Programming with *Jack* (Fourth Edition)

MS-CIS-91-19
GRAPHICS LAB 39

Cary B. Phillips

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

February 1991

Programming with Jack

Fourth Edition
Second Revision
Jack Version 5.6

January 26, 1993

Author: Cary B. Phillips
Revisions: John P. Granieri

Computer Graphics Research Laboratory
Department of Computer and Information Sciences
University of Pennsylvania
Philadelphia, Pennsylvania 19104-6389

The development of this software is partially supported by Lockheed Engineering and Management Services (NASA Johnson Space Center), NASA Ames Grant NAG-2-426, NASA Goddard through University of Iowa UICR, FMC Corporation, Siemens Research, NSF CISE Grant CDA88-22719, Air Force HRL/LR ILIR-40-02 and F33615-88-C-0004 (SEI), and ARO Grant DAAL03-89-C-0031 including participation by the U.S. Army Human Engineering Laboratory, Natick Laboratory, and TACOM.

Contents

Chapter 1

Programming with Jack

This manual describes the implementation of *Jack*™, with emphasis on how to extend it and modify it. The principle purpose of this manual is to describe what functions in the *Jack* libraries are available to be used in writing new features for *Jack*. The manual also gives an overview of how *Jack* works, for those interested in modifying its current behavior. This manual assumes that you already know how to use *Jack*, and are familiar with its basic terminology.

The foundation of *Jack* is ***peabody,*** which is a representation for articulated geometric objects. It represents ***figures*** composed of ***segments*** connected by ***joints***, also under the influence of ***constraints***. *Jack* is a facility for modeling, displaying, manipulating, and animating objects represented by peabody. It provides a standard user and programmer interface to routines that operate on the environment. This manual describes the low level routines that access and control the peabody environment, as well as the higher-level routines that allow you to define new commands in *Jack*.

The major bulk of *Jack* source code is written in C. It is a total of about 100,000 lines long. The source code is divided into several libraries, giving the source code a logical order. This manual is meant to be more of a guide to the source code than a complete description of how to use it without needing to look at it. As of version 5.6, the source code is compiled through the C++ compiler. All new features of *Jack* will be written in C++, but you are free to mix C and C++.

Jack as a program is fairly simple: it does not rely on any complex systems programming or networking concepts. As you work with it, you will make your changes and extensions in "your own version" of *Jack*. This means you will produce a program that looks like the "real *Jack*" when you run it, but it will have your extensions and modifications linked in as well. When what you do becomes stable and robust, it can be included into the officially installed version of the libraries.

1.1 The *Jack* Source Code

Jack is organized into several major libraries. The source code files are prefixed with the name of the library the belong to.

jmenu The *Jack* menus. These are the functions which set up the default menus in *Jack*.

jcmds This is the source code *Jack* commands, which loosely correspond to the commands available in the menus.

jdev The *Jack* device library. This contains the utilities and commands for accessing devices such as the network, the Flock of Birds, audio, video recorders, etc.

jack The *Jack* system, which provides the basic *Jack* windowing and command structure. This includes the routines for drawing, picking, inputting, moving, etc.

human The routines in *Jack* for dealing specifically with human figures.

peabody The peabody library, which contains the routines for reading, accessing, and maintaining peabody figures.

psurf The psurf library, which contains the routines for reading and maintaining psurfs, the geometric primitive.

alt The attribute, light, and texture library, which has routines for representing surface attributes.

vec The library of low level miscellaneous routines, including ones for basic vector and matrix operations. Whatever doesn't have a place elsewhere ends up here.

The source code is organized under the directory **gen/src/lib**. In the Graphics Lab at Penn, the most recent version of the source code is maintained in **/pkg/jack/gen/src/lib/**. The include files are in **gen/include**. Other versions of the source code may from time to time be kept in other places, but each version will have the same basic organization. For example, the lastcouple versions of *Jack* source code are stored in **/pkg/jack/VERSION**, where **VERSION** is **5.6** or **5.5**.

The libraries in *Jack* are layered, in the sense that each library in the list above is depends only on the libraries below it. For example, the peabody library depends on the psurf, alt, and vec library, but no code in the peabody library depends upon the jack, jcmds, or jmenu libraries. This organization is important to follow in adding new routines to the source code. Routines should *as far down* in the library hierarchy as possible. This ensures that routines will be as general as possible without causing circular relationships between the libraries.

1.2 Your Own Version of *Jack*

To make your own version of *Jack*, you need two files: a **Makefile** and the file **menu.c**. These can be found in the directory **gen/src/jack**. The **Makefile** is shown in Figure 1.1, with one important change: the name of the executable has been changed to **myjack**. The **Makefile** lists only one object file: **menu.o**. As you write new routines, you should put them in other files, and place a reference to each file on the **OBJ** line in the **Makefile**. This will cause them to be linked in. If you need to modify some internal part of *Jack*, locate the file that needs the change, copy it into your directory, along with the **Makefile** and **menu.c**, and put a reference to it in the **Makefile**. The linker will use your version instead of the one in the library.

```
EXE = myjack

OBJ = menu.o

SRC = $(OBJ:.o=.c++) Makefile
LIBS = -L$(LIBDIR) -ljmenu -ljcmds -lhuman -ljdev -ljack -lpea -lpsurf \
    -lalt -lvec -lrle -lgl_s -lfm_s -lsun -lbsd -lmalloc -lm -lc_s

all:  $(EXE)

$(EXE) : $(OBJ) Makefile
    $(C++) -o $(EXE) $(LDFLAGS) $(C++FLAGS) $(OBJ) $(LIBS)

include $(INCLUDEDIR)/make.h
```

Figure 1.1: **Makefile**

The file **menu.c** is shown in Figure 1.2. This file contains the function **InitMenus**, which the *Jack* main procedure invokes to initialize the menus. This is your *hook* into the *Jack* command structure. The details of how to define other commands are described in Section 2.3.

```
#include "jack.h"
#include <signal.h>

extern int CMDquit();

MENU *
InitMenus()
{
    MENU *menu,*m;

    menu = MkPopupMenu("main");

    m = InitViewMenu();
    MkSubmenuCmd(menu,m);

    m = InitCreateMenu();
    MkSubmenuCmd(menu,m);

    m = InitWriteMenu();
    MkSubmenuCmd(menu,m);

    m = InitEditMenu();
    MkSubmenuCmd(menu,m);

    m = InitInfoMenu();
    MkSubmenuCmd(menu,m);

    m = InitOptionsMenu();
    MkSubmenuCmd(menu,m);

    m = InitUtilityMenu();
    MkSubmenuCmd(menu,m);

    m = InitHumanMenu();
    MkSubmenuCmd(menu,m);

    MkMenuCmd(menu,"quit",CMDquit,SIGQUIT);

    return(menu);
}
```

Figure 1.2: `menu.c`

1.3 Perusing the Source Code

This manual describes some, but not all, of the workings of the *Jack* source code. Writing such a manual is like shooting at a moving target. By its very nature, this manual is not as up-to-date as the software itself. You should consult the source code directly if you have further questions about the workings described here. In particular, this manual frequently describes the data structures very briefly, and there are likely to be extra fields given in the actual include files.

For users of gnu emacs, there is help in perusing the *Jack* source code, in the form of "tags." Tags are wonderful things that allow you go directly to the location in the source code file where a particular routine is located. You can position the cursor over a call to a subroutine and execute the emacs command **find-tag-other-window**. Emacs will automatically find the file that contains the definition of the function, read it in, and place the cursor at the beginning of the function.

The file **/pkg/jack/gen/src/lib/TAGS** is an emacs tag table. To use tags, put the following lines in your **~/.emacs** file:

```
(setq tags-file-name "/pkg/jack/gen/src/lib/TAGS")
(global-set-key "\C-Xt" 'find-tag-other-window)
```

This binds the command **find-tag-other-window** to **^Xt**.

Chapter 2

How to Program With Jack

This chapter describes the higher level routines which are available to programmers in writing extensions to *Jack*. These extensions are usually in the form of new *Jack* commands, and this chapter describes how to go about writing these commands and making them executable inside a version of *Jack*. Many of these routines are user-interface routines, or the routines which get input from the user, by entering values from the keyboard or by picking things with the mouse.

2.1 The `main` Procedure

The **`main`** procedure in *Jack* is in the library **`-ljack`**, so you don't need one yourself. It performs a lot of bookkeepping operations, and it then invokes the main *Jack* control loop which polls the user for commands. The commands are your "hooks" into *Jack*. The operation of the main procedure is described in Section 3.1.1.

2.2 Your Own Version of Jack

The information you need in order to link your own version of *Jack* is described in Section 1.2. You need two files: a **`Makefile`**, shown in Figure 1.1 and a copy of the file **`menu.c++`** shown in Figure ??.

2.3 Defining Menus and Commands

Commands are the means through which *Jack* translates "user" action into operations on internal data structures. In this context, the "user" may not be a real person sitting at the console moving the mouse and selecting things from the pop-up menus. The input may be coming from the keyboard or from a file, or even from some external I/O source.

Commands are grouped together into menus. The menus are then grouped together into one "main" menu. The **`main`** procedure calls the function **`InitMenus`** to initialize the main menu. This function is the responsibility of your version of *Jack*, although the file **`menu.c++`** in the *Jack* program is also a good place to start. These functions are illustrated in the function **`InitMenus`** in Figure ??. Each file in the directory **`gen/src/lib/jmenu_*`** defines a function which initializes one of the menus in *Jack*, so you can also look there for examples of how to use these functions.

```
MENU *
MkMenu(name)
char    *name;
```

```
MkMenuCmd(menu,name,func)
MENU   *menu;
char   *name;
int    (*func)();


MENU *
MkSubMenu(menu,submenu)
MENU     *menu;
MENU     *submenu;


MkCmd(name,func)
char   *name;
int    (*func)();
```

Menus are created with the function **MkMenu**, which takes a character string argument giving the name of the menu. When the menu appears as a submenu, the parent menu item has the name of the submenu with the word "menu" appended. Therefore, the *name* argument should be a single word.

Commands are created with **MkMenuCmd**, which creates a command and places it in a menu. The items in a menu are collected top to bottom as they are created, so the ordering of the items in the menu depends upon the sequence in which the commands were created. There is no way of altering this order once it has been established.

MkMenuCmd takes a character string *name* which is the name of the command. This is the name which will appear in the menu. This name should generally consists of several English words separated by blank spaces. The name should consist of only alphanumeric charaters, *not* special characters such as parentheses or punctuation marks. The JCL name of the command is inferred from this name by replacing the spaces with underscores.

The *func* argument to **MkMenuCmd** is the command's function, which is the function which is called when the command is executed. By convention, these functions have names beginning with **CMD**, to distinguish them from other functions.

2.4 CMD Functions and How to Write Commands

By convention, the functions which are associated with *Jack* commands have names beginning with **CMD**. These functions are invoked by the command executioner in a special way, with the *arguments* supplied to the function. These arguments specify what "objects" the command will act upon. The argument list may contain character strings or numbers, or it may contain special values instructing the command to pick the arguments interactively.

The **CMD** functions all take two arguments. The first argument is a list of **VALUE** structures. The second argument is a pointer to a character string for the "output arguments," which is a record of all arguments used in the execution of the command. This enables *Jack* to generate JCL scripts of commands and their arguments.

It is primarily the responsiblity of the **CMD** functions to decipher the argument list to determine what objects and values are to be operated upon, and then pass the objects on to other routines which perform the "meat" of the operation. As a general rule of thumb, the **CMD** functions should be relatively short, with the bulk of the routine dedicated to accessing the argument list.

This rule of thumb is related to the general principle of modular programming. When developing an application, it is best to organize the functionality of the program into small routines which do bits and pieces of the work. The **CMD** function should then call these lower level functions after deciding what objects and values the other functions need.

By convention, the **CMD** functions rely upon the **Input** functions to get input from the argument list. There are **Input** functions for "inputting" practically everything: numbers, strings, peabody constructs, psurf items, etc. These functions encapsulate the interactive nature of *Jack*, and then decipher the argument list for the **CMD**

functions. Generally, the CMD functions simply pass the argument list and output arguments directly to the Input functions, which are responsibly for deciding what to do with the arguments.

The return value of the CMD functions specify whether the command was successfully executed. If the CMD function returns 1, then the command is recorded in the JCL list of executed commands. If the CMD function returns 0, then the command is ignored.

2.5 Input Functions

The functions in Jack which get input from the user all begin with the prefix Input. These functions operate on the arguments which are passed to the CMD function. You do not need to understand *how* these functions work, but only what they do.

The following example illustrates the typical way in which the Input functions are called.

```
CMDAnExampleCommand(args,outargs)
VALUE   *args;
char    **outargs;
{
        int     n;
        Segment *segment;

        if (!InputInt(&args,outargs,&n,"enter an integer:  ")) {
            return(0);
        }

        if (!(segment=InputSegment(&args,outargs,"Pick a segment")) {
            return(0);
        }

        :

        return(1);
}
```

A return value of 0 from an Input function specifies that the intended value was *not* input successfully. The CMD function may deal with this situation as it sees fit. Typically, this means aborting the command.

The files in the directory **gen/src/lib/jcmds_*** contain definitions for all the CMD functions in *Jack*, so you can refer to them for more examples of how to use the Input functions.

2.5.1 Inputting Strings and Filenames

There are several Input functions for entering strings and filenames. Filenames are really just strings, except that the command does filename completion and it ensures that the users doesn't enter a filename which will overwrite an existing file unless that's what he or she wants to do.

```
char *
InputString(arglist,outargs,string,prompt)
VALUE   **arglist;
char    **outargs;
char    *string;
char    *prompt;


char *
InputStringComplete(arglist,outargs,string,prompt, ncomp,compstrs,previnput,menu)
VALUE   **arglist;
char    **outargs;
char    *string;
char    *prompt;


char *
InputInputFile(arglist,outargs,filename,prompt)
VALUE   **arglist;
char    **outargs;
char    *filename;
char    *prompt;


char *
InputOutputFile(arglist,outargs,filename,prompt)
VALUE   **arglist;
char    **outargs;
char    *filename;
char    *prompt;
```

Each of these functions uses the value passed in the parameter *string* or *filename*, as the default value in the edit buffer, so it is essential that you initialize the string to a reasonable value. You can fill it in with a default string or just set it to the null string by placing a null character in the first position. Each of the functions returns a pointer to the string it enters, or `NULL` if none was entered.

`InputString` gets a character string and places it in *string*. `InputStringCompete` does the same thing but it does automatic completion, based on the values specified by *ncomp* and *compstrs*. *compstrs* is an array of character strings. Its length is given by *ncomp*. The user *may* enter any string he or she wishes, although the automatic completion will be done only on these values. These values will also be placed in the pop-up menu which the user gets by pressing the right mouse button. The *previnput* argument is a list of previously input strings, which the user may retrieve by hitting `^P` and `^N`. The *menu* argument should be `NULL`.

The functions `InputInputFile` and `InputOutputFile` do automatic completion of file names. They should be used to get the name of files which your application intends to read or write. Also, `InputOutputFile` tests to see whether the named file exists. If it does, it prompts the user to enter another name or to acknowledge that the file will be overwritten. If the user chooses to overwrite the file, the command renames the file by appending a tilde character, `~`, to the beginning of the filename, a la emacs.

These commands just return the names of files. They don't actually open them.

2.5.2 Inputting Values

The functions `InputFloat`, `InputInt`, and `InputVector` input floating point numbers, integers, and vectors, respectively, as you might guess from their names.

```
Boolean
InputFloat(arglist,outargs,f,prompt)
VALUE     **arglist;
char      **outargs;
float     *f;
char      *prompt;


Boolean
InputInt(arglist,outargs,n,prompt)
VALUE     **arglist;
char      **outargs;
int       *n;
char      *prompt;


Boolean
InputVector(arglist,outargs,vec,len,prompt)
VALUE     **arglist;
char      **outargs;
float     vec[];
int       len;
char      *prompt;
```

The default value comes from the initial the argument, so the value should be initialized before it is passed in.

