOPSIS

rletoabA62 [ -N ] [ -f n ] [ -n n ] [ infile ]

DESCRIPTION

RletoabA62 converts a raster file in the Utah Raster Toolkit RLE format into a format suitable for writing to an Abekas A62 dump tape and subsequent loading onto the Abekas disk. The generated image is 768 pixels wide and 512 pixels high. If the input is larger, it is truncated. If it is smaller, it is padded on the top and right with black. The output is written to stdout, and should be written to a tape in 24K byte blocks with dd as in the following:

dd of=/dev/rmt8 obs=24k
Normally, the output is processed with a simple digital filter; this feature may be turned off with an option. *RletoabA62* normally writes two consecutive frames, normally starting at frame 1.

Input is taken from *stdin* unless a file name is given on the command line. Only a single file may be given, and so if multiple invocations of *rletoabA62* are performed in a script, care must be taken to tell the program to convert the data for the proper Abekas frame number (1-4). Otherwise, the colors will appear wrong; they will be rotated on a vector scope diagram.

**EXAMPLE**

The following example converts all files ending in `.rle` in the current directory and writes them to a tape. Two frames are written per image and the frame number is incremented accordingly.

```bash
frame=1
count=2
for file in *.*
do
  rletoabA62 -f $frame $file
  frame=`expr \(( $frame - 1 \) + $count \) % 4 + 1`
done
```

`dd of=/dev/rmt8 obs=24k`

**OPTIONS**

Options are parsed by `getopt(3)`.

- `-N` Do not apply digital filtering.

- `-fn` Create the first frame as Abekas frame number `n`, having a value from one to four. Consecutive frames increment this number modulo four. The default is one.

- `-nn` Write `n` frames of output, incrementing the frame number each time. The default is two.

**SEE ALSO**

`rul(1), RLE(5)`.

**AUTHOR**

Bob Brown, RIACS.

**BUGS**

This program does not preserve the aspect ratio of the input.

**RLETOASCII**

---

**NAME**

`rletoascii` – Print an RLE image as ASCII chars.

**SYNOPSIS**

```
rletoascii [-S asciistr] [-r] [-o outfile] [ infile ]
```
**DESCRIPTION**

*Rletoascii* reads a file in *RLE(5)* format, converts it to black and white, then dumps it as ASCII characters. The 0 to 255 range of pixel values in the image is scaled to the length of *asciistr* and a character at that position in the string is printed for each pixel. Input will be read from *infile* if specified, from standard input, otherwise. Output dumps to standard output, or *outfile*, if specified. Usually, the input will need to be resized by *fant(1)* or *rlezoom(1)* to make it small enough to fit on the screen and to adjust the pixel aspect ratio to the "character aspect ratio" of the terminal. To get it "right side up", use *rlefip(1)* with the -v option. Finally, it may be helpful to maximize the dynamic range with *rlespiff(1)*.

**OPTIONS**

- **-S asciistr** Specifies the range of ascii characters for conversion. The default string (@BR*#$PX0woIcv:+!" ) was designed to look good with the X 6x13 font.
- **-r** Reverse video. This causes the 0 to 255 range to be mapped to the reverse of the ascii string.

**SEE ALSO**

*fant(1)*, *rlefip(1)*, *rlespiff(1)*, *rlezoom(1)*, *urt(1)*, *RLE(5)*.

**AUTHOR**

Rod G. Bogart, University of Michigan.

**DEFCIENCIES**

Could be rewritten to use overprinting for output to a real printer.

---

**RLETOGIF**

**NAME**

*rletogif* - Convert RLE files to GIF format.

**SYNOPSIS**

`rletogif [ -o outfile.gif ] [ infile.rle ]`

**DESCRIPTION**

This program converts an *RLE(5)* image file to *GIF* format. The input file must be a single channel (8 bit) image. Three channel (24 bit) images can be converted to single channel images using the programs *tobw(1)*, *to8(1)*, *mcutf(1)*, or *rlequant(1)*. The input image will be flipped vertically, since the *GIF* origin is in the upper left, and the *RLE* origin is in the lower left. Only a single image will be converted.

**OPTIONS**

- **-o outfile.gif** If specified, the output will be written to this file. If *outfile.gif* is "-", or if it is not specified, the output will be written to the standard output stream.
- **infile.rle** The input will be read from this file. If *infile.rle* is "-" or is not specified, the input will be read from the standard input stream.
SEE ALSO

to8(1), mcut(1), rlequant(1), giftorle(1), urt(1), RLE(5).

AUTHOR

Bailey Brown, University of Michigan

RLETOGRAY

NAME

rletogray – Splits an RLE format file into gray scale images.

SYNOPSIS

rletogray [ -o prefix ] [ infile ]

DESCRIPTION

Rletogray reads a file in RLE(5) format and splits the file into unencoded binary files, one for each channel in the RLE file. The output file names will be constructed from the input file name or a specified prefix.

If an input infile is specified, then the output file names will be in the form "rlefileroot.{alpha, red, green, blue}", where rlefileroot is infile with any "rle" suffix stripped off. If the option -o prefix is specified, then the output file names will be of the form "prefix.{alpha, red, green, blue}". If neither option is given, then the output file names will be "out.{alpha, red, green, blue}". Input will be read from infile if specified, from standard input, otherwise. If more channels than just red, green, blue, and alpha are present in the input, numeric suffixes will be used for the others.

OPTIONS

- oprefix  Specifies the output file name prefix to be used.

infile  This option is used to name the input file. If not present, input is taken from stdin.

SEE ALSO

rletoraw(1), urt(1), RLE(5).

AUTHOR

Michael J. Banks, University of Utah.

RLETOPAINT

NAME

rletopaint – convert an RLE file to MacPaint format using dithering

SYNOPSIS

rletopaint [ -l ] [ -r ] [ -g gamma ] [ -o outfile.paint ] [ infile ]
DESCRIPTION

Rletopaint converts a file from RLE(5) format to MacPaint format. The program uses dithering to convert from a full 24 bit color image to a bitmapped image. If the RLE file is larger than a MacPaint image (576720) it is cropped to fit. Because MacPaint files have their coordinate origin in the upper left instead of the lower left, the RLE file should be piped through rlefip(1) -v before rletopaint. The resulting file can be downloaded to a Macintosh in binary mode, and should be given a type of PNTG and a creator of MPNT, so it will be recognized as a MacPaint file.

OPTIONS

-l Use a linear map in the conversion from 24 bits to bitmapped output.
-g [gamma] Use a gamma map of gamma (gamma is 2.0 if not specified).
-r Invert the sense of the output pixels (white on black instead of black on white). For normal images, you probably want this flag.