2.5.3 Inputting Peabody Things

The following functions input peabody constructs.

```
Figure *
InputFigure(arglist,outargs,prompt)
VALUE      **arglist;
char       **outargs;
char       *prompt;


Segment *
InputSegment(arglist,outargs,prompt)
VALUE       **arglist;
char        **outargs;
char        *prompt;


Site *
InputSite(arglist,outargs,prompt)
VALUE    **arglist;
char     **outargs;
char     *prompt;
```

```
Joint *
InputJoint(arglist,outargs,prompt)
VALUE    **arglist;
char     **outargs;
char     *prompt;


Segment *
InputLight(arglist,outargs,prompt)
VALUE      **arglist;
VALUE      **arglist;
VALUETYPE  type;
char       *prompt;


Attribute *
InputAttribute(arglist,outargs,prompt)
VALUE        **arglist;
char         **outargs;
char         *prompt;
```

2.5.4 Inputting Psurf Items

The following functions input a psurf item. Each function returns a boolean value specifying whether the argument was successfully input. Each function sets the *segmentp* pointer to point to the corresponding segment.

```
InputNode(arglist,outargs,prompt,segmentp,itemp)
VALUE    **arglist;
char     **outargs;
Segment  **segmentp;
short    *itemp;


InputEdge(arglist,outargs,prompt,segmentp,itemp)
VALUE    **arglist;
char     **outargs;
Segment  **segmentp;
short    *itemp;


InputFace(arglist,outargs,prompt,segmentp,itemp)
VALUE    **arglist;
char     **outargs;
Segment  **segmentp;
short    *itemp;


InputCurve(arglist,outargs,prompt,segmentp,itemp)
VALUE    **arglist;
char     **outargs;
Segment  **segmentp;
short    *itemp;
```

```
InputPatch(arglist,outargs,prompt,segmentp,itemp)
VALUE    **arglist;
char     **outargs;
Segment  **segmentp;
short    *itemp;
```

2.5.5 How the Input Functions Work

All of the `Input` functions take as an argument the address of a list of `VALUE` structures. Each function inspects the first entry in the list for the argument of interest. This argument may be of several types, and the action taken depends upon the type:

`V_NUMBER` A number. This is recognized by `InputInt` and `InputFloat`.

`V_STRING` A character string. This can be interpreted differently by different functions. `InputString` simply returns the string. The functions `InputFigure`, `InputSegment`, `InputSite`, and `InputJoint` expect that the string names a peabody construct and it looks for the construct with that name.

`V_UNDEF` An error. This occurs because of a syntax error in the string from which the argument came.

`V_UNSUPPLIED` This special value means to pick the value interactively. Strings and numbers are entered from the keyboard. Peabody constructs are picked interactively.

All of the `Input` functions behave in a consistent manner. If the intended argument was successfully input, then each function returns with the argument. If there was an error, such as a syntax error in the argument list, then the `Input` functions prompt the user to enter the appropriate value.

2.5.6 Inputting Windows

Some operations in *Jack* manipulate the appearance of a window by modifying the parameters of the window. These functions must be able to refer to a specific window. To do this, they use `InputWindow`. Interactively, this function returns the current window, which is the window which currently has the input focus.

```
InputWindow(arglist,outargs)
VALUE  **arglist;
char   **outargs;
```

2.6 Handling Output Messages

Jack has several facilies for printing informational messages. Generally, messages can go in the status window (the blue, one-line window across the bottom of the screen), or in the message window (the text window below the graphics window), or directly in the graphics window.

2.6.1 Using the Status Window

The principal routine for displaying information in the status window is `StatusMsg`, takes a single character string argument.

```
StatusMsg(msg)
char  *msg;
```

The message will remain displayed in the window until another message is written. It is good idea to display a message in the message window before any internal computation which is likely to take more than a second or so. This keeps the user informed about what is going on.

A major use of this facility is in reporting errors. In this case, the message needs to be displayed long enough for the user to see it, but then control usually passes to another part of the program. For this situation, there is the function `StatusError`, which displays a message, beeps the keyboard bell, and pauses one second. The function `StatusPause` does the same thing without the beep.

```
StatusError(msg)
char  *msg;

StatusPause(msg)
char  *msg;
```

2.6.2 Using the Message Window

Messages may be displayed in the message window using `LogMsg`. *The character string msg must end in a newline character!*

```
LogMsg(msg,type)
char  *msg;
int   type;
```

The *type* controls how the text is printed. The following values may be used:

`WSH_NORMAL` Ordinary text

`WSH_HIGHLIGHT` Highlighted text.

`WSH_REVERSE` Reverse video text.

`WSH_UNDERLINE` Underlineed text.

`WSH_HIGHLIGHT_REVERSE` Highlighted reverse video text.

`WSH_UNDERLINE_REVERSE` Underlined reverse video text.

`-1` Use the value from the previous call to `LogMsg`.

2.6.3 Screen Messages

```
int
ScreenMsg(line,buf)
int   line;
char  *buf;

int
ScreenBuf(line,buf)
int   line;
char  *buf;
```

ScreenMsg prints a message on the graphics window at the given line. **ScreenBuf** takes a buffer and chops it into lines delimited by newline characters and displays the lines on the screen using **ScreenMsg**, with the lines decreasing from the given line. It returns one less than the line number of the bottom line written. The function **screenlines** returns the number of lines in the current window.

Screen messages stay on the screen only until the window is redrawn. In other words, the messages are never really erased, they're just "drawn over". Jack automatically redraws all of the windows after it executes every command, so messages written with **screenmsg** disappear after a command finishes.

2.6.4 Reporting Errors

It is important for *Jack* to deal efficiently with errors which occur during the use of the program. The function **StatusError** is generally only useful for reporting small *user* errors, such as incorrect keystrokes, or abort messages.

There are times when more information needs to be displayed than will conveniently fit in the message window. The error reporting facility is described in the chapter on the VEC library. This uses the function **error**, which in the *Jack* environment uses **infomsg** to display the messages on the screen and scrolls automatically when necessary.

```
error(msg)
char  *msg;
```

2.6.5 Reporting Status During Long Operations

An informed user is a happy user. When a *Jack* command does something that takes a long time, it is always a good idea to have the command print occasional status messages that let the user know it is still working, as opposed to being hung up or in an infinite loop. You can do this with the function **timeformessage**.

```
Boolean
timeformessage(n)
int      n;


resetmessagetimer()
```

This function looks at the current time, in milliseconds, and compares it to the last time a message was printed. If more than *n* seconds have passed, it returns **TRUE**. Otherwise, it returns **FALSE**. Embed this function inside of a loop and make it print a status message when it returns true. The function **resetmessagetimer** resets the timer and should be called to initialize the process.

The following example illustrates this.

```
resetmesssagetimer();

for (i=0; i<niterations; i++) {
    DoSomethingComplicated();
    if (timeformessage()) {
        sprintf(msg,"thinking...  (iteration %d of %d)",i,niterations);
```

```
            StatusMsg(msg);
        }
    }
```

2.7 Drawing New Kinds of Objects

Jack has two mechanisms for drawing geometric things other than psurfs in the peabody windows.

2.7.1 Auxillary Drawing Functions

The first mechanism is a list of functions called *auxiliary drawers* which are invoked as each peabody window is drawn, after the environment. These are created with the function `BindAuxDrawer`:

```
SimFunc *
BindAuxDrawer(func,args)
int         (*func)();
void        *args;


UnBindAuxDrawer(ad)
SimFunc  *ad;
```

The function passed in to `BindAuxDrawer` may call any graphics routines it wishes. It is invoked with the *args* arguments, which may point to any allocated chunk of memory. It is called with the current viewing on the matrix stack. The return value of the function is a pointer which can be passed to `UnBindAuxDrawer` to delete the function from the list of drawers.

2.7.2 Drawing Segments without Psurfs

Normally, *Jack* draws a segment in the peabody windows by drawing the segment's psurf. If there is no psurf, the segment is not drawn. It is also possible to have the segment drawn by another means by assigning the segment a *drawer* field. If the *drawer* field of the segment is set, then the psurf, if one exists, is ignored and the the *drawer* function is invoked, with the segment and the segment's data field as arguments. This function is invoked when the current modeling transform for the segment is on the matrix stack, so the drawing function should draw the object in local coordinates. This allows you to define arbitrary kinds of objects which can be treated as peabody segments, meaning they can be picked and moved around.

2.8 Dealing with Windows

Sometimes a command needs to redraw the screen. Generally, this is done with `DrawWindows`, which draws all of the windows. This is generally the best way to redraw the screen. If your applications is causing the motion of an object, it is important to redraw *all* the windows so the object will appear to move from all views. `DrawWindow` redraws a single window, but it does *not* swap the buffers, so the newly drawn window will not appear until `swabuffers` is called. This allows you to draw auxiliary information over the window if necessary.

Sometimes is it convenient to draw all the windows *except* for the current one, probably because you want to draw the current one explicitly. This can be done with `DrawOtherWindows`.

DrawWindows does not draw "frozen" windows. If the user wants to disable the display of some of the windows for efficiency, he may "freeze" the windows. The function **DrawAllWindows** draws all the windows, even the frozen ones. This is done automatically after each command is executed.

```
DrawWindow(window)
Window  *window;

DrawOtherWindows(window)
Window  *window;

SwapWindowBuffers()

DrawWindows()

DrawAllWindows()
```

The functions **DrawWindow** and **DrawOtherWindows** do *not* swap the buffers using **swapbuffers**, so their effect will not be apparent until this is done. The function **SwapWindowBuffers** calls **swapbuffers** in each window. This routine ensure that the all *Jack* windows are updated simulaneously, rather than one at a time when the drawing is slow.

The window drawing routines leave the matrix stack with the viewing transform. This enables other routines to draw auxiliary information over a window without having to restate the view.

It is also possible to inquire about the "current" window, which is the window with the current input focus. The variable **Jack.window** always points to the window with the current input focus.

2.8.1 Creating New Windows

Windows are created with **NewWindow**:

```
NewWindow(type,name,l,r,b,t)
WindowType  type;
char        *name;
long        l,r,b,t;
```

The type is an identifier which differentiates the window from other types of windows. If *name* is nil, then **NewWindow** generates a unique name itself. The l, r, b, t arguments specify the position of the window in screen coordinates. If $l \geq r$ or $b \geq t$, then the window is opened interactively by the window manager.

The function **FindWindow** returns a pointer to the window with a given name, or nil if a window with that name cannot be found. The function **FindWid** returns a pointer to the window with a given window manager id, or nil if a window with that id cannot be found.

```
Window *
FindWindow(name)
char       *name;
```

```
Window *
FindWid(id)
int       id;
```

2.8.2 Making Your Own Kind of Windows

It is relatively straightforward to create new kinds of windows which draw different types of things inside of *Jack*. It is important, however, to create these windows using the *Jack* window structure, rather than just opening them with the IRIS GL routine `winopen`.

The *Jack* window structure is described in Section 3.2. The window has a *drawer* function and a *data* field. The data field may point to a location in memory which contains data relevant to the window. The *drawer* function is the function which draws the windows. This function is called with the window structure as an argument. This function should call the IRIS GL subrouting `winset` to make sure that the graphics are directed to the proper window. The *drawer* function is completely responsible for drawing the window, including setting up the view. This function is called frequently, so the function should be as efficient as possible. For example, the function should not do a large amount of computation. If necessary, it can compute data for the graphics and store it a field in the *data* pointer.

Different types of windows may also process mouse and keyboard events differently. Each window has an *eventhandler* function and this function is called with events that occur while the mouse is inside that window. This process is described in Section 3.3. The *eventhandler* function is invoked with the type of event which has occurred, but the event has not yet been read from the queue. The handler function should then read from the queue, using the IRIS GL routine `qread`. The event handling function is free to interpret the event in any way.

The process of creating a new kind of window begins by calling `NewWindow`. You should pass it a type which does not conflict with other window types. Choose an integer greater than 100 and you'll be safe. Then assign the drawer and event handler functions to the appropriate fields in the window structure.

2.9 Writing Interactive Applications

One of the most important functions of the *Jack* interface is its management of the mouse and keyboard. The mouse and key event manager is built on top of the IRIS queuing routines and provides a means of defining significant events and waiting for them to occur. When events occur, they are automatically placed by the hardware in an *event queue*. The event manager governs how the events are interpreted from the queue. Most actions in *Jack* are initiated and terminated with clicks of the mouse. These routines provide a way of using the mouse and keyboard to control the program.

It is important to use these functions in processing events because the window manager generates special events for redrawing windows and changing input focus which cannot be ignored but are difficult to process.

There are two different types of events: mouse events and keyboard events. Mouse events occur when the state of a mouse button changes, i.e. goes up or comes down. The down click and up click of a mouse button are different events. A single keyboard event occurs when a key is pressed. Releasing the key does not generate an event.

```
Device
WaitForEvent(val,wintype)
Device  *val;
int     wintype;


Device
WaitForKeyEvent(wintype)
int     wintype;
```

```
Device
WaitForMouseEvent(val,wintype)
Device  *val;
int     wintype;


Device
WaitUntilEvent(wintype)
int     wintype;
```

WaitForEvent operates just like the IRIS Graphics Library routine **qread**: it returns the device, with its value stored in *val*. **WaitForKeyEvent** returns with the ASCII code for the key pressed. **WaitForMouseEvent** returns the device (mouse button) pressed, with setting *val* to 1 if the click was "up," and 0 if the click was "down." **WaitUntilEvent** waits until an event occurs, but it doesn't read the event from the queue.

Each of these functions takes a *wintype* argument which specifies the type of window in which it wants the event to occur. If an event occurs in another type of window, it will be ignored. If the *wintype* is -1, it this effectively ignored and any window type is acceptable. This mechanism does not work reliably, and it is best to use -1 to receive an event in any window then test to ensure that the window is the proper type.

The control key and the shift keys cannot be queued, but their position may be sensed with the macros **CONTROL** and **SHIFT**. This gives a way of interpreting control characters and capital characters.

```
CONTROL()


SHIFT()
```

The function **devname** is useful for debugging event-driven routines. It returns the character string name of an event, thus allowing code to print debugging information about what events have occurred. All IRIS events are included, event the weird ones.

```
char *
devname(dev)
Device  dev;
```

2.9.1 Keyboard Input

The keyboard is treated as a *device,* so that it is not convenient to read from the terminal using standard I/O routines like **scanf** and **gets**. However, keyboard input may be input directly with the function **getkeyboardstring**, which displays a prompt in the message window and then reads a string from the keyboard.

```
char *
getkeyboardstring(buf,prompt,ncomp,compstr,previnput,menu)
char    buf[];
char    *prompt;
int     ncomp;
char    *compstr[];
List    previnput;
MENU    menu;
```

This function does automatic completion, based on the values specified by *ncomp* and *compstrs*. *compstrs* is an array of character strings. Its length is given by *ncomp*. The user *may* enter any string he or she wishes, although the automatic completion will be done only on these values. These values will also be placed in the pop-up menu which the user gets by pressing the right mouse button. The *previnput* argument is a list of previously input strings, which the user may retrieve by hitting `^P` and `^N`. The *menu* argument should be `NULL`.

`InputStringComplete` invokes this function to get the value from the keyboard. `InputStringCompete` should be used unless you are absolutely sure you want the input to come from the keyboard and not from a JCL script.

2.10 The Jack Movement Operator

The movement routines provide a means of inputting general homogeneous transforms interactively. The basic movement operator is `MoveTransform`, which interactively determines a transform from the position of the mouse, while executing an arbitrary function as it goes.

```
Boolean
MoveTransform(Transform *trans,
        Transform *ref,
        char *message,
        int flags,
        int (*action)(void *args[]),
        int (*drawer)(void *args[]),
        int (*binder)(int key,Boolean *done, void *args[]),
        int (*initializer)(void *args[]),
        void *args[])
```

This *action* function may be used, for example, to update the position and orientation of objects, the displacement of joints, the location of nodes or control points, or the goals of constraints.

The *trans* transform is a local transformation. It specifies a global position relative to *ref*, which is in turn a global transformation, i.e. specified with respect to the base coordinate frame. `MoveTransform` continuously updates *trans* based on the input from the user. The *ref* transform remains fixed. The *invert* argument specifies whether or not the transformation being manipulated interactively is the inverse of the transform requested in *trans*.

`MoveTransform` is a loop which continuously does the following:

1. read the state and position of the mouse and keyboard
2. update *trans* accordingly
3. call *action* with *args*:

```
(*action)(args)
```

4. execute the function `AdvanceSimulation`.

5. redraw the graphics windows (draw the current one last, so that it leaves this as the current GL window).

6. call *drawer* with *args*.

The *msg* argument to `MoveTransform` is displayed in the message window to inform the user of what is going on. This arrangement allows the *action* function to distribute the effect of the new transform to the appropriate parts of the environment, then give the *drawer* function the opportunity to draw important information on the screen, in terms of highlighted segments, icons, or screen messages. Notice that `MoveTransform` does not take a window as an argument. It moves the transform in whatever window the mouse cursor is in when the buttons go down.

`MoveTransform` returns a boolean value specifying whether the transform was successfully moved. It will return `FALSE` if an error occurs or if the user aborts the move.

`MoveTransform` is currently called as a part of several facilities in *Jack*:

- The `move figure` command. In this case, *ref* is the identity transform, and the *action* function applies the `trans` transform to the location of the figure being moved, using `SetFigureLocation`.

- The `move site` command. In this case, *ref* is the global transform of the site's segment, and the *action* function applies the `trans` transform to the location of the site being moved, using `SetSiteLocation`.

- The constraint moving function `MoveConstraint`. This is used by the `interactive reach` command, and by the human figure manipulation commands. In this case, the constraint must have a goal type of `V_MATRIX`. The *trans* transform is the identity, and the *action* function is `updateconstr`, which applies the *ref* transform to the goal of the constraint.

2.11 Simulation Functions

Jack maintains a list of *simulation functions* which it invokes at each interactive iteration. You may add functions to this list with the routine `BindSimulationFunction`. This routine returns a pointer to a `SimFunc` structure. This value may be passed later to `UnBindSimulationFunction` to unbind the function.

```
SimFunc *
BindSimulationFunction(func,args)
int         (*func)();
void        *args;


UnBindSimulationFunction(sf)
SimFunc   *sf;
```

These functions may do anything you like. The purpose of these functions is to provide access to the interactive nature of *Jack*. These functions are invoked repeatedly, both as *Jack* is waiting for input from the user and as the user moves objects or changes the view. Therefore, if you write a simulation function which causes an object to move in a certain way, you may still interact with the object by changing the view or moving other objects, even as it moves under the control of the function.

Section 3.4 describes more about the *Jack* simulation mechanism.

2.12 The RecordArgument Functions

This section describes the **RecordArgument** functions, which are used mostly by the *Jack* **Input** functions to add elements to the *outargs* parameter. The *outargs* parameter is passed to the **CMD** functions to keep a record of what parameters were entered. This enables *Jack* to record in JCL format the commands it has executed, with the appropriate arguments. Each **Input** function takes an *args* parameter of *input* values, and an *outargs* parameter, which is the address of a character string pointer. The **RecordArgument** functions take this parameter and copy a character string representation of a value input by the user into the string.

```
RecordArgument(outargs,string)
char  **outargs;
char  *string;


RecordArgumentString(outargs,string)
char  **outargs;
char  *string;


RecordArgumentInt(outargs,n)
char  **outargs;
int   n;


RecordArgumentFloat(outargs,f)
char   **outargs;
float  f;
```

The function **RecordArgument** takes a string and records it as is. This is appropriate for the names of peabody constructs. **RecordArgumentString** records a string but encloses it in double quotes. This is appropriate for quantities like the weight functions or velocity controls.

These functions are called in the **Input** functions. They are also called in the **inputparams** functions for motions, as described in Section 2.12.

2.13 Named Types

Jack has a facility for associating character string names with enumerated types. This mechanism is convenient for allowing the use to choose between several options by selecting a character string name. The user may enter the name from the keyboard or may select it from a pop-up menu.

The **NamedType** structure is defined as:

```
typedef struct {
        char    name;
        int     type;
} NamedType;
```

The most important function in using this facility is the **Input** function **InputNamedType**, which allows the user

to enter the type based on the names in the array of **NamedType** structures. This uses the automatic completion facility, and the user may also select the items from a pop-up menu.

```
int
InputNamedType(arglist,outargs,typep,types,prompt)
VALUE       **arglist;
char        **outargs;
int         *typep;
NamedType   types[];
char        *prompt;
```

InputNamedType returns the *type* associated with the *name* selected. It is your responsibily as a programmer to construct the array of **NamedType** structures. The array must be terminated with a type with a **NULL** name pointer. This signifies the end of the array. Figure 2.1 shows an example of the use of this facility.

```
NamedType balance_controls[] = {
    { "between feet",           BC_FEET },
    { "hold current balance poinBC_HOLD },
    { "hold current elevation", BC_HOLD_ELEV },
    { "release elevation",      BC_RELEASE_ELEV },
    { "seated",                 BC_SEATED},
    { NULL, 0},
};


    if (!InputNamedType(arglist,outargs,&type,balance_controls,"balance:  ")) {
        return(0);
    }
    if (type == BC_FEET) {
        ...
    }
```

Figure 2.1: **InputNamedType**

There are also the following useful functions for dealing with arrays of **NamedType** structures.

```
char *
TypeName(type,namedtypes)
int         type;
NamedType   namedtypes[];


Boolean
FindType(typep,name,namedtypes)
int         *typep;
char        *name;
NamedType   namedtypes[];
```

```
int
TypeNames(names,namedtypes)
char        *names[];
NamedType   namedtypes[];
```

TypeName takes a *type* and returns a pointer its *name*. **FindType** takes a *name* and returns the corresponding *type*, placing it in the *typep* argument. The function **FindType** returns **TRUE** if it finds the named type, or **FALSE** if the name does not correspond to a legal type. **TypeNames** fills the name pointers into the array *names*. It returns the number of entries in the array.

Chapter 3

How Jack Works

This chapter describes how *Jack* works. The purpose of this chapter is to give an overview of the flow of control and data structures in *Jack*. This chapter differs from the other chapters in this manual in that the functions described here are not sub-routines which can be used in other contexts, such as in the development of other *Jack* features. The reason for discussing these function is to explain what happens inside of *Jack*.

The programmer interface for defining new commands is fairly clean, so that it is generally not necessary to understand completely how the internals of *Jack* work, unless you need to modify it in some way. However, it is a good idea to read this chapter just so you have an overall picture of how *Jack* works.

The easiest way to understand how *Jack* works is to actually follow the source code, since it is inherently more up to date than this manual.

This chapter quotes directly from the source code in an attempt to explain critical features, but in most cases the code given here is just a synopsis of the actual code. This is to avoid some of the confusing aspects of the implementation. The purpose of showing the code here is to give the overall picture of how the individual routines works.

3.1 The Jack Program Structure

Jack as a program consists of several major components.

- The peabody environment.
- The notion of windows and the *Jack* window drawing routines.
- The notion of *Jack* commands and how they are executed.
- The *Jack flow of control* mechanism.
- The simulation mechanism
- The motion system

The primary part of *Jack* is its peabody environment. This is the internal "database" of geometric information. The purpose of *Jack* as a program is to display and manipulate these objects in various useful ways. *Jack* has a single peabody environent to which all geometric information belongs.

Jack's windows are its mechanism of displaying information. *Jack* has an object-oriented representation for windows which makes it convenient to define windows which draw different types of graphics. The primary type of window, called a "peabody" window, displays the geometric environment. When *Jack* draws such window, it looks into the peabody environment to determine what to draw.

Jack is a command driven system: it executes a sequential stream of instructions. The commands may be invoked from the menus, from the keyboard, or from a file or input stream. The *Jack* commands roughly correspond to the items in the pop-up menus. All communication between the "user" and the internals of *Jack* is accomplished through these commands. New features may be added to *Jack* by writing new commands.

Commands have arguments, which are the operands of the operation. The arguments are the things which the user must enter or pick after executing a command, such as numbers, strings, or parts of the peabody environment. *Jack* takes a *verb-object* approach to command execution. The user first picks the operation and then specifies which objects are to be acted upon.

Jack is primarily an interactive system, so the most common way of executing commands is by interactively picking items from the pop-up menus. However, there are many circumstances in which it is necessary to control *Jack* non-interactively, so that the sequence of commands and arguments comes not by selecting items from the menus and picking things with the mouse but from the keyboard or from a text file. By "non-interactive" we mean simply that the commands and arguments are not picked interactively but are read from a text string or file.

The method of controlling *Jack* non-interactively is called the *Jack Command Language*, or JCL. This refers to the way in which command names and arguments are read in text form. JCL is the way of communicating with *Jack* in a syntactic, textual way. One of the features of JCL is the ability to instruct *Jack* to pick certain arguments interactively, so that JCL "scripts" may actually be *semi interactive*, meaning that some of the arguments are provided in text form and some are to be picked interactively.

3.1.1 The Jack `main` Procedure

The `main` procedure for *Jack* is a part of the *Jack* library, in the source code file `gen/src/lib/jack_main.c++`. The responsibility of the `main` procedure is to:

1. initialize the *Jack* internals
2. read command line options
3. create the message window
4. create the status window
5. create a peabody window
6. initialize the commands and menus
7. read files given on the command line
8. enter the *Jack* control loop, and continuously execute commands

3.1.2 Jack Variables

Jack's global variables are kept in a single global structure called `Jack`. This is largely for clarity in the source code, since it causes most of the important *Jack* parameters to be prefixed by `Jack.`, thus identifying them as *Jack* variables.

Some of the variables in the `Jack` structure are parameters, in the sense of being variables which control the program's behavior. This structure is also a holding place for global lists and pointers. The meaning of the individual values is described in the *Jack* include file `jack.h`. The most important ones are described in the text below.

The main procedure in `gen/src/lib/jack_main.c++` has the default settings for the *Jack* parameters.

3.1.3 Jack's Colors

Jack has a set of colors which it uses to draws various things. These colors are kept in the global `Jack` structure, and they are initialized in `main.c++`. The colors of the peabody objects are an exception, since they are taken from the surface attributes to which the objects refer.

Each color value is a long integer. Internally speaking, this value is used as the argument to the IRIS GL routine `cpack`. This 4-byte quantity stores the red value in the low byte, the green value in the second byte, and the blue value in the third byte. It is convenient to initialize these values in hex, so that

```
color = 0xff077f;
```

corresponds to the rgb value (128,7,255). (7f in hex is 128 for red, 07 in hex is 7 for green, and ff in hex is 255 for blue.)

3.2 Jack Windows

Jack has an object oriented notion of a window. The structure of a *Jack* window is:

```
typedef struct {
    int             type;
    int             wid;
    char            *name;
    float           aspect;
    void            *data;
    int             (*drawer)();
    int             (*eventhandler)();
} Window;
```

The *type* field specifies the type. This is for identification purposes only. The function of the window is defined through the functions *drawer* and *eventhandler*. The purpose of the *eventhandler* is described in Section 3.3. The *data* field points to an arbitrary data block containing parameters which control the window. This data pointer is handled only by the *drawer* and *eventhandler* functions. The *name* of the window is displayed in its title bar. The *wid* field is the **mex** window identifier, used to identify the window to the window manager. Other information about the window, such as its dimensions and location on the screen, can be inquired directly from the window manager, using the *wid* field. All window types are doublebuffered and use IRIS GL **RGBmode**.

Peabody windows draw the *peabody* environment. The corresponding *drawer* function is **DrawPeabodyWindow**, described in Section 3.2.2.

3.2.1 How Jack Draws the Windows

Windows are drawn with the function **DrawWindow**, which is shown in Figure 3.1. As you can see, **DrawWindow**

```
DrawWindow(window)
Window      *window;
{
    (*window->drawer)(window);
}
```

Figure 3.1: **DrawWindow**

does very little work: it passes the responsibility to other routines which draw different types of windows.

In the **4Sight** environment, graphics commands go to the "current" window, which is set with the IRIS Graphics Library function **winset**. The lower level drawing routines, such as **DrawPeabodyWindow** and **DrawMeterWindow**, should set the window before drawing.

3.2.2 Peabody Windows

Peabody display the *peabody* environment. These windows use the parameters in the structure **PeaWin**, which describe the view, shading parameters,and the background grid.

```
typedef struct {
    Site    *camera;
    float   vrpd;
    short   fov;
    float   near,far;
    Grid    *grid;
    int     mode;
    float   attenuation[3];
    Vector  ambient;
    int     lmodel;
} Peawin;
```

Peabody windows are drawn internally by the function **DrawPeabodyWindow**, which is defined in the source code file **gen/src/lib/jack_peawin.c++**. This function is fairly intricate because of the requirements of initializing the zbuffer, the matrix stack, and the lighting model. An outline of the critical features of this function is:

- It always uses the Z-buffer. The time savings for not using the z-buffer in wireframe mode are not worth the artifacts it causes.
- It represents the view through the global location of the window's camera site, as described in Section 3.2.2.1.
- It stores the projection matrix internally in the *projmat* , and it uses **MPROJECTION** mode to put the projection matrix on the stack. It then draws the windows in **MVIEWING** mode.
- The *lmodel* field is the IRIS GL lighting model for the window. Each window has a different lighting model because it may have different lighting parameters.
- It draws the peabody environment using the function **TraverseEnvironment**. This function calls the functions **tree_drawsegment** and **tree_drawsite** at each segment and site, respectively, i.e. when the appropriate modeling transform is currently on the matrix stack. These functions must set the lighting model based on the window mode, because shaded and wireframe objects may occur in the same window.
- The coordinate axis projections are drawing in **MSINGLE** mode because they involve a 0-scale component which is not legal in **MVIEWING** mode.
- In addition to the peabody environment, the function draws constraint icons, traces, transforms, and "auxiliary" objects.

3.2.2.1 The View

The view for a *peabody* window is represented by a site, pointed to by the *camera* field. The global position of the camera site defines the location and orientation of the eye point, or center of projection. The line of sight is down the negative z-axis of the camera site. The global y axis of the camera is the "view up" vector, with the global x axis extending to the right of the screen. This convention makes the inverse of the global transformation of

the camera site the viewing matrix. Thus `DrawPeabodyWindow` sets the viewing projection with the IRIS Graphics Library function `perspective`, and then inverts and multiplies by the global transform of the camera site.

The *near* and *far* fields maintain the near and far clipping planes. Normally, the settings of the clipping planes are not very important. The *fov* angle represents the field of view, which is usually 40°. Note that the *aspect* field is common to all types of windows.

The *vrpd* and *defaultcamera* fields are provided for convenience during adjusting the view. The *Jack* viewing routines move the view around a *view reference point*, which is located along the line of sight at a distance of *vrpd* from the center of projection. This value effectively changes the radius of the revolution of the camera. The *defaultcamera* points to the site to which the view is normally attached. The view may be attached to any site in the environment. This allows the view to be attached to arbitrary figures, but it can result in some confusing things unless there is an easy way to reset the view. The *defaultcamera* site is the one to which the view should be reset after it has been attached to something else.