SEE ALSO

painttoR(1), urt(1), RLE(5).

AUTHOR

John W. Peterson. Byte compression routine by Jim Schimpf.

BUGS

Should use a color map in the file, if present.

RLETOPPM

NAME

rletoppm - convert a Utah RLE image file into a PBMPLUS/ppm image file.

SYNOPSIS

rletoppm [ -h ] [ -v ] [ infile ]

DESCRIPTION

This program converts Utah RLE(5) image files into PBMPLUS full-color (ppm) image files. Rletoppm will handle four types of RLE files: Grayscale (8 bit data, no color map), Pseudocolor (8 bit data with a color map), Truecolor (24 bit data with color map), and Directcolor (24 bit data, no color map). Since the origins for the RLE and PBMPLUS image file formats are in different locations, it may be necessary to "flip" the RLE image before converting to the PBMPLUS format - see examples.

OPTIONS

-v This option will cause rletoppm to operate in verbose mode. Header information is printed to "stderr".
-h This option allows the header of the RLE file to be dumped to "stderr" without converting the file. It is equivalent to using the -v option except that no file conversion takes place.
 infile  The input will be read from this file. If infile is "-" or is not specified, the input will be read from the standard input stream. The resulting PBMPLUS/ppm data will be sent to "stdout".

EXAMPLES

rletoppm -v lenna.rle >lenna.ppm  While running in verbose mode, convert lenna.rle to PBMPLUS/ppm format and store resulting data in lenna.ppm.

rleflip -v lenna.rle | rletoppm >lenna.ppm  Flip the contents of lenna.rle then convert to PBMPLUS/ppm format and store resulting data in lenna.ppm.

rletoppm -h test.rle  Dump the header information of the RLE file called test.rle.

SEE ALSO

ppmtorle(1), pgmtorle(1), urt(1), RLE(5)

AUTHOR

Wesley C. Barris
Army High Performance Computing Research Center (AHPCRC)
Minnesota Supercomputer Center, Inc.

RLETOPS

NAME

rletops - Convert RLE images to PostScript

SYNOPSIS

rletops [-C] [-a aspect] [-c center] [-h height] [-o outfile.ps] [-s] [infile]

DESCRIPTION

Rletops converts RLE(5) images into PostScript. The conversion uses the PostScript image operator, instructing the device to reproduce the image to the best of its abilities. If infile isn’t specified, the RLE image is read from stdin. The PostScript output is dumped to stdout, or to outfile.ps, if specified.

OPTIONS

-a aspect  Specify aspect ratio of image. Default is 1.0 (note PostScript uses square pixels).

-C  Causes a color PostScript image to be generated. This creates larger files and uses the PostScript colorimage operator, which is not recognized by all devices. The default is monochrome.

-c center  Centers the images about a point center inches from the left edge of the page (or left margin if -s is specified). Default is 4.25 inches.

-h height  Specifies the height (in inches) the image is to appear on the page. The default is three inches. The width of the image is calculated from the image height, aspect ratio, and pixel dimensions.
Specifies image is to be generated in "Scribe Mode." The image is generated without a PostScript showpage operator at the end, and the default image center is changed to 3.25 inches from the margin (which usually is 1 inch). This is to generate PostScript files that can be included in Scribe documents with the @Picture command. Images may also be included in LaTeX documents with local conventions like the \special{psfile=image.ps} command.

NOTES

On devices like the Apple LaserWriter, rletops generates large PostScript files that take a non-trivial amount of time to download and print. A 512x512 image takes about ten minutes. For including images in documents at the default sizes, 256x256 is usually sufficient resolution.

SEE ALSO

avg4(1), urt(1), RLE(5).

AUTHORS

Rod Bogart, John W. Peterson, Gregg Townsend.
Portions are based on a program by Marc Majka.

BUGS

Due to a mis-understanding with the PostScript interpreter, rletops always rounds the image size up to an even number of scanlines.

RLETORAST

NAME

rletorast — Convert an RLE file to a Sun rasterfile.

SYNOPSIS

rletorast [ -o outfile.ras ] [ infile ]

DESCRIPTION

This program converts an RLE(5) file to a Sun raster file.

-o outfile.ras If specified, the output will be written to this file. If outfile.ras is "-", or if it is not specified, the output will be written to the standard output stream. The input file should have either 1 or 3 channels, and may have an alpha channel. Depending on the input, either a gray scale or color raster file will be generated. If an alpha channel is present, a 32 bit raster will always be made.

infile The input will be read from this file. If infile is "-" or is not specified, the input will be read from the standard input stream.

The programs mcut(1), rletorast(1), to8(1), and tobw(1) will make a 1 channel RLE image from an 3 channel (full color) image. If the original image also had an alpha channel, rleswap -d -1 can be used to delete it.

SEE ALSO

mcut(1), rastorle(1), rlequant(1), rleswap(1), to8(1), tobw(1), urt(1), RLE(5).
APPENDIX H. UTILITY PROGRAMS

AUTHOR

Ed Falk, Sun Microsystems.

RLETORAW

NAME

rletoraw - Convert RLE file to raw RGB form.

SYNOPSIS

rletoraw [ -o outfile ] [ infile ]

DESCRIPTION

This program converts an RLE(5) image to a raw RGB form. The output file is a stream of pixels (RGBRGB...), in left-to-right, bottom-to-top order. The width and height of the input image will be printed on the standard error stream.

OPTIONS

- outfile
  If specified, the output will be written to this file. If outfile is "-", or if it is not specified, the output will be written to the standard output stream.

infile
  The input will be read from this file. If infile is "-" or is not specified, the input will be read from the standard input stream.

SEE ALSO

rawtorle(1), urt(1), RLE(5).

AUTHOR

Martin Friedmann

BUGS

Input files must have red, green, and blue channels. If not, bogus data will be generated in the missing colors.

RLETORLA

NAME

rletorla - convert a Utah RLE image file into a Wavefront "rlb" image file.

SYNOPSIS

rletorla [ -h ] [ -v ] [ infile ]
DESCRIPTION

This program converts Utah RLE(5) image files into Wavefront rlb image files. Rletorla will handle four types of RLE files: Grayscale (8 bit data, no color map), Pseudocolor (8 bit data with a color map), Truecolor (24 bit data with color map), and Directcolor (24 bit data, no color map). In each case the resulting Wavefront image file will contain RGB data as well as a matte channel. If no alpha channel is found in the RLE file, the Wavefront matte channel will be computed using the RGB or mapped data. The entire area of the Wavefront image will be run length encoded. The size of the Wavefront "bounding box" data structure will be set to that of the total image area.

OPTIONS

- v This option will cause rletorla to operate in verbose mode. Header information is printed to "stderr".

- h This option allows the header of the RLE file to be dumped to "stderr" without converting the file. It is equivalent to using the -v option except that no file conversion takes place.

infile The name of the RLE image data file to be converted. The name of the resulting Wavefront file will be derived from the name of the input file - the extension will be changed from "rle" to "rla". (Note: if you use the extended input file names described in urt(1), this will result in a very strange filename for the Wavefront file.

EXAMPLES

rletorla -v lenna.rle While running in verbose mode, convert lenna.rle to Wavefront rlb format and store resulting data in lenna.rla.

rletorla -h test.0001.rle Dump the header information of the RLE file called test.0001.rle.

SEE ALSO

rlatorle(1), urt(1), RLE(5).

AUTHOR

Wesley C. Barris
Army High Performance Computing Research Center (AHPCRC)
Minnesota Supercomputer Center, Inc.

RLETOTIFF

NAME

rletotiff - Convert 24 bit RLE image files to TIFF.

SYNOPSIS

rletotiff [ -{cC} ] -o outfile.tif [ -v ] [ infile.rle ]

DESCRIPTION

This program converts a 24 bit image in RLE(5) format into TIFF form. Only a single image will be converted.
OPTIONS

-ec Sets the type of compression used in the output file. -c (the default) will cause the output file to be compressed using the Lempel-Ziv-Welch (LZW) algorithm. -C will suppress any compression.

-o outfile.tif The output will be written to this file. outfile.tif must be a real file, the special cases described in utr(1) do not apply. Note also that this "option" is not optional. The -o flag is required for consistency with the other tools.

-v Verbose output.

infile.rle The input will be read from this file. If infile.rle is "-" or is not specified, the input will be read from the standard input stream.

SEE ALSO
tiff2rle(1), utr(1), RLE(5).

AUTHOR
Bailey Brown, University of Michigan.

RLE ZOOM

NAME
rlezoom - Magnify an RLE file by pixel replication.

SYNOPSIS
rlezoom factor [ y-factor ] [ -f ] [ -o outfile ] [ infile ]

DESCRIPTION
This program magnifies (zooms) an RLE(5) file by a floating point factor. Each pixel in the original image becomes a block of pixels in the output image. If no y-factor is specified, then the image will be magnified by factor equally in both directions. If y-factor is given, then each input pixel becomes a block of factor y-factor pixels in the output. If factor or y-factor is less than 1.0, pixels will be dropped from the image. There is no pixel blending performed. Input is taken from infile, or from the standard input if not specified. The magnified image is written to the standard output, or outfile, if specified.
You should use rlezoom over fant(1) if you just want a quick magnification of an image with the pixel boundaries showing. It is significantly faster than fant because it does no arithmetic on the pixel values. If you need blending between pixels in the magnified image, then fant is the correct program to use. Use rlezoom -f factor y-factor to produce an image the same size as fant -p 0 0 -s factor y-factor for previewing purposes.
Note: due to the way that scanargs(3) parses the arguments from the command line, if the name of infile is a number, and it is in the current directory, you should prefix it with "./" so that it will not be confused with factor or y-factor.

SEE ALSO
fant(1), utr(1), scanargs(3), RLE(5).

AUTHOR
Spencer W. Thomas, Gerald A. Winters.
SMUSH

NAME

smush – defocus an RLE image.

SYNOPSIS

smush [ -m maskfile ] [ -n ] [ -o outfile ] [ levels ] [ infile ]

DESCRIPTION

Smush convolves an image with a 5x5 Gaussian mask, blurring the image. One may also provide a mask in a text file. The file must contain an integer to specify the size of the square mask, followed by size*size floats. The mask will be normalized (forced to sum to 1.0) unless the -n flag is given. The resulting image is the same size as the input image, no sub-sampling takes place. The levels option, which defaults to one, signifies the number of times which the image will be blurred. Each successive blurring is done with a more spread out mask, so a smush of level 2 is blurrier than piping two level one smush calls. If no input file is specified, smush reads from stdin. If no output file is specified with -o it writes the result to stdout.

SEE ALSO

avg4(1), urs(1), RLE(5).

AUTHOR

Rod G. Bogart

BUGS

Smush should probably automatically generate different sized gaussians and other common filters.

TARGATORLE

NAME

targatorle – Convert Targa 32 TIPS images to RLE format.

SYNOPSIS

targatorle [ -h headerfile ] [ -o outfile.rle ] [ infile.tga ]

DESCRIPTION

Targatorle converts a file from Targa 32 TIPS format into RLE format. Because TIPS and RLE agree on which end is up, rleflip is not necessary to preserve orientation. If no input file is specified, the data is read from stdin.

OPTIONS

-h    Allow the program to write Targa header information to headerfile

-o    Use outfile as output instead of stdout.

SEE ALSO

urs(1), RLE(5).
APPENDIX H. UTILITY PROGRAMS

AUTHOR
Hann-Bin Chuang

TIFFTORLE

NAME

tifftorle - Convert 24 bit TIFF image files to RLE.

SYNOPSIS

tifftorle [ -o outfile.rle ] infile.tif

DESCRIPTION

This program converts a 24 bit TIFF image to RLE(5) format.

OPTIONS

- If specified, the output will be written to this file. If outfile.rle is "-", or if it is not specified, the output will be written to the standard output stream.

infile.tif - The input will be read from this file. infile.tif must be a real file, the special cases described in urt(1) do not apply here.

SEE ALSO

tifftorle(1), urt(1), RLE(5).

AUTHOR

Bailey Brown, University of Michigan.

TO8

NAME

to8 - Convert a 24 bit RLE file to eight bits using dithering.

SYNOPSIS

to8 [ -g display.gamma ] [ -{iI} image.gamma ] [ -o outfile ] [ infile ]

DESCRIPTION

To8 Converts an image with 24 bit pixel values (eight bits each of red, green and blue) to eight bits of color using a dithered color map (the special color map is automatically written into the output file). If no input file is specified, to8 reads from stdin. If no output file is specified with -o it writes the result to the standard output.

Other options allow control over the gamma, or contrast, of the image. The dithering process assumes that the incoming image has a gamma of 1.0 (i.e., a 200 in the input represents an intensity twice that of a 100.) If this is not the case, the input values must be adjusted before dithering via the -i or -I option. The input file may also specify the gamma of the image via a picture comment (see below). The output display is assumed to have a gamma of 2.5 (standard for color TV monitors). This may be modified via the -g option if a display with a different gamma is used.