3.2.2.2 The Background

The *grid* field in the window structure points to a background grid structure, which is the plane of lines drawn as the background. This structure is defined as:

```
typedef struct grid Grid;

struct grid {
    float       origin[3],bounds[3];
    float       majorgridsize,minorgridsize;
    Object      obj;
    Object      stars;
    unsigned    fillbkg :  1;
    unsigned    xoplane :  1;
    unsigned    yoplane :  1;
    unsigned    zoplane :  1;
    unsigned    xbplane :  1;
    unsigned    ybplane :  1;
    unsigned    zbplane :  1;
};
```

The grid structure represents a six-plane volume in space. The "lower" corner of the volume is specified by the *origin* field. The "upper corner of the volume is specified by the *bounds* field. The "plane" flags specify which planes are to be displayed. Normally, only the *yoplane* is on. Each plane is drawn as a grid of lines drawn in the `minorgridcolor` spaced at intervals of *minorgridsize* and lines drawn in the `majorgridcolor` spaced at intervals of *majorgridsize*. The axis lines are drawn in the `axiscolor`. The *obj* object is the graphical object which draws the grid. The grid is drawn in unit space, so the coordinate system must be scaled before drawing.

3.2.2.3 The Coordinate Axis Projections

The projections onto the ground plane are drawn by encorporating a zero scaling factor into the viewing tranform. The funcion `projectfunc` does this. This function inspects the grid flags of the window and draws the projections onto the appropriate ones by setting the scale to zero in the appropriate direction and calling *func*, with *args*. Before `projectfunc` is called, the matrix stack should be initialized with the viewing transform, but *not* the modeling transform. `projectfunc` is responsible for setting the modelling transform itself.

3.2.2.4 The Peabody Environment

The function `TraverseEnvironment` from the peabody library traverses the environment tree, multiplying matrices across segments and joints, invoking user-supplied functions at the sites, segments, and joints when the appropriate modeling transforms are on the stack. `DrawPeabodyWindow` uses this function to draw the segments and sites. It uses the functions `tree_drawsegment`, defined in `gen/src/lib/jack_segment.c++`, and `tree_drawsite`, in `gen/src/lib/jack_site.c++`. Three is no joint-drawing function because joints are not physical "things" which need to be drawn. Both of these functions take as arguments an array of `void` pointers, which are cast to the proper types. The arguments include the window's drawing mode and the window's lighting model.

The function `tree_drawsegment` must determine how to draw the object based on the window drawing mode and the segment drawing mode. The mode values are:

-1 Coordinate axis projections: draw in wireframe in half-intensity.

0 Use the segment's drawing mode.

1 Ignore the segment's drawing mode and draw in wireframe.

2 Ignore the segment's drawing mode and draw in shaded mode, using the lighting model.

The psurf is drawn using the psurf display lists, as described in Section 8.1.4. The function examines the field *display.on* before drawing the segment. This allows segments to be turned off.

The function `setlighting` takes a lighting model as a parameter and sets the GL lighting model accordingly. A value of 0 turns off lighting, which is appropriate for wireframe drawing. The macro `CPACK` takes a color parameter and a drawing mode and sets the current color using `cpack`. If the mode is -1, it first descreases the intensity by half.

The function `tree_drawsegment` also examines the segment's *drawer* function before drawing the segment. If this function exists, it invokes it instead of drawing the psurf. It passes the segment's *data* field as an argument. This allows *Jack* to draw segments with geometry other than psurfs.

3.3 The Jack Flow of Control

Jack is an event driven system, in the sense that its flow of control is governed by events, as represented through the IRIS GL event queue. Events are generated through the mouse and keyboard. *Jack* does not use standard input at all. All of its keyboard input is derived through the event queue.

Jack loops continuously until an event occurs, and then it processes it. The act of *processing* an event involves not only examining the type of event but the context in which the event occurred.

The event processing mechanism in *Jack* is closely tied to its notion of windows. Different types of windows may handle events in different ways. Each window has an event handler function which is invoked when an event occurs when that window is *current*. A window is current when it has the input focus, which usually means the mouse is inside that window.

The source code file `gen/src/lib/jack_control.c++` has the main *Jack* control loop, which is the function `DoCmds`. A synopsis of `DoCmds` is shown in Figure 3.2. This synopsis shows that the function is quite simple: it waits for an event and then *handles* it. The first step is done by the function `SimulateUntilEvent`. A synopsis of this function is shown in Figure 3.3. The argument to this function is a type of window. The function loops until an event occurs in this type of window. Events which occur in other windows are processed transparently. A value of -1 means that any type of window is acceptable.

The function `IsEvent` detects when an event occurs, but this function is designed to transparently handle window manager events like `REDRAW` and `INPUTCHANGE`. `REDRAW` events are generated by the window manager when a window is moved or resized. `INPUTCHANGE` events signify when the mouse moves into or out of a window. It is absolutely essential that *Jack* handle these events properly. This means that only the event handling routines should access the IRIS GL event queue directly.

The critical aspect of the function `SimulateUntilEvent` is that it calls the function `AdvanceClock`. This function is at the heart of *Jack*'s notion of time. It is the responsibility of this function to make changes in the state of *Jack* and the peabody environment. This function is discussed in detail in Section 3.4.

When `SimulateUntilEvent` detects an event, it returns. Note that it does not read the event from the queue, it merely returns with the type of event which has occurred. The function `DoCmds` then invokes the function

```
void
DoCmds(mainmenu)
MENU *mainmenu;
{
    short   dev;

    while (!Jack.done) {

            /*
             * Wait until an even occurs...
             */
            dev = SimulateUntilEvent(-1);

            /*
             * ...  and process what has happened.
             */
            windoweventhandler(dev);
        }
    }
}
```

Figure 3.2: `DoCmds`

```
short
SimulateUntilEvent(wintype)
int wintype;
{
    short   dev;

    while (!(dev=IsEvent(wintype))) {

        AdvanceSimulation();

        DrawOtherWindows(Jack.messagewindow);
        DrawWindow(Jack.messagewindow);

        SwapWindowBuffers();
    }
    return(dev);
}
```

Figure 3.3: `SimulateUntilEvent`

windoweventhandler, which reads the event from the queue and invokes the current window's handler function to act accordingly.

For peabody windows, the handler function is the function **PeaWinHandler**. This procedure performs the ordinary *Jack* input function. If the event is the right mouse button, it puts up a pop-up menu to select a command. If it is a keystroke, i.e. a **KEYBD** event, it looks to see if the key is bound to a command. If so, it executes the command. If not, it begins getting the input from the keyboard with the command **DoKeyboardCommand**. This command reads a string from the keyboard with the function **getkeyboardstring**, which uses automatic completion to assist in entering the data. In addition, the function **DoKeyboardCommand** allows the string which comes from the keyboard to include both the command name and arguments. It then peels the command name off the front of the string and passes the remaining words to the command function as its arguments. This allows the user to type command arguments on the same line as the command name.

3.3.1 Other Occurrences of Events

Any operation in *Jack* which pauses for input from the user, particularly through the mouse, must be sure to handle the event properly. The two most common situations calling for this behavior are the view changing mechanism and the direct manipulation mechanism (moving figures, adjusting joints, etc). These routines loop continuously, waiting for events to occur, similar to the *Jack* control loop in **DoCmds**. When an event occurs, each routine can handle it as it sees fit. In the case of **ChangeView** and **MoveTransform**, each routine looks to see if the event was a keystroke bound to a command.

3.3.2 Implemention Issues

The event handling routines in *Jack*, **WaitForEvent**, **WaitForKeyEvent**, **WaitUntilEvent**, **SimulationUntilEvent**, and **IsEvent**, each accept an argument which describes the type of window inside with the event should occur. This mechanism was never really fully developed, and it doesn't work, so these values should be -1 to signify that any type of window is acceptable. It is generally much better to get the event and then test to see if the current window is of the proper type.

3.3.3 Jack Commands

The command facility defines data structures for commands and menus. A *command* has a character string name, a pointer to a function, and some auxiliary arguments to pass to the function when it is called.

```
typedef struct cmd CMD;

struct cmd {
    char    *name;
    int     (*func)();
    ARGS    args;
    MENU    *menu;
};
```

Commands may be executed in several ways, and the name of the command may be different depending upon how it is used. The *name* field of a command is the character string used for the menu item in the pop-up menus. This string typically consists of several English "words" separated by spaces. When commands are executed non-interactively, the command names are read from a text string, either from the keyboard or from a file. In this case, the name of the command must be a legal identifier, thus it must consist of letters, digits, and underscores.

The operation associated with a command is performed by the function pointed to by the `func` field, which is a pointer to a function. When a command is executed, this function is called with certain arguments. There is an important distinction between a command and a command's function. The function is a C subroutine; the command is a *Jack* menu item. Every *Jack* menu item has a function associated with it, but in certain circumstances different *Jack* commands may use the same function.

3.3.4 The Execution of Commands

Jack was originally designed to be an interactive system, with the assumption that the best way to operate on objects is to pick them by pointing at them with the mouse and move them by moving the mouse until the position looks right on the screen. This makes *Jack* a useful interface, but there are many circumstances when it is convenient to be able to control *Jack* non-interactively, i.e. provide information about commands and arguments in text form.

This is a simple matter in the case of selecting commands, since *Jack* commands have names, and given a character string name, it is easy to find the associated command. In the case of the command *arguments*, the matter is much more complex. The arguments are generally the operands of an operation. Interactively, the arguments are the things which the user enters. Non-interactively, the arguments are the parameters to the command "call."

Jack must have some mechanism of disecting a text string into arguments for a command, and each command's function must be able to perform the same action regardless of whether the arguments are being entered interactively or being disected from a text string.

3.3.5 Command Arguments

The approach which *Jack* uses is to supply to the command's function a list of "arguments" in the form of `VALUE` strutures, defined by the peabody library. The `VALUE` structure contains a "type" and a "value." The type field is of type `VALUETYPE`. The legal types for command arguments are `V_STRING`, `V_NUMBER`, `V_MATRIX`, or the special types `V_UNDEF` and `V_UNSUPPLIED`, in which the value is empty. For a description of the `VALUE` structure, see Section 7.6.

The next chapter describes how to go about defining commands and writing command functions, and it gives guidelines for extracting the arguments from the `VALUE` structures. This section attempts to explain how the execution of the commands is accomplished.

When *Jack* executes a command, it calls the commands function with three basic arguments. The first argument is a list of `VALUE` structures which are the command's arguments. The function should inspect this list to determine what arguments have already been provided and what further input, if any, is required from the user.

The second argument to the function call is the *output* arguments, which is a record of which arguments were used for the execution of the command. This is a pointer to a character string, into which the function should print the arguments it has received. This buffer collects the executed arguments as a JCL record of what has been executed. This allows sequences of commands to be recorded and written to JCL files.

The third argument to the command functions is a chunk of type `ARGS`. These arguments are declared in the definition of the command itself, usually done by `MkMenuCmd`, described in the next chaper. The extra arguments given in the command definition are passed to the command's function as the third argument when the function is called.

Jack executes commands with the function `EvalCmd`. This is the *single* place in which the function associated with a command is called.

3.4 The Jack Simulation Procedure

The *Jack* simulation procedure is `AdvanceSimulation`. The purpose of this function is to advance the notion of time by one clock tick. This is the foundation of the interactive manipulation mechanism in *Jack*, as well as the animation system. The function `AdvanceSimulation` is shown in Figure 3.4.

The simulation procedure is set up to work slightly differently if *Jack* is advancing a simulation or just doing interactive manipulation. This is signified by the boolean flag *Jack.advancetime* specifies whether *Jack* is

```
void
AdvanceSimulation(void)
{
    if (Jack.simulation == 0) {
        return;
    }

    if (Jack.advancetime) {
        if (Jack.currenttime >= 0 && !FindFrame(Jack.currenttime)) {
            SaveFrame(Jack.currenttime);
        }
        Jack.currenttime++;
        Jack.advancetime = AdvanceTime();
    } else {
        Jack.currenttime++;

        ExecuteBehaviorFunctions();
        EvaluateConstraints(Jack.constr.timelimit,Jack.constr.stepfactor,NULL);
        ExecutePostBehaviorFunctions();
    }

    if (Jack.collision_detection != CD_NO_DETECTION) {
        HandleCollisions();
    }

    ExecuteSimulationFunctions();
}
```

Figure 3.4: AdvanceSimulation

currently advancing the time clock of the simulation. If this is set, it invokes the function **AdvanceTime**, described in Section 5.4. This is the heart of the animation system. If it is not advancing time, then it executes the behavior functions and evaluates the constraints, subject to the *timelimit* and *stepfactor* parameters.

The **AdvanceTime** function is described in Section 5.4. It is the heart of the animation system. It would be possible to have the implementation of the manipulation system and animation system folded together. To do this, there would be no special case for manipulation in **AdvanceSimulation**. This would be the case if the *Jack.advancetime* flag were always true. There is no real need to have these be separate, but they were originally implemented that way and so they have remained that way. The biggest difference between these two cases is that during an animation, there should not be oscillation in the constraints, particularly with the balance. This oscillation is inherent whith the center of mass constraint, so it is necessary to evaluate the constraints repeatedly until the center of mass converges. This is not done during interactive manipulation, because it is more time consuming. This is described in Section 5.4.

AdvanceSimulation calls the function **ExecuteBehaviorFunctions**, which does several things to human figures, described in Section 4.3. Since **AdvanceSimulation** is invoked during the main *Jack* control loop and while interactively manipulating any object, these behaviors apply during these times as well as during motions. **AdvanceSimulation** evaluates the constraint subject to the time limit and step factor parameters, as described in Section 6.6. The function **ExecutePostBehaviorFunctions** executes behavior functions which need to be performed after the evaluation of the constraints.

The function **AdvanceSimulation** is called in several places in *Jack*:

- In the function **SimulateUntilEvent**. This occurs in the main control loop, when *Jack* is waiting for input from the mouse or keyboard, and also in the direct manipulation operators **MoveTransform** and **ChangeView** when they are waiting for input as well.

- Inside the iteration loops of the direct manipulation operators **MoveTransform** and **ChangeView**, while the mouse is down and they are repeatedly reading mouse coordinates and generating geometric information.

The simulation procedure **AdvanceSimulation** then checks for collision detection, if the collision detection flag is set. This is done by the function **HandleCollisions**.

Finally, **AdvanceSimulation** executes the simulation functions. These functions provide a convenient way of adding functionality to the simulation loop.

3.5 Picking

Jack has an interactive picking mechanism built around the function **Pick**, which is defined in source code file **gen/src/lib/jack_pick.c++**. This function provides an object oriented approach to picking by defining the objects to pick through a set of functions. It is possible to use this function to interactively pick new types of objects by constructing a set of functions which perform the proper operations on the objects.

The function **Pick** is define as:

```
Pick(type,name,message,segmentp,itemp,
          picklister,namer,highlighter)
VALUETYPE  type;
char       *name;
char       *message;
Segment    **segmentp;
short      *itemp;
int        (*picklister)();
int        (*namer)();
int        (*highlighter)();
```

The argument *type* is a **VALUETYPE** which is used only to save the previous values when picking elements like segment, sites, joints, and figures. The *name* is a string name of the thing being picked, such as **"segment"**. The *message* argument is used as the prompt. If this pointer is nil, then the routine constructs a prompt from the name by:

```
sprintf(promptbuf,"Pick a %s",name);
```

The *segmentp* and *itemp* arguments are used only for picking psurf items.

The funtion argument *picklister* generates a *pick list* using the IRIS GL routine **pick**. Models for this routine include **segmentpicklist** for picking segments, defined in the source code file **gen/src/lib/jack_segment.c++**, and **sitepicklist** for picking sites, defined in the source code file **gen/src/lib/jack_site.c++**. This function fills in an array of values that are *current*. The **Pick** routine then selects amoung them. The **namer** argument is used to convert a character string into a pointer to the object. It takes a character string name and returns a pointer. This is invoked by **Pick** when the user hits ^K and enters a string. Finally, the *highlighter* function highlights the object.