To8 will put a picture comment into the output file indicating the display gamma assumed in constructing the dithering color map.
OPTIONS

-i image.gamma Specify the gamma (contrast) of the image. A low contrast image, suited for direct display without compensation on a high contrast monitor (as most monitors are) will have a gamma of less than one. The default image gamma is 1.0. Image gamma may also be specified by a picture comment in the RLE (5) file of the form image.gamma=gamma. The command line argument will override the value in the file if specified.

-I image.gamma An alternate method of specifying the image gamma, the number following -I is the gamma of the display for which the image was originally computed (and is therefore 1.0 divided by the actual gamma of the image). Image display gamma may also be specified by a picture comment in the RLE (5) file of the form display.gamma=gamma. The command line argument will override the value in the file if specified.

-g display.gamma Specify the gamma of the X display monitor. The default value is 2.5, suitable for most color TV monitors (this is the gamma value assumed by the NTSC video standard).

-o outfile If specified, the output will be written to this file. If outfile is "-", or if it is not specified, the output will be written to the standard output stream.

SEE ALSO
tobw(1), getx11(1), mcut(1), rlequant(1), urt(1), dither(3), RLE(5).

AUTHOR
Spencer Thomas

TOBW

NAME
tobw – Convert a 24 bit RLE file to eight bits of gray scale value.

SYNOPSIS
tobw [ -t ] [ -o outfile ] [ infile ]

DESCRIPTION
Tobw converts an image with 24 bit pixel values (eight bits each of red, green and blue) to eight bits of grayscale information. The NTSC Y transform is used. If the -t flag is given, then the monochrome pixel values are replicated on all three output channels (otherwise, just one channel of eight bit data is produced). If no input file is specified, tobw reads from stdin. If no output file is specified with -o, it writes the result to stdout.

SEE ALSO
tob8(1), urt(1), rgb_to_bw(3), RLE(5).

AUTHOR
Spencer Thomas
APPENDIX H. UTILITY PROGRAMS

UNSLICE

NAME
unslice – Quickly assemble image slices

SYNOPSIS
unslice [ -f controlfile ] [ -y ymax ] [ -o outfile ] infiles ...

DESCRIPTION

Unslice quickly assembles a number of horizontal image strips into a single output image. A typical use for unslice is to put together portions of an image ("slices") computed independently into a single output picture. Because unslice uses the "raw" RLE library calls to read and write the images, it runs much faster than doing the equivalent operations with crop and comp.

Unslice has two modes of operation. If given the -f flag, unslice reads a control file telling it how to assemble the images. This is a text file with two decimal numbers on each line, one line for each slice to be assembled into the output image. Each line gives the starting and stopping scanlines (inclusive) for each slice. These must be in ascending order. This is useful if the slices have excess image area that should be cropped away.

If no control file is given, the -y flag is used. This tells unslice what the maximum Y value of the output image is. Unslice reads the files in order, using the RLE headers to determine where to place the slices. If two slices overlap, the first scanlines from the second slice are thrown away. In both cases, the slices must be in ascending order, and are expected to be of uniform width.

SEE ALSO

crop(1), rlecomp(1), rlepatch(1), repos(1), urt(1), RLE(5).

AUTHOR

John W. Peterson

BUGS

Unslice has really been superceded by rlepatch(1).

URT

NAME
urt – overview of the Utah Raster Toolkit

SYNOPSIS

applymap Apply color map to image data.
avg4 Simple 2x2 downsizing filter.
crop Crop image.
cubitorle Convert Cubicomp format to RLE.
dvirlle Typeset TeX ".dvi" files as RLE images.
fant Image scale/rotate with anti-aliasing.
get4d Display on SGI Iris/4D display.
get_orion Display on "Orion" display.
getap Display on Apollo.
getbob Display under HP window system.
getcx3d Display RLE on Chromatics CX3D.
getfb Display using BRL generic fb library.
getgmr Display on Grinnell GMR-27 frame buffer.
getiris Display on SGI 2400/3000 w/o window manager.
getmac Display on Mac under MPW.
getmex Display on SGI under the window manager.
gerqer Display on Matrix QCR camera.
geren Display on HP SRX.
getsun Display using SunTools.
getx10 Display on X10 display.
getx11 Display using X11.
giftorle Convert GIF files to RLE.
graytorle Convert separate rrr ggg bbb files to RLE.
mcut Median cut color quantization.
mergechan Merge colors from multiple images.
painttorle Convert MacPaint to RLE.
pgmtoRle Convert PBMPLUS pgm format to RLE.
ppmtorle Convert PBMPLUS ppm format to RLE.
pyrmask Generate "pyramid" filter mask.
rastorle Convert Sun Raster to RLE.
rawtorle Convert various raw formats to RLE.
read98721 Read the screen of an HP 98721 "Renaissance" to an RLE file.
repos Reposition an image.
rlelorate Convert Wavefront RLA format to RLE.
rleClock Draws a clock face.
rledocom Add comments to an RLE file.
rleadddeof Add an EOF code to an RLE file.
rlebg Generate a "background".
rlebox Find bounding box of an image.
rlecomp Image composition.
rledither Floyd-Steinberg dither an image to a given colormap.
rleflip Flip an image or rotate it 90.
llehdr Print info about an RLE file.
llehisto Make a histogram of an image.
lleldmap Load a new colormap into a file.
llemandl Make a Mandelbrot image.
lrenoise Add noise to an image.
rlepach Patch smaller images on a big one.
lleprint Print all pixel values in image.
lrelequant Variance based color quantization.
lreselect Select images from an RLE file.
lresetbg Set the background color of an image file.
lreskel Skeleton tool. Programming example.
lrespiff Simple contrast enhancement.
lresplice Splice two images horizontally or vertically.
lresplit Split concatenated images into files.
lreswap Swap or select color channels.
rletoab A60 Convert RLE to Abekas A60 format.
rletoab A62 Convert to Abekas A62 format.
rletoascii Make a line-printer/CRT version of an RLE image.
rletogif Convert RLE images to GIF format.
rletogray Convert RLE to separate rrr ggg bbb files.
rletopaint Convert RLE to MacPaint.
rletoppm Convert RLE to PBMPLUS ppm format.
**APPENDIX H. UTILITY PROGRAMS**

- **rletops** Convert RLE to (B&W) PostScript.
- **rleorast** Convert RLE to Sun Raster.
- **rletoraw** Convert RLE to rgbrgb raw format.
- **rletorla** Convert RLE to Wavefront RLA format.
- **rletotiff** Convert RLE to TIFF 24 bit format.
- **rlezoom** Scale image by sub- or super-sampling.
- **smush** Generic filtering.
- **targatorle** Convert TARGA to RLE.
- **tifftorle** Convert TIFF 24 bit images to RLE.
- **to8** 24 to 8 bit ordered dither color conversion.
- **tobw** ColorB&W conversion.
- **unexp** Convert "exp" format to normal colors.
- **unslice** Paste together "slices" into a full image.
- **wasatchrle** Convert Wasatch paint system to RLE.