Chapter 4

Human Figures

4.1 The Human Data Structure

Although the principal purpose of *Jack* is to model human figures, the peabody data structure is designed to model arbitrary kinds of articulated figures, human figures being just one example. Some applications only apply to human figures, so it is convenient to be able to access the "human" parts of a figure. This is done with the *human* field in the figure structure. For human figures, this field points to a **Human** structure, which contains pointers to certain commonly used reference points on the body, as well as the set of constraints which control the body in its natural state.

```
typedef struct {
    struct {
        Site    *toes;
        Site    *heel;
        Site    *hand;
        Site    *eye;
        Joint   *shoulder;
        Joint   *elbow;
        Joint   *knee;
        struct {
            Constraint  *toes;
            Constraint  *heel;
            Constraint  *kneespring;
            Constraint  *elbowspring;
            Constraint  *eye;
        } constr;
        Vector  footbalancept;
    } left,right;
    Site    *lowertorso;
    Joint   *neck;
    Joint   *waist;
    Site    *headsight;
    Vector  balancepoint;
    Vector  eyefixation;
    struct {
        Constraint  *com;
        Constraint  *pelvis;
```

```
        Constraint  *spine;
        Constraint  *headsight;
    } constr;
    Spine   spine;
    struct {
        float           footbalanceinterpx;
        float           footbalanceinterpz;
            unsigned    torso       : 4;
            unsigned    pelvis      : 1;
            unsigned    root        : 1;
            unsigned    balance     : 2;
            unsigned    torsobalance: 1;
        struct {
            unsigned    hand        : 4;
            unsigned    foot        : 4;
            unsigned    heel        : 4;
            unsigned    footstepping: 1;
        } left,right;
    } behavior;
    int             support;
    unsigned        hasfingers      : 1;
    unsigned        balanced        : 1;
    float           leglength;
    Transform       pelvisoffset;
    Vector          lt,lh,rt,rh;
} Human;
```

The *left* and *right* fields hold pointers to the reference points on the left and right side of the body. The fields are as follows:

left.toes	The site `left_toes.distal`.
right.toes	The site `right_toes.distal`.
left.heel	The site `left_foot.distal`.
right.heel	The site `right_foot.distal`.
left.hand	The site `left_palm.palmcenter`.
right.hand	The site `right_palm.palmcenter`.
left.eye	The site `left_eye.sight`.
right.eye	The site `right_eye.sight`.
left.shoulder	The joint `left_shoulder`.
right.shoulder	The joint `right_shoulder`.
left.elbow	The joint `left_elbow`.
right.elbow	The joint `right_elbow`.
left.knee	The joint `left_knee`.
right.knee	The joint `right_knee`.
lower_torso	The site `lower_torso.proximal`.
neck	The single joint which contains the substring "neck".
waist	The single joint which contains the substring "waist"
headsight	A site on the head segment located between the eyes

The *headsight* is used for aiming the head in a particular direction and as a reference site for the view in a *Jack* eye window.

This structure is created with the function `InitHumanFigure`, which looks for the sites and joints with the proper names. If it can't find the appropriate sites and joints, it returns 0 and concludes that the figure is not a human figure. The function `InitHumanFigureConstraints` creates the constraints. If the *human* field of the figure structure is `NULL`, then the figure is not a human figure. These functions are invoked automatically from the commands in *Jack* which expect to operate on human figures. These functions are defined in the source code file `gen/src/lib/jack_figure.c++`.

4.1.1 Is it Human?

To determine whether or not a figure is a human figure, the function `InitHumanFigure` looks for a joint named `left_shoulder`. If it doesn't find one, then it assumes that the figure is not a human. Therefore, if you need for any reason to bypass these functions with a human-like figure, just rename the left shoulder.

4.2 The Human Figure Constraints

The constraints in the *constr* struct and in the two *constr* struct in the *left* and *right* structs are the *behavior* constraints on the human figure. They allow *Jack* to control the posture of the human figure. Each of these constraints has a goal type of `V_MATRIX`, except in the special cases outlined below.

The most intuitive analogy for how these constraints work is that of a marionette puppet, controlled by strings. The constraints are the strings. Each of these constraints defines a desired location in space for a reference point on the figure. Collectively, they specify the complete posture for the figure. The inverse kinematics algorithm finds a set of joint angles that places the figure in a posture that satisfies the positioning criteria given by the constraints.

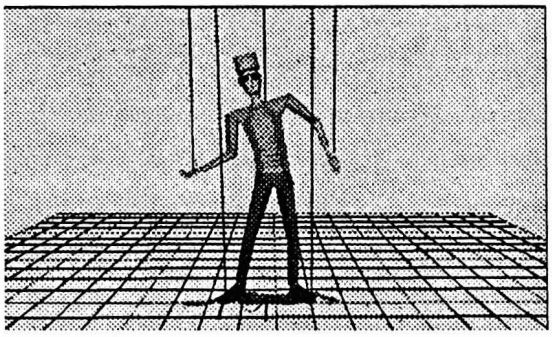

Figure 4.1: A Marionette Puppet

The user in *Jack* does not really see these constraints. They are created automatically by the function `InitHumanFigureConstraints`, described below. Only the effect of the constraints is visible. The user only sees a figure that can be manipulated in a variety of ways through the manipulation commands. The interface for manipulating human figures in *Jack* is designed to shield the user from the terminology of constraints. The user only sees manipulation commands like move foot and bend torso, and behavioral controls like `keep torso vertical`. Internally, the manipulation commands change the goal values of the constraints. The behavior control commands in *Jack* change the properties of these constraints.

There are only two ways through which the goal locations for these constraints may be set: through *Jack*'s manipulation commands, and through *Jack*'s *behavior functions*. The user does not interact with the human figure in *Jack* by adjusting joint angles or by moving the figure with the move figure command. The only access to the posture of the figure is through the manipulation commands, and these commands change the goals of the constraints. The behavior functions also determine positions for the goals.

In order to understand how these constraints work, try not to think of the figure as a jointed mechanism whose controls are the joint angles. Do not think of interacting with the figure by specifying joints angles. Instead, think of the figure as a puppet and think of manipulating it by moving the strings. The end of the strings specify the goals of the constraints. The inverse kinematics algorithm repeatedly computes the posture based on the goal values of the constraints.

4.2.1 Interactively Moving Constraints

The direct manipulation operator in *Jack*, `MoveTransform`, is described in Section 2.10. This function is object-oriented in the sense that it accepts functional arguments which define its behavior. It is used to interactively move positions and orientations in many different situations in *Jack*. It is the heart of the human figure manipulation commands.

The manipulation operator for constraints is the function `MoveConstraint`. This function invokes the manipulation operator `MoveTransform` with an action function, `updateconstr`, that applies the manipulation transform to the goal matrix of a constraint. Each of the human figure manipulation commands in *Jack* (except for bend torso) use this routine.

```
Boolean
MoveConstraint(Constraint *constr,char *prompt)
```

4.2.2 The Figure Root

The control of the posture of a human figure in *Jack* is complicated by the need to designate a single point on the figure as the root. The inverse kinematics algorithm cannot itself position the root. The inverse kinematics algorithm operates on chains of joints that emanate from the root. This means that if, for example, there is a constraint on the right toes when the figure is rooted through the right toes, then the constraint cannot have any effect on the position of the toes during the execution of the inverse kinematics algorithm. This means that the toes must be controlled through another technique.

Rather than designating a single point on the figure as the permanent figure root, *Jack* allows the root to change from time to time. The human figure behavior routines outlined below allow the posture of the figure to be described *completely* through the constraints. The behavior functions provide a means of automatically repositioning the figure root according to the constraints. The behavior functions also provide a way of changing the setting of the figure root in order to ensure that the figure "behaves" well.

The behavior functions described below make it possible to control the entire figure through its constraints, without worrying about the location of the figure root.

4.2.3 Human Figure Goals

4.2.4 getsitegoal

The goals of the constraints on the human figure serve as the handles for controlling the figure's posture. As subtle but significant aspect of the behavior functions is that the goals themselves serve to define the location of the appropriate body parts, rather than the actual location of the body part itself. For example, if a behavior function needs to know where the left toes are, it should consult the goal of the constraint on the left toes, not query the toe site itself. The reason for this is that the goal defines where the toes should be in the next iteration. The behavior functions are invoked before the constraints are evaluated but after the direct manipulation operator has moved one of the goals. Therefore, the behavior functions should consult the goals in order to define where the other goals should go.

This situation is complicated by the need to sometimes control parts of the figure through several constraints. This is particularly true with the animation system, in which motions controlling parts of the figure may overlap, meaning two conflicting constraints pull the body part in different directions. This is not a problem for the inverse kinematics algorithm because it simply weights each constraint and finds an minimum energy solution. But for behavior functions that need to know, for example, where the right toes are, it is not so simple if there are several constraints on the toes.

The function `GetSiteGoal` takes a site and returns with the goal of constraints on the site, i.e. constraints which have this site as an end effector. If there are multiple constraints which are currently active, then it averages the position and orientation of the goals, based on the goal weights and the position/orientation weights. It returns with a homogeneous transform giving the goal for the site.

```
GetSiteGoal(Site *site,Transform *goal)
```

This function is called in several key places:

- In `FindFootBalanceInterp` and `ComputeFootBalancePoint` to find the location of the toes for use in computing the balance point for the center of mass constraints.
- In `BalanceBehavior` to find the goal location of the site which is currently the figure root. This is defined in Section 4.4.

4.3 Human Figures and the Jack Simulation Procedure

Section 3.4 describes *Jack*'s simulation procedure. This consists of the function `AdvanceSimulation`, which is called inside of *Jack*'s interaction loops. It is defined in the file `src/lib/jack_time.c++`, and it is shown in Figure 3.4. `AdvanceSimulation` is called inside of the direct manipulation operator `MoveTransform`, inside the viewing changing command, and the top level command loop as well. `AdvanceSimulation` makes the sequence of calls:

```
ExecuteBehaviorFunctions();
EvaluateConstraints(Jack.c++onstr.timelimit,Jack.constr.stepfactor,NULL);
ExecutePostBehaviorFunctions();
```

This structure for the behavior functions is designed to be very general, but in fact it currently executes only the function `HumanBehaviors` on every human figure. This function, shown in Figure 4.2, is very simple.

```
ExecuteBehaviorFunctions()
{
    List l;
    Figure *figure;

    l = 0;
    while ((l=circlistiterator(env->humans,l))) {
        figure = LISTDATA(Figure,l);
        HumanBehaviors(figure);
    }
}
```

Figure 4.2: `ExecuteBehaviorFunctions`

The purpose of the behavior functions is the take the geometric information generated by the direct manipulation operator and make it known to the rest of the figure. This provides a sense of coordination between the parts of the figure. An example of this is the way the balance point follows the feet when the feet move. In one of the simplest cases, the constraint on the center of mass is automatically adjusted to a point half way between the figure's feet. This is described below in the discussion of the `BalanceBehavior` function. In this case, the *Jack* command move foot invokes the function `MoveConstraint` to interactively manipulate the goal of the constraint on the feet. The balance behavior function `BalanceBehavior` is responsible for interpolating between the goal locations for the feet to determine the proper location of the goal for the center of mass.

After executing the behavior functions, `AdvanceSimulation` evaluates the constraints. After this, it evaluates the function `ExecutePostBehaviorFunctions`. The *post-behavior functions* are simply behavior functions which need to be evaluated after the constraints rather than before. Currently, the only post-behavior function is `VerticalizeTorso`, the routine to keep the torso vertical.

4.3.1 The HumanBehaviors Function

The function **HumanBehaviors** is defined in the file **src/lib/human_behavior.c++**, and it is shown in Figure 4.3. It invokes behavior functions which perform operations on different parts of the figure. The most important ones are **BalanceBehavior**, shown in Figure 4.4, and **RootBehavior**, described in Section 4.3.1.2.

Jack uses the *behavior* field in the human structure to specify what the behavior functions are supposed to do. These flags generally correspond to the behavior controls described in the *Jack User's Guide*. Each of these fields has values defined in the include file **human.h**. The mapping between the character string names used by the user interface routines, and seen by the interactive user in *Jack*, is done with the **NamedType** structures defined in **src/lib/human_behavior.c++**.

There are two especially important fields of the human structure: *support* and *behavior.balance*. The *behavior.balance* flag registers whether the figure is sitting or standing, and if it is standing, how it is maintaining its balance. Some of the behavior functions operate differently in these two cases. The *behavior.balance* field is set with the function **SetBalanceControl**. The term "support" is really a euphemism for the figure root. The function **SetFigureSupport** sets the figure root and records it in the *support* field. The only options for the human figure are the left toes, the right toes, and the lower torso. A seated figure is always rooted through the lower torso. For a standing figure, the function **RootBehavior** determines the best location for the figure root.

4.3.1.1 The BalanceBehavior Function

The **BalanceBehavior** function, shown in Figure 4.4, computes a new location for the balance constraing based on the location of the feet. It uses the function **ComputeFootBalancePoint**, described in Section 4.4.1.

The function **BalanceBehavior** actually does more than govern balance, because it also serves for seated figure to transmit positional and orientational information from the center of mass and pelvis constraints.

When the figure is rooted through the left toes, the left toe constraint cannot do any positioning, and likewise for the right toes. The same applies to the lower torso and the center of mass and pelvis constraints. In this case, these constraints serve to describe a desired location for the figure root. The function **BalanceBehavior** looks at how the figure is rooted, and it moves the end effector of the constraint at the figure root to its goal.

This means that the posture of the figure can be described solely through the goal values of the constraints. No control mechanism needs to reference the figure root explicitly. The **RootBehavior** and **BalanceBehavior** functions ensure that the figure is rooted in the best possible way, and it automatically repositions the figure root according to the positioning criteria described by the constraints.

The function **BalanceBehavior** determines whether the balance point has moved very much. If it hasn't moved more than than a threshold amount, given by **Jack.balance.epsilon**, then it doesn't update the center of mass constraint. Otherwise, the center of mass constraint will be needlessly re-evaluated for a very small amount of positioning. Without this test, the constraints would always be evaluated, sometimes causing undesireable oscillations.

4.3.1.2 The RootBehavior Function

The **RootBehavior** function is responsible for determining the best site to use for the root of the figure, given the figure's current posture. It is not shown here because it is long, messy, and uniteresting. It is defined in the file **src/lib/human_behavior.c++**.

RootBehavior uses the following rules:

- 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.