**DESCRIPTION**

The *Utah* Raster Toolkit is a collection of programs and C routines for dealing with raster images commonly encountered in computer graphics. A device and system independent image format stores images and information about them. Called the RLE(5) format, it uses run length encoding to reduce storage space for most images.

The programs (tools) currently included in the toolkit are listed above, together with a short description of each one. Most of the tools read one or more input RLE files and produce an output RLE file. Some generate RLE files from other information, and some read RLE files and produce output of a different form.

An input file is almost always specified by mentioning its name on the command line. Some commands, usually those which take an indefinite number of non-file arguments (e.g., `rleaddcom`) require a `-i` flag to introduce the input file name. If the input file name is absent the tool will usually read from the standard input. An input file name of `"-"` also signals that the input should be taken from the standard input.

An output file is almost always specified using the option `-o outfile`. If the option is missing, or if `outfile` is `"-"`, then the output will be written to the standard output.

On Unix systems, the special file name forms above may also be used for output files. File names starting with `"1"` are taken as a command to which the tool output will be sent. If the file name ends in `".Z"` (and which does not begin with a `"1"`) will be decompressed by the `compress(1)` program. Both of these options supply input to the tool through a pipe. Consequently, certain programs (those that must read their input twice) cannot take advantage of these features. This is noted in the manual pages for the affected commands.

An output file is almost always specified using the option `-o outfile`. If the option is missing, or if `outfile` is `"-"`, then the output will be written to the standard output.

On Unix systems, the special file name forms above may also be used for output files. File names starting with `"1"` are taken as a command to which the tool output will be sent. If the file name ends in `".Z"`, then `compress` will be used to produce a compressed output file.

Several images may be concatenated together into a single file, and most of the tools will properly process all the images. Those that will not are noted in their respective man pages.

**Picture comments.** Images stored in RLE form may have attached comments. There are some comments that are interpreted, created or manipulated by certain of the tools. In the list below, a word enclosed in `<>` is a place-holder for a value. The `<>` do not appear in the actual comment.

```
image_gamma=<float number>
```

Images are sometimes computed with a particular "gamma" value - that is, the pixel values in the image are related to the actual intensity by a power law,

```
pixel_value=intensity^image_gamma
```

Some of the display programs, and the `buildmap(3)` function will look for this comment and automatically build a "compensation table" to transform the pixel values back to true intensity values.
display_gamma=<float number> The display_gamma is just 1/image_gamma. That is, it is the "gamma" of the display for which the image was computed. If an image_gamma comment is not present, but a display_gamma is, the displayed image will be gamma corrected as above. The to8 program produces a display_gamma comment.

colormap_length=<integer> The length of the colormap stored in the RLE header must be a power of two. However, the number of useful entries in the colormap may be smaller than this. This comment can be used to tell some of the display programs (getx11, in particular) how many of the colormap entries are used. The assumption is that entries 0 - colormap_length-1 are used. This comment is produced by mcut, rlequant, and rledither.

image_title=<string> This comment is used by getx11 to set the window title. If present, the comment is used instead of the file name. (No other programs currently pay attention to this comment.) The comments IMAGE_TITLE, title, and TITLE are also recognized, in that order. No programs produce this comment.

HISTORY=<string> All toolkit programs (with the exception of readdcom) create or add to a HISTORY comment. Each tool appends a line to this comment that contains its command line arguments and the time it was run. Thus, the image contains a history of all the things that were done to it. No programs interpret this comment.

exponential_data This comment should be present in a file stored in "exponential" form. See unexp(1) and float_to_exp(3) for more information. The unexp program expects to see this comment.

SEE ALSO
compress(1), sh(1), RLE(5).

AUTHOR
Many people contributed to the Utah Raster Toolkit. This manual page was written by Spencer W. Thomas, University of Michigan.

RLE NAME
rle - Run length encoded file format produced by the rle library

DESCRIPTION
The output file format is (note: all words are 16 bits, and in PDP-11 byte order):

Word 0 A "magic" number 0xc52. (Byte order 0x52, 0xc.)

Words 1-4 The structure (chars saved in PDP-11 order)

```c
{
    short xpos, /* Lower left corner
```
/* Size of saved box */

Byte 10 (flags) The following flags are defined:

0.5i

H.CLEARFIRST (0x1) If set, clear the frame buffer to background color before restoring.

H.NO.BACKGROUND (0x2) If set, no background color is supplied. If H.CLEARFIRST is also set, it should be ignored (or alternatively, a clear-to-black operation could be performed).

H.ALPHA (0x4) If set, an alpha channel is saved as color channel -1. The alpha channel does not contribute to the count of colors in ncolors.

H.COMMENT (0x8) If set, comments will follow the color map in the header.

Byte 11 (ncolors) Number of color channels present. 0 means load only the color map (if present), 1 means a B&W image, 3 means a normal color image.

Byte 12 (pixedbits) Number of bits per pixel, per color channel. Values greater than 8 currently will not work.

Byte 13 (ncmap) Number of color map channels present. Need not be identical to ncolors. If this is non-zero, the color map follows immediately after the background colors.

Byte 14 (cmaplen) Log base 2 of the number of entries in the color map for each color channel. I.e., would be 8 for a color map with 256 entries.

Bytes 15—... The background color. There are ncolors bytes of background color. If ncolors is even, an extra padding byte is inserted to end on a 16 bit boundary. The background color is only present if H.NO.BACKGROUND is not set in flags. IF H.NO BACKGROUND is set, there is a single filler byte. Background color is ignored, but present, if H.CLEARFIRST is not set in flags.

If ncmap is non-zero, then the color map will follow as ncmap*2^cmaplen 16 bit words. The color map data is left justified in each word.

If the H.COMMENT flag is set, a set of comments will follow. The first 16 bit word gives the length of the comments in bytes. If this is odd, a filler byte will be appended to the comments. The comments are interpreted as a sequence of null terminated strings which should be, by convention, of the form name=value, or just name.