- If the height of the center of mass above the feet dips below 70% of the length of the leg, then the root 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.

```
HumanBehaviors(Figure *figure)
{
    Human   *hum;

    hum = figure->human;

    FigureCenterOfMass(figure);

    /*
     * Compute the support polygon.  This is the convex hull of the projections
     * of the toes and heel of both feet.
     */
    SupportPoly(figure);

    /*
     * Do the balance behaviors.  If the figure is sitting, this
     * involves setting the figure location.
     */
    BalanceBehavior(figure);

    /*
     * Do the root behavior, but only if the figure is not seated.
     * If it's seated, the root never changes.
     */
    if (hum->behavior.balance != BC_SEATED) {
        if (hum->behavior.root) {
            RootBehavior(figure);
        }

        if (hum->behavior.pelvis) {
            PelvisBehavior(figure);
        }
        /*
         * Do the stepping behaviors, but only if the figure is not currently
         * stepping.  It can't take two steps at once.
         */
        if (hum->stepping == FALSE && !Jack.advancetime) {
            if (hum->behavior.step.balance && hum->constr.com->beingmoved) {
                BalanceSteppingBehavior(figure);
            }
            if (hum->behavior.step.pelvis && hum->constr.pelvis->beingmoved) {
                PelvisSteppingBehavior(figure);
            }
        }
    }
}
```

Figure 4.3: HumanBehaviors

```
BalanceBehavior(Figure *figure)
{
    Human       *hum;
    Vector      L,R,BL,bp;
    Transform   D,G,E,M;
    float       d;
    Transform   C,Ginv;

    hum = figure->human;

    if (!hum->seated && hum->behavior.balance == BC_FEET) {
        ComputeFootBalancePoint(figure,bp,L,R,BL);
        vecsub(D.v.p,bp,hum->balancepoint);
        d = MAG(D.v.p);

        if (Jack.advancetime || d > Jack.balance.epsilon) {
            GetSiteGlobal(figure->centerofmass,&G);
            cpvector(hum->balancepoint,bp);
            cpmatrix(E.matrix,hum->constr.com->goal.v.matrix);
            E.v.p[0] = hum->balancepoint[0];
            E.v.p[1] = G.v.p[1];
            E.v.p[2] = hum->balancepoint[2];
            SetHoldConstraint(hum->constr.com,&E);
        }
    }

    if (hum->support == SUP_LEFT_FOOT) {
        GetSiteGoal(hum->left.toes,&G);
        SetFigureLocation(figure,&G);
    } else if (hum->support == SUP_RIGHT_FOOT) {
        GetSiteGoal(hum->right.toes,&G);
        SetFigureLocation(figure,&G);
    } else if (hum->support == SUP_LOWER_TORSO) {
        GetSiteGoal(hum->lowertorso,&G);
        GetGoalTransform(hum->constr.pelvis,&E);
        if (hum->behavior.balance == BC_SEATED) {
            GetSiteGlobal(hum->constr.com->end.v.site,&M);
        } else {
            GetSiteGlobal(hum->constr.com->end.v.site,&E);
            cpvector(E.v.p,G.v.p);
            GetSiteGlobal(hum->constr.com->end.v.site,&M);
        }
    }

    /*
     * If the figure is standing, then set the xy location of the
     * heel constraints.  This only affects the drawing.  This will
     * make the constraints be drawn as a vertical line.
     */
    if (!hum->seated) {
        GetEndEffectorTransform(hum->left.constr.heel,&E);
        hum->left.constr.heel->goal.v.matrix[3][0] = E.v.p[0];
        hum->left.constr.heel->goal.v.matrix[3][2] = E.v.p[2];
        GetEndEffectorTransform(hum->right.constr.heel,&E);
        hum->right.constr.heel->goal.v.matrix[3][0] = E.v.p[0];
        hum->right.constr.heel->goal.v.matrix[3][2] = E.v.p[2];
    }

    GetEndEffectorTransform(hum->constr.pelvis,&E);
    cpvector(hum->constr.pelvis->goal.v.matrix[3],E.v.p);
}
```

Figure 4.4: BalanceBehavior

4.4 Human Figure Controls

The following functions set the behavioral parameters of the constraints:

```
SetFigureSupport(Figure *figure,int type)

SetTorsoControl(Figure *figure,int type)

Constraint *
SetHandControl(Figure *figure,int type,int side, int startjoint,Segment *segment)

Constraint *
SetFootControl(Figure *figure,int type,int side,Segment *segment)
```

`SetFigureSupport` accepts one of `SUP_LOWER_TORSO`, `SUP_LEFT_FOOT`, or `SUP_RIGHT_FOOT`, and sets the *support* field of the human structure accordingly. The is called by the `RootBehavior` function, and by the `SetBalanceControl` function when it makes the figure seated.

`SetTorsoControl` accepts one of `TC_NONE`, `TC_VERTICAL`, `TC_SETPARAM`, and `TC_HOLD`, and it sets *behavior.torso* accordingly, except that the special type `TC_SETPARAM` just sets the values of the torso parameters, i.e. the low, high, initiator, and resistor joints.

`SetPelvisControl` accepts one of `PC_HOLD` or `PC_FEET`, and it sets *behavior.torso* accordingly.

`SetHandControl` accepts one of `HC_RELEASE`, `HC_HIPS`, `HC_KNEES`, `HC_GLOBAL`, `HC_LOCAL`, or `HC_SITE`, and it sets either *behavior.left.hand* or *behavior.right.hand* accordingly. If the type is `HC_LOCAL`, then the *segment* argument specifies the segment to which the hand is to be constrained. In this case, the goal type of the constraint changes to a segment. If the type is `HC_SITE`, then the *segment* argument is the site, cast as a segment.

`SetFootControl` accepts one of `FC_RELEASE`, `FC_PIVOT`, `FC_LOCAL`, or `FC_GLOBAL`, and it sets either *behavior.left.foot* or *behavior.right.foot* accordingly. If the type is `FC_LOCAL`, then the *segment* argument specifies the segment to which the hand is to be constrained. In this case, the goal type of the constraint changes to a segment.

4.4.1 Balance Control

```
SetBalanceControl(Figure *figure,int type,float interpx,float interpz)
```

`SetBalanceControl` accepts one of `BC_FEET`, `BC_HOLD`, `BC_SEATED`, `BC_HOLD_ELEV`, or `BC_RELEASE_ELEV`, and it sets the *behavior.balance* field accordingly, except that `BC_HOLD_ELEV` and `BC_RELEASE_ELEV` are special cases and are *not* assigned to *behavior.balance*. They only cause a change in the position type of the center of mass constraint, between `C_POS` and `C_LINE`. In the case of `BC_FEET`, the extra *interpx* and *interpz* specify the parametrization of the location of center of mass goal with respect to the feet. If the `type` is not `BC_FEET`, then the extra parameters are ignored.

The two functions `ComputeFootBalancePoint` and `FindFootBalanceInterp` help out in the balance behavior. The parametrize the location of the center of mass in terms of the placement of the feet. Given the location of the center of mass, the function `FindFootBalanceInterp` computes two parameters, *footbalancinterpx* and *footbalancinterpz*, shown in Figure 4.5 as x and z. To do this, it projects the balance point on the $y = 0$ plane, which is the point $\hat{\mathbf{b}}$. It then finds the point on the balance line closest to this point, and calls it $\hat{\mathbf{p}}$. z is the distance between $\hat{\mathbf{b}}$ and $\hat{\mathbf{p}}$, that is, the balance point's distance forward from the balance line. Likewise, x is the

interpolation factor which gives $\hat{\mathbf{p}}$ in terms of the left and right foot reference points, normalized between 0.0 and 1.0, with $x = 0$ being the left foot. If x is outside of the $[0, 1]$, then the balance point is to the side of the support polygon. 40 40

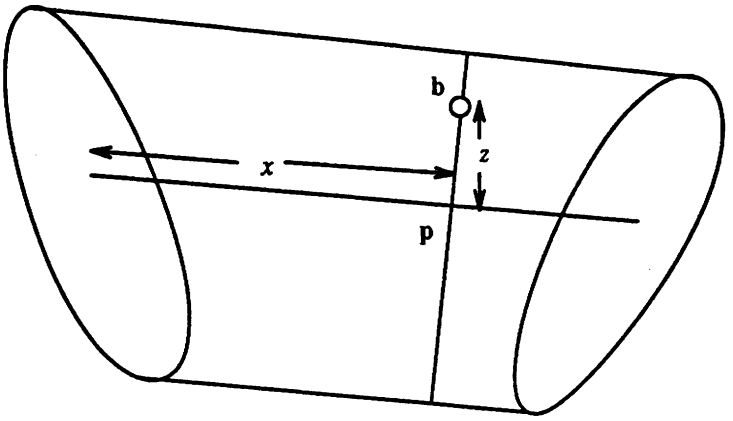

Figure 4.5: The Parametrization of the Balance Point

`ComputeFootBalancePoint` does the opposite: it takes the balance parameters and computes a new balance point based on the placement of the goals for the constraints on the feet.

```
ComputeFootBalancePoint(Figure *figure,Vector balancepoint,Vector L,Vector R,Vector D)

FindFootBalanceInterp(float *x,float *z,Figure *figure)
```

When the balance control is `BC_SEATED`, `SetBalanceControl` turns off the constraints on the toes and turns the heel constraints into position and orientation constraints. Then the move foot command will use the heel constraint as the end effector for moving the foot. This seems to work well for seated figures.

4.4.2 Stepping Behaviors

The `HumanBehaviors` function also calls the stepping behaviors functions `BalanceSteppingBehavior` and `PelvisSteppingBehavior`. These funtions look at the location of the center of mass and the orientation of the pelvis to see if the figure need to take a step. If it does, then they generate a foot motion, using the motion system described in Chapter 5. This motion causes the foot to "step" from its current location to a new location, during the course of the manipulation. `BalanceSteppingBehavior` need only be invoked while moving the center of mass, and `PelvisSteppingBehavior` need only be invoked while rotating the pelvis.

4.5 Inputting Human Figures

Human figures may be picked by the user with the function `InputHumanFigure`. If there is only one human figure in the environment, the function returns it without prompting the user to pick it.

```
Figure *
InputHumanFigure(arglist,outargs,prompt)
VALUE      **arglist;
char       **outargs;
char       *prompt;
```

4.5.1 The List of Humans

There is also a list of human figures in the **Environment** structure, called **env->humans**. This field is a list of **Figures** which have valid **Human** structures.

Chapter 5

The Motion System

5.1 The Motion Data Structure

A synopsis of the `Motion` data structure is:

```
typedef enum {
    M_NONE, M_ENDEFFECTOR, M_JOINT, M_SITE, M_LEFT_FOOT, M_RIGHT_FOOT, M_LEFT_HEEL,
    M_RIGHT_HEEL, M_COM, M_PELVIS, M_TORSO, M_LIGHT, M_SUPPORT, M_LEFT_HAND, M_RIGHT_HAND,
} MotionType;

struct motion {
    char        *name;
    char        *description;
    Figure      *figure;
    MotionType  mtype;
    long        starttime;
    long        duration;
    int         weightfunction;
    int         velocitycontrol;
    void        *mvar;
    int         (*alloc)();
    int         (*apply)();
    int         (*preaction)();
    int         (*postaction)();
    int         (*inputparams)();
    int         (*recordparams)();
    struct {
        unsigned    movetype :  3;
        unsigned    afterconstrs :  1;
        unsigned    current :  1;
    } flags;
    char    *jclprefix;
};
```

The motion data structure models a change in a parameter of a specific time interval. The motion simulation procedure in *Jack* simulates time over several possibly overlapping motions. The effect of each motion is described

through the set of function fields. The process of writing new motion primitives involves writing a new set of functions which cause the desired effects.

Motions are illustrated in the *Jack* animation window. This window draws motions in tracks, on a horizontal time line. The window is a port-hole into which the user can look at a set of motions which happen over a certain portion of time. The user can scroll the window forwards or backwards in time, or expand or shrink it to show a larger or smaller region of time. The window draws only the motions which happen during the time interval displayed in the window.

The animation window sorts the motions according to the figure and "body part" they control, so that the window gives the user a sense of how the controls on a part of the figure change over time. This sorting is done based on the *figure* field and the *mtype* field. This is currently the only application for the *mtype* field, since the behavior of the motion is defined in terms of its function fields.

The *name* field of the motion is the identifier that is placed in the left column of the animation window next to the figure's name. This usually describes the body part which the motion controls. The *description* field of the motion is the text string which goes along with the icon on the timeline in the window. This usually gives information about the parameters of the motion. The *figure* field points to the figure to which the motion applies.

The *starttime* and *duration* fields of the motion describe what time the motion is active. In the current implementation, these units are integral frame numbers, starting at 0. The ending time for the motion can be computed from the starting time and the duration.

5.1.1 Weight Functions

The *weightfunction* field is available for use by the functions which define the motions effects. In the currently implemented motions, the weight function is used to control the weight of the constraints which the motions control. The weight function can be one of:

WF_CONSTANT
WF_DECAY
WF_EASEIN_EASEOUT

The function `weightfunction` implements these functions, base in the input argument *type*. It maps a normalized time value *t*, in the range 0.0 to 1.0, to weight value between 0.0 and *weight*.

```
float
weightfunction(type,weight,t)
int    type;
float  weight;
float  t;
```

5.1.2 Velocity Controls

The *velocitycontrol* field is available for use by the functions which define the motions effects. In the currently implemented motions, it is used to control the velocity of the goals of the constraints which the motions control. The velocity can be one of:

VC_CONSTANT
VC_ACCELLERATE
VC_DECELERATE
VC_EASEIN_EASEOUT

The function `velocitycontrol` implements these functions, base in the input argument *type*. It maps a normalized time value *t*, in the range 0.0 to 1.0, to position interpolation value between 0.0 and 1.0. This value may then be used to interpolate between a starting and ending position by the motion function.

```
float
velocitycontrol(type,t)
int   type;
float  t;
```

5.1.3 The Motion Functions

The effects of the motion are controlled through the functions *preaction*, *apply* and *postaction*. In developing a new type of motion primitive, it is your responsibility to write the functions and have them generate the desired effect, according to the following rules. The *preaction* function is invoked once when the start time of the motion is reached. The *postaction* function is invoked once when the ending time for the motion is reached. The *apply* function is invoked at each time increment between the starting and ending time, inclusive. Each function is invoked with a pointer to the motion structure as an argument. The *apply* function is invoked with the current time as the second argument.

```
(*m->preaction)(m);

(*m->apply)(m,currenttime);

(*m->postaction)(m);
```

The *inputparams* and *recordparams* functions are responsible for inputting and recording the parameters of the motion, respectively. Each parameter of the motion should be input with `Input` functions inside of the *inputparams* function. This function is invoked with first two arguments to match the arguments of the `Input` functions. The third argument is the *mvar* field of the motion, and fourth is the a pointer to the motion itself.

```
inputparams(VALUE **arglist,char **outargs,void *mvar,Motion *m)
```

This function is used in two places. First of all, it is used to input the parameters when the motion is created. It is also invoked when the parameters of the motion are *changed*, using the command change motion. Therefore, the *inputparams* function should be structured so that it can be invoked multiple times. It should take the current value of the parameters as the default values.

The *inputparams* function should *not* input the start time and duration, because these parameters are generally changed graphically, by sliding the end points of the motion around with the mouse in the animation window.

The *recordparams* function records the parameters of the motion in a string in the format of JCL, so that the motion can be written to a JCL file. The *jclprefix* of the motion contains the name of the command used to create the motion. This function is invoked with a pointer to the motion structure as the first argument and the *outargs* argument second.

```
recordparams(Motion *m,char *outargs)
```

This function should use the `RecordArgument` functions to record the parameters.

5.2 An Example Motion

This section describes an example motion, taken directly from the source code file `src/lib/human_motion.c++`. This is the implementation of the create pelvis motion command. This is a good generic example of how motions work in *Jack*.

The definition of the motion includes the functions `pelvis_preaction`, `pelvis_postaction`, `pelvis_apply`, `pelvis_inputparams`, `pelvis_recordparams`, and `CMDCreatePelvisMotion`, along with a special structure called

PelvisMotion, which hold information needed by the motion functions. In this case, the necessary information includes the constraint on the pelvis, the starting and ending pelvis orientations (stored as homogeneous transforms), and the constraint weight.

There are some important point to consider:

- Recall that a motion is an instruction to move something *to* a desired location, beginning at a certain time, *from* its current location at the starting time. Therefore, the *start* field of the **PelvisMotion** structure is filled in automatically by the preaction function. This sets the starting orientation of the pelvis to its value when the preaction function is executed. This happens only at the starting time of the motion.

- The postaction function deactivates the motion constraint and activates the human body constraint, initializing it to hold the pelvis in its final orientation.

- The apply function interpolates between the starting and ending position, according to the current time, passed in as a parameter. In this case, the apply function linearly interpolates between the two orientations, using the function **linterpmatrix**. Notice how it uses the **velocitycontrol** function to map the time into the interpolation parameter. It also computes the weight using the **weightfunction**.

- The apply function is responsible for turning off the human body constraint on the pelvis and turning on the **PelvisMotion**'s constraint. The human body constraint should not be active while the motion is in effect. This *must* be done in the apply function, not the preaction function, becaue motions like this can overlap. It is possible to have two motions controlling the same part of the body at the same time. Since the postaction function activates the body constraint to hold the pelvis in its current orientation after the motion passes, if two motions are overlapping and one expires before the other, the body constraint will be activated while the long motion is still in effect. Therefore, the apply function must turn off the body constraint at each iteration to ensure that it is off.

  This is true of all motions which control parts of the human body in this manner.

- The inputparams function calls the manipulation command **RotatePelvis** so that the user can manipulate the pelvis into the desired orientation. After this, it gets the final value of the goal of the pelvis constraint and saves that as the ending orientation for the motion.

- It is absolutely essential that the inputparams function deal properly with the JCL arguments, through the *arglist* and *outargs* arguments, since JCL is the way in which motions are stored in files and later retrieved. In this case, the function **RotatePelvis** does this. In other circumstances, it may be necessary to parse the arguments explicitly.

- The recordparams function uses the **RecordArgument** functions to generate a JCL representation of the arguments of the motion command. It is abolutely essential that the arguments match in number, type, and, order of occurrence between **inputparams** and in **recordparams**.

- The *Jack* command function **CMDCreatePelvisMotion** is responsible for allocating the **PelvisMotion** structure an for creating the constraint and setting its parameters. This is done only once. This constraint should be off initially. It should also have its *motion* field set to indicate that it belongs to a motion.

5.3 Creating Motions

Motions may be create with the function **NewMotion**.

```
Motion *
NewMotion()
```

```
typedef struct {
    Constraint  *constr;
    Transform   start,end;
    float       weight;
} PelvisMotion;

pelvis_preaction(m)
Motion      *m;
{
    PelvisMotion    *pm;
    Human           *hum;

    pm = (PelvisMotion *) m->mvar.data;
    hum = m->figure->human;

    GetSiteGlobal(hum->lowertorso,&pm->start);
}

pelvis_postaction(m)
Motion      *m;
{
    PelvisMotion    *pm;
    Human           *hum;

    pm = (PelvisMotion *) m->mvar.data;
    hum = m->figure->human;

    SetConstraintStatus(pm->constr,FALSE);
    SetHoldConstraint(hum->constr.pelvis,NULL);
    SetConstraintStatus(hum->constr.pelvis,TRUE);
}

pelvis_apply(m,t)
Motion      *m;
long        t;
{
    PelvisMotion    *pm;
    Human           *hum;
    float           s,f;
    Transform       H;

    pm = (PelvisMotion *) m->mvar.data;
    hum = m->figure->human;

    SetConstraintStatus(pm->constr,TRUE);
    SetConstraintStatus(hum->constr.pelvis,FALSE);

    s = (t-m->starttime) / (float)(m->duration - 1);
    f = velocitycontrol(m->velocitycontrol,s);

    linterpmatrix(H.matrix,pm->start.matrix,pm->end.matrix,f);
    SetHoldConstraint(pm->constr,&H);

    pm->constr->weight = weightfunction(m->weightfunction,pm->weight,s);
}
```

Figure 5.1: PelvisMotion, Part I

```
pelvis_inputparams(arglist,outargs,pm,m)
VALUE       **arglist;
char        **outargs;
PelvisMotion*pm;
Motion      *m;
{
    Human           *hum;
    float           angle;
    char            buf[80];
    int             n;

    hum = m->figure->human;

    if (RotatePelvis(arglist,outargs,m->figure)) {
        GetGoal(hum->constr.pelvis,&pm->end);
    }

    if (!InputFloat(arglist,outargs,&pm->weight,"weight:  ")) {
        return(0);
    }
    if (!InputWeightFunction(arglist,outargs,m)) {
        return(0);
    }

    if (!InputVelocityControl(arglist,outargs,m)) {
        return(0);
    }

    sprintf(buf,"[%.1f]",pm->weight);
    if (m->description) {
        free(m->description);
    }
    m->description = strdup(buf);

    return(1);
}

pelvis_recordparams(m,args)
Motion      *m;
char        **args;
{
    PelvisMotion*pm;

    pm = (PelvisMotion *) m->mvar.data;

    RecordArgumentTime(args,m->starttime);
    RecordArgumentTime(args,m->starttime+m->duration);

    RecordArgumentTransform(args,&pm->end);
    RecordArgumentFloat(args,pm->weight);
    RecordArgumentString(args,TypeName(m->weightfunction,weight_functions));
    RecordArgumentString(args,TypeName(m->velocitycontrol,velocity_controls));
}
```

Figure 5.2: Pelvis Motion, Part II

```
CMDCreatePelvisMotion(args,outargs)
VALUE       *args;
char        **outargs;
{
    Figure          *figure;
    int             duration;
    Motion          *m;
    PelvisMotion    *pm;
    Vector          D;
    Human           *hum;
    char            buf[80];

    if (!(figure=InputHumanFigure(&args,outargs,0))) {
        return(0);
    }

    if (!(hum=InitHumanFigureConstraints(figure))) {
        return(0);
    }

    m = NewMotion(0);
    m->figure = figure;
    m->mtype = M_PELVIS;
    m->jclprefix = strdup(Jack.currentargs);
    m->name = strdup("pelvis");

    if (!InputStartTime(&args,outargs,m)) {
        goto abort;
    }

    if (!InputEndTime(&args,outargs,m)) {
        goto abort;
    }

    pm = talloc(PelvisMotion,1);
    m->mvar.data = (void *) pm;

    pm->weight = 5.0;

    pm->constr = CreateHoldConstraint(hum->lowertorso,"pelvis");
    pm->constr->motion = TRUE;
    pm->constr->off = TRUE;
    pm->constr->ptype = C_NONE;
    pm->constr->poweight = 0.0;
    m->weightfunction = WF_DECAY;

    m->apply = pelvis_apply;
    m->preaction = pelvis_preaction;
    m->postaction = pelvis_postaction;
    m->inputparams = pelvis_inputparams;
    m->recordparams = pelvis_recordparams;

    if (!pelvis_inputparams(&args,outargs,pm,m)) {
        goto abort;
    }

    Jack.resortmotions = TRUE;
    CreateAnimationWindow();
    return(1);

    abort:
    DeleteMotion(m);
    return(0);
}
```

Figure 5.3: Pelvis Motion, Part III

This function **NewMotion** allocates memory for the motion structure and fills in default values for some of the parameters, but it doesn't do anything else. You must fill in the fields explicity after allocating it. The process for doing this is described below.

The following functions are useful in inputing parameters of motions:

```
int
InputWeightFunction(arglist,outargs,m)
VALUE   **arglist;
char    **outargs;
Motion  *m;


int
InputVelocityControl(arglist,outargs,m)
VALUE   **arglist;
char    **outargs;
Motion  *m;


int
InputSide(arglist,outargs,side)
VALUE  **arglist;
char   **outargs;
int    *side;
```

InputSide selects the keywords **right** or **left**, for motions that apply to a part of the body on the right or left side, like hands or feed. It returns 0 for left, 1 for right.

5.4 How the Animation System Works

The animation system works through the simulation procedure described in Section 3.4. The simulation procedure **AdvanceSimulation** invokes the function **AdvanceTime**, provided the **Jack.advancetime** flag is set. This function advances the current time indicator **Jack.currenttime**. The **Jack.advancetime** provides a means of turning "time" off and on. As described below, the function **AdvanceTime** resets this flag when it detects that nothing else is to occur in the future. There are other functions for setting the time counter backwards, or to an arbitrary point. The function **AdvanceTime** is shown in Figure 5.4.

Jack's notion of time is currently an integer corresponding to frames numbers, i.e. 30 time ticks per second. However, this notion of time is quite arbitrary until a sequence of frames is put onto videotape, because *Jack* usually cannot generate the animation at 30 frames per second. What you see interactively is much slower.

This function invokes the function **UpdateActiveMotions** which examines each motion to determine if it is "active" at the current time. The term *active* here means that **Jack.currenttime** is between the motion's starttime and ending time, so the motion is currently in effect. It sets the motion's *flags.active* field accordingly. Thus, this flag designates which motions are currently in effect. In addition, **UpdateActiveMotions** executes the *preaction* functions for the motions whose *starttime* coincides with the current time, and it executes the *postactions* for the motions whose ending time is equal to the current time.

AdvanceTime then invokes **ApplyActiveMotions** to execute the *apply* functions for the currently active motions.

AdvanceTime then evalutes the constraints. The constraint evaluation process automatically considers only constraints which are currently "on," as described in Section 6.4.1. The process here involves handling the special cases of the center of mass constraints. This process is done in a loop to ensure that First, the procedure computes the center of mass of each human figure and stores it in the the *human-¿balancepoint* field. It then

```
AdvanceTime()
{
    Figure      *human;
    List        l;
    int         niter;
    Transform   G;
    int         done;
    Vector      v;
    float       d;
    Boolean     more;

    Jack.currenttime++;

    /*
     * Execute the appropriate preaction, postaction, and apply functions
     * for the current time.  If this is a motion's starttime, execute its
     * preaction.  If this is a motion's ending time, execute its postaction.
     */
    more = UpdateActiveMotions();
    ApplyActiveMotions();
    ExecuteBehaviorFunctions();

    /*
     * Evaluate the constraints in a loop that terminates when the
     * centers' of mass converge.
     */
    niter = 0;
    do {
        l = 0;
        while ((l=circlistiterator(env->humans,l))) {
            human = LISTDATA(Figure,l);
            FigureCenterOfMass(human);
            GetSiteGlobal(human->centerofmass,&G);
            cpvector(human->human->balancepoint,G.v.p);
        }

        EvaluateConstraints(0,0.0,NULL);

        /*
         * Get each human's center of mass again and compare it to its
         * previous location.  If any center of mass moved more than
         * 'threshold', we're not done yet.
         */
        done = TRUE;
        l = 0;
        while ((l=circlistiterator(env->humans,l))) {
            human = LISTDATA(Figure,l);
            FigureCenterOfMass(human);
            GetSiteGlobal(human->centerofmass,&G);
            vecsub(v,human->human->balancepoint,G.v.p);
            d = MAG(v);
            if (d > Jack.balance.threshold) {
                done = FALSE;
            }
            cpvector(human->human->balancepoint,G.v.p);
        }
    } while (!done && niter++ < Jack.nbalanceiterations);

    ExecutePostBehaviorFunctions();

    SaveFrame(Jack.currenttime);

    return(more);
}
```

Figure 5.4: AdvanceTime

```
UpdateActiveMotions()
{
    List    l;
    long    endtime;
    Motion  *m;
    Boolean moremotions;

    moremotions = FALSE;

    l = 0;
    while ((l = circlistiterator(Jack.motions,l))) {
        m = LISTDATA(Motion, l);
        endtime = m->starttime + m->duration;

        if (endtime > Jack.currenttime) {
            moremotions = TRUE;
        }

        if (m->starttime == Jack.currenttime && m->preaction) {
            (*m->preaction)(m);
        }

        if (endtime == Jack.currenttime && m->postaction) {
            (*m->postaction)(m);
            m->flags.active = FALSE;
        }

        if (m->starttime <= Jack.currenttime && Jack.currenttime < endtime) {
            m->flags.active = TRUE;
        } else {
            m->flags.active = FALSE;
        }
    }
    return(moremotions);
}
```

Figure 5.5: `UpdateActiveMotions`

```
ApplyActiveMotions()
{
    List    l;
    Motion  *m;

    l = 0;
    while ((l = circlistiterator(Jack.motions,l))) {
        m = LISTDATA(Motion, l);
        if (m->flags.active && !m->flags.afterconstrs && m->apply) {
            (*m->apply)(m,Jack.currenttime);
        }
    }
}
```

Figure 5.6: `ApplyActiveMotions`

```
ExecutePostActions()
{
    List    l;
    Motion  *m;

    l = 0;
    while ((l = circlistiterator(Jack.motions,l))) {
        m = LISTDATA(Motion, l);
        if (m->flags.active) {
            if (m->postaction) {
                (*m->postaction)(m);
            }
            m->flags.active = FALSE;
        }
    }
}
```

Figure 5.7: `ExecutePostActions`

5.5 Frames

Jack records frames in terms of joint angles and figure locations in a **Frame** data structure, defined in the include file `frame.h`. Frames can be saved with `SaveFrame` and set with `SetFrame`.

```
Frame *frame
SaveFrame(int id)


SetFrame(Frame *frame)
```

`SaveFrame` takes an *id* argument which is the time indicator for the frame. If *id* is greater than or equal to 0, then the frame is stored in a list of frames which can be played back with the command play frames and recorded to the video disk with record frames to vdisk. If *id* is negative, then the frame is not added to the list of frames. These frames can be used to store other kinds of postures.

The **Frame** structure also holds information about the goals of constraints which are active for that frame. The function `SetFrame` sets the joint angles, figure root, figure location, and constraint goals associated with that frame.

5.6 Controlling Time

```
SetCurrentTime(long t)
```

The function `SetCurrentTime` sets *Jack*'s current time. It also sets the frame to be that tie, if such a frame exists. Because this might be called while motions are active, and because it may be moving the current time into an interval in which other motions are active, it is important to execute the motion postactions before setting the time, and the moiton preactions after it has been set. `SetCurrentTime` is shown in Figure 5.8.

```
void
SetCurrentTime(long t)
{
    int     f;

    if ((f=FindFrame(t))) {
        SetFrame(f,TRUE);
    }

    ExecutePostActions();

    Jack.currenttime = t;

    ExecutePreActions();

}
```

Figure 5.8: `SetCurrentTime`

Chapter 6

Constraints

6.1 Constraints

This section describes the implementation of constraints. This assumes that you are already familiar with the terminology of constraints, as described in the Jack User's Guide, including *end effectors*, *goals*, *starting joints*, *objective types*, and *weights*.

Constraints are regarded as an integral part of the peabody environment, so many of the routines for dealing with them are distributed throughout the source code into such files as `src/lib/pea_new.c++`, `src/lib/pea_delete.c++`, etc. In addition, the files `src/lib/pea_constraint.c++` includes many of the low-level routines described below. The higher-level functions having to do with constraint evaluation as a part of *Jack* are defined in `src/lib/jack_constraint.c++`. The constraint solver `SolveConstraints` is defined in the file `src/lib/pea_solve.c++`.

The most important aspect of constraints is now they are evaluated. This is done by the inverse kinematics algorithm, commonly referred to as "Jianmin's Reach Algorithm." This is discussed in Section 6.6.

6.2 The Constraint Data Structure

A synopsis of the constraint data structure is:

```
typedef struct constraint Constriant;

struct constraint {
    char            *name;
    ObjectiveType   ptype;
    ObjectiveType   otype;
    float           poweight;
    float           weight;
    VALUE           end;
    VALUE           goal;
    Joint           *startjoint;
    Transform       displacement;
    int             priority;
    List            joints;
    PosParam        pos;
    OrientParam     orient;
    float           distance;
    unsigned        rooting :  1;
```

```
    unsigned        off :  1;
    unsigned        hasdisplacement :  1;
    unsigned        justcreated :  1;
};
```

The *ptype* and *otype* fields are of type `ObjectiveType`, which is defined as:

```
typedef enum {
    C_UNDEF, C_NONE,
    C_POS, C_LINE, C_PLANE, C_EDGE, C_FACE,
    C_FRAME, C_AIM, C_DIR, C_VIEW, C_PLANEDIR,
    C_RESTANGLE, C_JOINTLIMIT,
} ObjectiveType;
```

The *poweight* and *weight* fields are the position/orientation weight and the constraint weight, respectively. The *end* and *goal* fields are `VALUE` structures representing the end effector and goal, respectively. The end effector may be a `V_SITE` or a `V_NODE`. The goal may be a `V_SITE`, a `V_NODE`, a `V_FACE`, or a `V_MATRIX`. The *startjoint* field is the starting joint. The *displacement* field is a matrix offset which describes the placement of the "true" goal used for the inverse kinematics routine relative to the goal as specified in the *goal* field. This allows goals to be attached to objects at points or orientations other than that associated with sites, nodes, or faces. If this value is used, then the field *hasdisplacement* must be set. Otherwise, the *displacement* field is ignored.