Following the setup information is the Run Length Encoded image. Each instruction consists of an opcode, a datum and possibly one or more following words (all words are 16 bits). The opcode is encoded in the first byte of the instruction word. Instructions come in either a short or long form. In the short form, the datum is in the second byte of the instruction word; in the long form, the datum is a 16 bit value in the word following the instruction word. Long form instructions are distinguished by having the 0x40 bit set in the opcode byte. The instruction opcodes are:

SkipLines (1) The datum is an unsigned number to be added to the current Y position.

SetColor (2) The datum indicates which color is to be loaded with the data described by the following ByteData and RunData instructions. Typically, 0red, 1green, 2blue. The operation also resets the X position to the initial X (i.e. a carriage return operation is performed).
**SkipPixels** (3) The datum is an unsigned number to be added to the current X position.

**ByteData** (5) The datum is one less than the number of bytes of color data following. If the number of bytes is odd, a filler byte will be appended to the end of the byte string to make an integral number of 16-bit words. The X position is incremented to follow the last byte of data.

**RunData** (6) The datum is one less than the run length. The following word contains (in its lower 8 bits) the color of the run. The X position is incremented to follow the last byte in the run.

**EOF** (7) This opcode indicates the logical end of image data. A physical end-of-file will also serve as well. The **EOF** opcode may be used to concatenate several images in a single file.

SEE ALSO

`librle(3)`

AUTHOR

Spencer W. Thomas, Todd Fuqua
Index

* (keyword) ...................... 56, 58
.Z (file) ....................... 118, 162, 348
.a (file) ....................... 94, 120
.bin (file) .................... 232, 233, 235
.bps (file) .. 94, 117, 118, 120, 179, 230, 233, 347
cshrc (file) ..................... 250
.env (file) 28, 101, 113, 117, 177, 178, 232, 233, 235
.fig (file) 117, 178, 225
.jack.menu (file) .................. 19
.jack5.install (file) ............. 119, 121
.jcl (file) ...................... 117
.post (file) ..................... 225
.pss (file) 94, 117, 120, 179, 228, 347
.rgb (file) ..................... 114
.rle (file) 187, 231-233, 236, 240, 348
/tmp/flockd.log (file) ............. 210
#include (file) ................... 228
$JACK/bin (file) .................. 209
$JACK/demojack/ (file) ............ 19
$JACK/jacklib/human (file) ........ 94
$JACK/jacklib5/demo (file) ........ 101
$JACK/jacklib5/demo/animation (file) 101
$JACK/jacklib5/demo/radiosity (file) 246
$JACK/jacklib5/demo/rtrace (file) .. 231, 236 235 212
$JACK/jacklib5/jack.imagefilters.dat (file) ............. 164
$JACK/jacklib5/jack.sample.colors (file) ............. 152
$JACK/jacklib5/jack.standard.colors (file) ............. 152
$JACK/jacklib5/sass/ (file) ....... 250
$JACK/jacklib5/textures/ (file) . 162, 164
$JACK/src/jacket (file) ........... 203
"/./4Dmrc (file) .................. 20
"/./jack.jcl (file) ............... 193, 194
"/./jack5.install (file) 94, 119, 122
4Dwm (file) ..................... 20

add one obstacle (command) .... 210, 211
add path point (command) ........ 205, 311
adjust clipping planes (command) . 48, 311
adjust joint (command) 27, 47, 61, 62, 86, 194, 311
aim (keyword) .................... 136
ambient (keyword) ................ 153, 219
animation (menu) .................. 102
anthropometry (menu) ............. 25, 65, 89
apply deformation (command) .... 172, 311
applymap (program) ............... 350
apropos (command) ............... 311
ar (program) ..................... 120, 94
archive (keyword) ............... 120, 219
asurf (keyword) ................. 219
attach figure (command) .. 18, 74, 76, 83, 110, 139, 141, 311
attach view to site (command) . 25, 43, 47, 48, 312
attached figure (command) ........ 84
attribute (keyword) ............. 219, 220, 229
attribute (menu) ................. 153, 154, 158
attribute texture off (command) .... 161
attribute texture on (command) .... 161
attributes (keyword) ............. 219
auto map texture to figure (command) 161
auto map texture to segments (command) . 161
auto zmap texture to figure (command) 161
auto zmap texture to segment (command) . 163
auto zmap texture to segments (command) 161
avg4 (program) ................. 350
avoid collision (command) ....... 210, 211
background (menu) .............. 190, 191
behavior (menu) .................. 65
behavior info (command) ........ 38, 312
bend from waist (command) ...... 67, 68
bend torso (command) 66-70, 86, 312
bevel edge (command) ............ 312
bevel node (command) ............ 312
bind command to key (command) .. 194, 312
blur (keyword) .................. 219 86
bps (program) ................. 118, 347
INDEX

browse (command) ................. 30, 312
build (menu) .................... 172
cad (menu) ....................... 175, 176
center view (command) ............ 312
chain.fig (file) ............ 125, 139, 142, 153
chair.pss (file) ............ 23, 32, 51, 74, 172
change motion (menu) ............ 112
change cad object (command) ... 176, 313
change collision avoidance parameters (command) ............. 210, 211
change constraint (command) ... 124, 126, 132, 133, 315
change directory (command) .... 196, 313
change motion (command) ........ 112, 313
change motion (menu) ............ 112
change motor (command) .......... 148, 313
change view (command) 25, 34-36, 43, 45-47, 49, 55, 188, 194, 246, 313
channel (menu) .................. 114
check bbox collision (command) ... 86
check full collision (command) ... 86
clear frames (command) ........... 113, 313
clear trace (command) ............ 313
clipping planes (menu) .......... 25, 43
close command port (command) ... 204
close hand (command) ............. 85, 313
close jack port (command) ........ 208
cm (keyword) ..................... 196, 217
collapse all motion groups (command) 112, 314
collapse motion group (command) 112, 314
collision (menu) .................. 199
color (menu) ...................... 195
command (menu) ................ 30
compress (program) .............. 118
concave off (command) .......... 196, 314
concave on (command) .......... 196, 314
concentration (keyword) .......... 219
connect (keyword) ............... 64, 219, 221
constraint (keyword) ............ 219
constraint (menu) ............... 18, 123-126
constraint info (command) ........ 38, 314
contour body (menu) ............. 65, 95
contour.fig (file) ............... 95
contour.stick.fig (file) ........ 95
contrib (menu) .................. 22
convert (program) ............... 282
CQ check collisions (command) ... 201
CQ check environment (command) ... 199, 200, 310
CQ check figure (command) 199, 200, 310
CQ check figure pair (command) ... 199, 200, 310
CQ check segment (command) 199, 200, 310
CQ check segment pair (command) ... 199, 200, 310
CQ clear current figure (command) ... 199, 200, 310
CQ clear figure (command) ... 199, 200, 310
CQ clear figure pair (command) ... 199, 200, 310
CQ clear intra figure (command) ... 199, 200, 310
CQ clear queue (command) ... 199, 200, 310
CQ clear segment (command) ... 199, 200, 310
CQ clear segment pair (command) ... 199, 200, 310
CQ flash colliding segments (command) ........... 199, 201, 311
CQ flash queued segments (command) ... 199, 201, 311
CQ report collisions (command) ... 199, 201, 311
CQ update collisions (command) ... 199, 201, 311
create (menu) ................. 22, 23, 32, 117
create animation window (command) ... 314
create arm motion (command) 84, 85, 102, 106, 108, 314
create cad object (command) ... 176, 313
create camera motion (command) ... 109, 110, 314
create center of mass motion (command) ... 102, 106, 107, 314
create constraint (command) ... 124-126, 132, 315
create contour figure (command) ... 95, 315
create eye motion (command) ... 102, 106, 316
create eye window (command) ......... 95
create face (command) ............ 170, 316
create figure joint motion (command) ... 109, 316
create figure motion (command) ... 109, 110, 316
create figure path motion (command) ... 109, 110, 316
create finger motion (command) ... 102, 106, 109, 316
create foot motion (command) ... 102, 106-108, 317
create fractal (command) ........... 206, 317
create ghost (command) ........... 317
create ghost figure (command) ........... 202
create hand motion (command) ........... 66
create hand site (command) ........................................ 65, 85, 317
create heel motion (command) .................................... 102, 106, 107, 318
create human figure (command) .................................. 23, 38, 65, 66, 89, 94, 318
create jcl motion (command) ...................................... 109, 111, 318
create joint between segments (command) ...................... 141, 145–147, 205, 318
create joint between sites (command) ............................ 141, 145, 318
create joint chain motion (command) ............................ 109, 319
create joint limit spring (command) .............................. 124, 126
create joint motion (command) ................................... 109, 110, 319
create light (command) ............................................ 23, 117, 122, 319
create light motion (command) ................................... 109, 319
create light source (command) .................................... 159
create motion group (command) ................................... 109, 111, 319
create new psurf (command) ....................................... 170
create node (command) ............................................. 170, 319
create ordinary window (command) ............................... 47, 182, 187, 188, 320
create path (command) ............................................. 205, 320
create path motion (command) ..................................... 109, 110, 205, 320
create pelvis motion (command) .................................. 102, 106, 107, 320
create preview movie (command) .................................. 114, 320
create retina window (command) .................................. 212
create rooting constraint (command) ............................. 124, 126, 139, 320
create ruler (command) ............................................. 92, 321
create ruler between sites (command) ............................ 92, 321
create segment deformation (command) ......................... 172, 321
create shaded window (command) ................................ 182, 187, 320
create site (command) .............................................. 23, 117, 122, 321
create stick figure (command) .................................... 95, 321
create strength window (command) ................................ 96, 97
create time hand control (command) ............................. 106
create time head control (command) .............................. 106
create timed attachment (command) ...................... 108–110, 311
create timed balance control (command) ......................... 106, 334
create timed constraint (command) .............................. 109, 111, 321
create timed foot control (command) ............................ 108, 336
create timed hand control (command) ....................... 108, 336
create timed head control (command) .......................... 337
create timed pelvis control (command) .......................... 338
create timed torso control (command) ........................... 106, 107, 339
create torso motion (command) .................................. 102, 106–108, 322
create view cones (command) ...................................... 96, 322
create wireframe window (command) ............................. 182, 187, 320
create x window (command) ...................................... 323
create y window (command) ...................................... 323
create z window (command) ...................................... 323
damp (keyword) ..................................................... 219
definition (menu) ................................................... 172
deg (keyword) ....................................................... 196, 217
delete all constraints (command) .................................. 124, 126, 132, 323
delete all motors (command) ....................................... 148, 323
delete all non-rooting constraints (command) .................. 132
delete constraint (command) ....................................... 124, 126, 132, 133, 323
delete edge (command) ............................................ 172, 323
delete environment (command) ..................................... 141, 145, 323
delete face (command) ............................................. 172, 323
delete figure (command) .......................................... 49, 141, 145, 323
delete ghost (command) ............................................ 317, 323
delete ghost figure (command) ..................................... 202
delete intersecting faces (command) .............................. 323
delete joint (command) ............................................. 141, 147, 323
delete motion (command) .......................................... 112, 323
delete motion group (command) ................................... 112, 323
delete motions (command) .......................................... 109, 323
delete motor (command) ............................................ 148, 323
delete node (command) ............................................. 172, 324
delete nonrooting constraints (command) ......................... 124, 126, 324
delete path (command) ............................................. 205, 324
delete path point (command) ....................................... 205, 324
delete ruler (command) ............................................. 92, 324
delete segment (command) ......................................... 95, 141, 147, 324
delete segment deformation (command) ......................... 172, 175, 324
delete trace (command) ............................................. 202, 313
delete vertex (command) ........................................... 172, 324
delete view cones (command) ...................................... 96, 324
make texture plane (command) .. 161, 163
make viewer infinite (command) .. 182, 197, 328
make viewer local (command) .. 182, 197, 328
make window ordinary (command) .. 182, 328
make window shaded (command) .. 182, 185, 328
make window wireframe (command) .. 182, 185, 328
manipulation (menu) .................. 65
map texture to faces (command) . 161-163
mass (keyword) ......................... 219
maxtrans (keyword) ..................... 219
mcut (program) ......................... 356
measure anterior arm (command) .. 329
measure anterior arm reach (command) .. 93
measure arm span (command) .. 93, 329
measure biceps (command) .. 93, 329
measure biceps circumference (command) .. 93
measure buttoc knee (command) .. 329
measure buttoc-popliteal (command) .. 93
measure buttoc-knee length (command) .. 93
measure buttoc-popliteal length (command) .. 93
measure calf (command) .................. 329
measure calf circumference (command) .. 93
measure chest (command) .................. 329
measure chest circumference (command) .. 93
measure chest depth (command) .. 93, 329
measure crotch (command) .... 329
measure crotch height (command) .... 92
measure distance (command) .. 18, 38, 90, 91, 329
measure eye height (command) .... 92
measure eye sitting (command) .... 329
measure eye standing (command) .... 329
measure foot breadth (command) . 92, 329
measure foot length (command) .. 93, 329
measure forearm circumference (command) .. 93
measure forearm hand (command) .. 329
measure forearm-hand length (command) 93
measure functional (command) .... 329
measure functional reach (command) .. 93
measure hand breadth (command) . 92, 329
measure hand circumference (command) .. 93
measure hand length (command) .. 93, 329
measure head circumference (command) .. 93
measure head length (command) .. 93, 329
measure hip (command) ................. 329
measure hip breadth sitting (command) .. 93
measure hip breadth standing (command) .. 92
measure hip circumference (command) .. 93
measure hip sitting (command) ........... 329
measure hip standing (command) ........... 329
measure interpupillary breadth (command) .. 92
measure interpupillar circumference (command) .. 92
measure knee cap (command) ........... 329
measure knee cap height sitting (command) .. 92
measure knee cap height standing (command) 92
measure neck (command) ............... 329
measure neck circumference (command) 93
measure overhead (command) ............. 329
measure overhead reach (command) .... 93
measure palm length (command) .. 93, 329
measure popliteal (command) ........... 329
measure popliteal height (command) .... 92
measure shoulder breadth (command) .. 92, 329
measure shoulder circumference (command) .. 93
measure shoulder elbow (command) .. 329
measure shoulder height (command) .... 92
measure shoulder length (command) .. 93, 329
measure shoulder sitting (command) .. 329
measure shoulder standing (command) .. 329
measure shoulder-elbow length (command) .. 93
measure sitting eye height (command) .. 92
measure sitting height (command) ........ 92
measure standing height (command) ........ 92
measure stature sitting (command) .. 329
measure stature standing (command) .. 329
measure thigh circumference (command) .. 93
measure thigh clearance (command) .. 329
measure thigh clearance height (command) .. 92
measure vertical trunk (command) .. 329
measure vertical trunk circumference (command) .................. 93
measure waist (command) ............... 329
measure waist circumference (command) 93
measure wrist circumference (command) 93
measurements (menu) .................... 65, 92
merge segments (command) 141, 147, 329
mergechan (program) .................... 357
INDEX

set render parameters (command) 113, 115, 339
set rotation type (command) 197, 339
set ruler parameters (command) 92, 339
set scene ambient (command) 197
set scene aux scale (command) 196, 339
set scene scale (command) 196, 339
set segment density (command) 148, 149, 339
set segment mass (command) 148, 149, 339
set site color (command) 195
set star color (command) 195
set status background color (command) 195
set status highlight color (command) 195
set status text color (command) 195
set stepping behavior (command) 79, 339
set torso control (command) 69-71, 107, 108, 339
set torso dimensions (command) 90, 336
set trace color (command) 202
set upper arm dimensions (command) 90, 336
set upper leg dimensions (command) 89, 336
set vertical view gain (command) 25, 43, 337
set view glide (command) 196, 340
set window ambient (command) 182
set window attenuation (command) 182
set window location (command) 182, 190, 340
set window size ntsc (command) 113, 115
set zoom view gain (command) 25, 43, 337
shade windows (command) 182, 187, 194, 340
shademode (keyword) 219
shape hand (command) 85, 340
shell (command) 196, 340
show installed files (command) 38, 122, 340
show Jack version (command) 38, 340
show memory usage (command) 38, 340
show object files (command) 38, 122, 340
show peabody symbols (command) 38, 340
show player list (command) 208
site (keyword) 126, 127, 219, 220
site info (command) 38, 40, 341
smallcube.pss (file) 125, 129, 133, 139
smush (program) 389
snap (menu) 46, 91
snap view (command) 25, 43, 341
snug clipping planes (command) 48
specular (keyword) 219
sphere (keyword) 219
sphere.pss (file) 174
split edge (command) 169, 341
split face (command) 169, 341
split segment (command) 141, 147
start time (command) 111, 112, 341
step through frames (command) 113, 341
step time (command) 341
stexture (keyword) 219
stiff (keyword) 219
stiffness (keyword) 219
stop browsing (command) 30, 312
stop playback (command) 113, 341
stop time (command) 111, 341
strength (menu) 63
switch figure (command) 95, 341
T (keyword) 61, 146, 219, 221
targatole (program) 389
texture (keyword) 219
texture (menu) 161
texture subdivide off (command) 161
texture subdivide on (command) 161, 162
texture subdivision (command) 163
thaw joint (command) 141, 145, 326
thaw window (command) 182, 190, 212, 326
tifftorle (program) 390
time (menu) 101
to (keyword) 219, 221
tob (program) 390
tobw (program) 391
toggle psurf bounding boxes (command) 186, 341
toggle psurf nodes only (command) 186, 341
tolerance (keyword) 219
tools (file) 83
toroid (command) 176
torus.pss (file) 155, 156
trace (menu) 202
trace segment (command) 202, 341
trace site (command) 202, 342
tran (keyword) 56, 217, 219
transexp (keyword) 219
transparency (keyword) 219
triangulate face (command) 169
tumble view (command) 25, 43
turn all (one) obstacles on/off (command) 211
turn all motors off (command) 148, 343
turn all motors on (command) 148, 343
turn all obstacles on/off (command) 210
turn background off (command) 191, 246, 342
write environment (command) . 28, 88, 89, 94, 133, 143, 144, 156, 157, 169, 177, 178, 344
write environment files to vdisk (command) 203
write environment to psurf (command) . 28, 177, 281, 344
write environment with postures (command) 87, 88
write figure definition (command) . 28, 94, 177, 178, 344
write figure position (command) . 28, 177, 345
write figure posture (command) .... 178
write figure psurfs (command) 28, 177, 345
write fractal as psurf (command) 207, 317, 345
write frames (command) .... 113, 114, 345
write frames global positions (command) .... 28, 177
write frames to env files (command) . 113, 345
write global positions (command) . 28, 177
write global psurf (command) .... 118, 179
write image to vdisk (command) .... 202
write JCL log (command) .... 28, 177, 193, 194, 344
write motion group definition (command) . 109, 111
write motions (command) . 109, 113, 345
write positions (command) . 28, 177, 345
write psurf (command) . 28, 118, 177, 179
write psurf globally (command) .... 28, 177
write psurf[globally] (command) .... 345
write to vdisk (command) .... 182, 345
write window image (command) . 164, 182, 187, 345
write window to vdisk (command) .... 203

xv (program) ..................... 164
xyz (keyword) ........... 56, 197, 217-219

yd (keyword) ............... 196, 217
yes (keyword) ....................... 134