The *off* flag determines whether the constraint is to be evaluated along with the others. If this flag is set, the constraint is not evaluated.

Some of the fields in the constraint data structure are for internal bookkeeping. The *joints* field is the constraint's list of joints, generated by the function `ConstraintJoints`. It includes all joints between the end effector and the starting joint which have degrees of freedom and are not frozen.

The *pos* structure stores the position component of the goal, in both local and global coordinates. The `PosParam` structure is defined as:

```
typedef union {
    struct {
        Vector      P;
    } point;
    struct {Vector      v;
        Vector      V;
        Vector      P;
    } line;
    struct {
        Vector      n;
        Vector      N;
```

```
        Vector      P;
    } plane;
    struct {
        Vector      normal;
        int         nvertices;
        Vector      *vertices;
        Vector4     *eqns;
    } face;
} PosParam;
```

Actually, it's a union, with a structure corresponding to each position objective type. The upper case fields are position components resolved to global coordinates. They are filled in internally by the peabody routine `InitializeGoal`. The lower case fields are parameters of the constraint which must be set. The *line.v* field is the direction of the line for a line constraint, specified in the coordinate frame of the goal. The *plane.n* field is the direction of the plane for a plane constraint, specified in the coordinate frame of the goal. The face goal stores the vertices of the polygon of the face, and a set of equations used for testing whether a point is inside the face.

The *orient* structure stores the orientation component of the goal, in both local and global coordinates. The `OrientParam` structure is defined as:

```
typedef union {
    struct {
        Vector      v;
    } aim;
    struct {
        Vector      e;
        Vector      v;
        Vector      V;
    } dir;
    struct {
        Vector      e;
        Vector      v;
        Vector      V;
    } planedir;
    struct {
        Vector      X;
        Vector      Y;
    } frame;
    struct {
        Vector      Z;
        Vector      H;
    } view;
} OrientParam;
```

Actually, it's a union, too, with a structure corresponding to each position objective type. The upper case fields are in global coordinates, and the lower case fields are in local coordinates. These lower case fields are the ones which you must supply as parameters when creating a constraint.

The *aim.v* field gives the viewing vector in the coordinate frame of the end effector. The *dir.e* field gives the direction vector on the end effector, and the *dir.v* field gives the direction vector on the goal, both for a direction constraint. The *planedir* fields are similar. The *frame* and `view` fields do not have parameter values because they take their orientation directly from the orientation of the goal.

6.3 Creating Constraints

The process of creating constraints is rather tricky because of the way they are evaluated. Constraints are created with the function `NewConstriant`, but follow the instructions below.

```
Constraint *
NewConstraint(char *name)


List
ConstraintJoints(Constraint *constr,Joint *startjoint)


AddConstraintToGlobalList(constr)
Constraint  *constr;
```

The function `ConstraintJoints` sets the starting joint of a constraint. The return value of the function is the list of joints for the constraint. It's not necessary to do anything with this list. It is stored in the constraint in the field *joints*. It is acceptable to reference this field for information, but you should not assign directly to it.

Jack keeps constraints in the global list `env->constraints`. This list is the source of constraints that *Jack* evaluates automatically. The function `NewConstraints` does *not* add the constraint to this list. The reason for this is that the list of constraints is repeatedly evaluated in *Jack*, and the constraint is not valid until all of its fields are filled in — they aren't filled in with valid values by `NewConstraint` itself. The function `AddConstraintToGlobalList` does this. Therefore, to create a new constraint:

1. Create the data structure itself with `NewConstraint`.
2. Fill in the essential fields. These are:

   end The end effector. Set both the type and the value.

   goal The goal. Set both the type and the value.

   ptype The position objective type. If this value is `C_LINE`, then the *pos.line.v* field must be filled in. If this value is `C_PLANE`, then the *pos.plane.n* field must be filled in.

   otype The orientation objective type. If this value is `C_AIM`, then the *orient.aim.v* field must be filled in. If this value is `C_DIR`, then the *orient.dir.e* and *orient.dir.v* fields must be filled in.

   poweight The position/orientation weight.

   startjoint The starting joint. Set this with the function `ConstraintJoints`.

3. Add the constraint to the global list with, you guessed it, `AddConstraintToGlobalList`

6.3.1 More on Creating Constraints

```
Constraint *
CreateHoldConstraint(Site *site,char *name)


Constraint *
SetPreferredAngle(Joint *joint,float angles[],Constraint *constr)


Constraint *
SetJointLimitSpring(Joint *joint,float exp[],Constraint *constr)
```

The function `CreateHoldConstraint` creates a constraint whose goal type is `V_MATRIX` and whose end effector is a site, specified by the argument *site*. This is very common in *Jack*. The *name* argument gives a name to the constraint.

The function `SetPreferredAngle` creates a constraint whose type is a joint spring. The *angles* argument gives the angle to which the joint will spring. If the *constr* argument is not null, then no new constraint is created. Instead, this constraint is transformed into the joint spring constraint.

The function `SetJointLimitSpring` creates a constraint whose type is a joint limit spring. Human figures in *Jack* have these constraints on the elbows and knees to discourage them from becoming locked. The objective function for type of constraint adds an exponential amount of potential energy as the angle reaches the limit.

6.4 Controlling Constraints

In the original design of the inverse kinematics system, *Jack* required that end effectors and goals be sites. This proved to be be cumbersome, so the types of end effector and goal fields were expanded to include other data types. Now, the most common type of goal is a `V_MATRIX`, which is sometimes called a ***hold*** constraint because it causes the end effector to hold its current global position and/or orientation. *Jack* still uses the term ***hold constraint*** to mean a constraint which is not attached to an object. The function `SetHoldConstraint` sets the value of the goal matrix for such a constraint. If the transform T is nil, then the function uses the current placement of the end effector as the goal. `SetHoldConstraint` automatically eliminates the displacement of a constraint.

```
void
SetHoldConstraint(Constraint *constr,Transform *T)


void
SetActiveHoldConstraints(list)
```

Because of the way in which *Jack* determines whether a constraint set needs to be solved, it is essential that you use `SetHoldConstraint` to fill in the value of the matrix. It serves the dual purpose of actually filling in the value and notifying the constraint set that it should be evaluated again.

The function `SetActiveHoldConstraints` invokes `SetHoldConstraint` with a nil transform on each hold constraint which is currently on. This sets the goals for all hold constraints to be the current location of their end effectors.

Constraints can also have displacements, which are offsets between the goal as represented in the *goal* field of the constraint and the actual location of the goal as seen by the inverse kinematics algorithm. An example

of the need for this is when an end effector needs to be constrained to the position of a site but with a different orientation. The difference in orientation can be maintained in the displacement.

The function `SetConstraintDisplacement` sets the displacement. If the transform T is nil, then the function resets the displacement to the identity transform. `SetHoldConstraint` automatically eliminates the displacement of a constraint.

```
void
SetConstraintDisplacement(Constraint *constr,Transform *T)
```

6.4.1 Turning Constraints On and Off

Once constraints are created, they can be activated and deactivated with the function `SetConstraintStatus`. Turning a constraint off makes it transparent, as it if doesn't exist. However, it can be turned back on again later. This function changes the *off* field of the constraint, but it is essential that you not change this field except through the use of this function.

```
int
SetConstraintStatus(Constraint *constr,Boolean on)
```

The *on* argument tells whether the constraint is on (`TRUE`) or off (`FALSE`).

Alternatively, the constraint evaluation process can be disabled altogether with the flag `Jack.constr.on`. If this flag is not `TRUE`, then *Jack* skips the constraint evaluation procedure altogether.

6.4.2 Constraint Priority

Constraints can have a *priority*, which determines whether they should be evaluated collectively with other constraints. If two constraints have different priorities, *Jack* will evaluate the one with lower priority first, then the one with higher priority, even if they affect the same joints. Normally, all constraints have priority of 0.

```
int
SetConstraintPriority(Constraint *constr,int priority)
```

The constraints on the arms of human figures in *Jack* have priority 1.

6.5 Getting Information about Constraints

```
Segment *
EndEffectorSegment(Constraint *constr)


Segment *
GoalSegment(Constraint *constr)
```

```
int
GetEndEffectorTransform(Constraint *constr,Transform *E)


int
GetGoalPoint(Constraint *constr,Vector p)


int
GetGoalTransform(Constraint *constr,Transform *G)
```

Constraint goals and end effectors are abstract data types: the values of their fields can be of several different types. These functions return homogeneous transforms which describe the location of the end effector and the goal of a constraint. `EndEffectorSegment` returns the segment to which an end effector belongs. `GoalSegment` does the same for a goal, except that it will return nil if the goal does not belong to a segment, as in the case of a `V_MATRIX` goal, that is, a hold constraint. `GetGoalPoint` returns the vector at the origin of a goal. `GetGoalTransform` returns the transform describing the global coordinate frame of the goal, except that if the goal is a node or a face, then the transform is the transform of the segment, so it is necessary to call `GetGoalPoint` in order to find the actual global position.

`GetGoalTransform` automatically incorporates the constraint displacement into the returned value, if there is a displacement.

6.6 The Constraint Evaluation Process

Normally, the evaluation of constraints takes place in the *Jack* control structure, so it is not necessary to invoke the evaluation functions explicitly in other routines. The explanation given here provides background on how *Jack* performs the process.

Before *Jack* evalutes constraints, it divides them into independent sets. The constraint solution procedure `SolveConstraint` operates on constraint sets. For the most part, this is for efficiency, since the inverse kinematics algorithm will operate more efficiently when invoked twice with two small sets of constraints than it will if invokded once with a larger set. It is also efficient because some information maintained in the constraint set data structure can be reused from one evaluation to next. In addition, however, constraints can have a priority, which gives an order in which the constraints should be evaluated. This is sometimes beneficial when overlapping constraints should not affect one another. The mechanism for dividing constraints into independent sets examines the priority and groups constraints of similar priority into the same set.

The master constraint evaluation procedure is `EvaluteConstraints`, defined in the file `src/lib/jack_constraint.c++`. If the flag `Jack.constr.on` is not `TRUE`, this function returns immediately, so no constraints will be evaluted. `EvaluteConstraints` begins by generating constraint sets if necessary. *Jack* keeps the constraint sets in the global list `env->constraintsets`, and it uses this list to determine whether the sets need to be generated. If the list is non-nil, then the sets exist and should be evaluted. If the list is nil, then the constraint sets need to be generated, using the function `OrderConstraints`. This function computes the constraint sets and places them in the list `env->constraintsets`. This function traverses the peabody tree to find constraints which are completely independent of each other to place them in separate sets.

The function `OrderConstraints` arrives at lists of constraints which are dependent upon one another (Two constraints are dependent upon one another if either one contains any joints which affect the segments in the other's joint chain). It generates a constraint set data structure with each of these lists with the function `GenConstraintSet`. This function operates on a list of constraints, and it returns a pointer to the new constraint set data structure.

Before solving the constraints, `EvaluateConstraints` invokes the function `InitializeGoal` on each constraint. This function resolves the local coordinates in the goal and end effector structures to global coordinates. The constraint solver `SolveConstraints` needs information about the goal in global coordinates. It uses only the information in the *pos* structure in the constraint. `InitializeGoal` fills this in. `InitializeGoal` is also responsible for implementing the *step factor* feature, whereby the goals are not allowed to be too far away from the end

effectors. It measures the distance from the end effectors to their goals, and if this is beyond the allowable threshold, which is an argument to `EvaluateConstraints`, then it interpolates between the two and fills the interpolated data into the constraint's *pos* structure.

After initializing the goals, `EvaluteConstraints` then invokes `SolveConstraints` on each constraint set. This is the actual inverse kinematics algorithm. `SolveConstraints` also accepts a constraint time limit, and *Jack* keeps this data in the constraint set data structure. `EvaluateConstraints` attempts to balance the time limit between the constraint sets so that constraint sets which need more time get more time.

`SolveConstraints` will invoke the inverse kinematics algorithm on a constraint set only if the set is *out of date*, which means that something has moved since the last time it was evaluated. This avoids the overhead of invoking the inverse kinematics algorithm when no positioning needs to be done. The constraint set data structure has a time stamp that is set whenever it is evaluated. `SolveConstraints` calls the function `ConstraintSetOutOfDate` to determine whether any of the joints in any of the constraints has changed since the last evaluation. It also checks to see whether the goals of the constraints have moved. For goals which are sites, it compares the *time_of_update* setting of the site. For hold constraints, it uses the constraint's *time_of_update* field.

6.7 Evaluating Your Own Constraints

The process described above allows *Jack* to maintain a set of constraints which describe desired geometric relationships and to evaluate them in the presents of movement, coming either from the direct manipulation operator or from the motion system. However, it is sometimes convenient to use the inverse kinematics algorithm as a subroutine in some other operation. In this case, you must create constraints and evaluate them in a context which is not visible to the rest of *Jack*. This is easy to do using the procedure described here.

A good example of this type of operation is the function `PositionChain`. This function is shown in Figure 6.1. It accepts a site, a joint, and a homogeneous transform, and it uses the inverse kinematics algorithm to position the figure in such a way that the given site is located at the given transform, in both position and orientation. Only the joints between the given joint and the site will move.

```
PositionChain(Site *site,Joint *joint,Transform *T)
{
    Constraint  *constr;
    List        list;
    ConstraintSet*cs;
    float       f;
    int         drawreach();

    constr = NewConstraint(0);
    constr->end.type = V_SITE;
    constr->end.v.site = site;
    constr->goal.type = V_MATRIX;
    SetHoldConstraint(constr,T);
    ConstraintJoints(constr,joint);
    InitializeGoal(constr,0.0);

    list = 0;
    appendcirclist(&list,constr);
    cs = GenConstraintSet(list);
    f = SolveConstraints(cs,drawer,0);

    DeleteConstraint(constr);
    killcirclist(&list,0);
}
```

Figure 6.1: `PositionChain`

This function may do what you need, in which case you can just use it as is. If you need something slightly different, then just use it as a guide. This constraint is evaluated in isolation of the other constraints in the environment. The important points in the process are:

- Create the constraint with `NewConstraint`. Don't bother giving it a name.
- Set the types and values of the end effector and goal.
- Set the objective function. `PositionChain` doesn't do this because it uses the defaults: position and orientation, with a position/orientation weight of 0.5.
- Set the starting joint with the function `ConstraintJoints`.
- Initialize the constraint with `InitializeGoal`. Use a step factor of 0.0.
- If this is a temporary constraint, one that will be used only once and then discarded, do *not* add the constraint to the global list with `AddConstraintToGlobalList`.
- Generate a constraint set with `GenConstraintSet`. This requires that the constraint be in a list. This also makes it possible to the this same thing with two constraints simultaneously.
- Call `SolveConstraints` with constraint set. You can optionally pass in a function which will be executed at each iteration of the solution process. In this case, the function `drawreach` draws the graphics windows in *Jack*, which will illustrate the intermediates steps of the solution process.
- Use a time limit of 0 as the final argument to `SolveConstraints`.
- `SolveConstraints` returns the distance between the final position of the end effector and the goal, weighted according to the constraint's weight.
- When done, delete the constraint with `DeleteConstraint` and delete the temporary list.

6.8 Things to Watch Out For

When creating constraints or changing their parameters, it is essential that you follow the procedure outlined here. Some of the fields of the constraint structure can be referenced explicitly, but several should be set only through the routines outlined above because of the internal bookkeeping that the evaluation process maintains.

Because of the way in which constraints are grouped into constraint sets for evaluation, it's essential that the constraint sets be regenerated when necessary. The process for signaling this is to delete the constraint sets with `DeleteConstraintSets`. This function is called by *Jack* in the following circumstances:

- A new constraint is created (`NewConstraint`).
- A constraint is deleted (`DeleteConstraint`).
- A constraint is turned on or turned off (`SetConstraintStatus`).
- The starting joint of a constraint changes. In this case, it is up to you to ensure that the constraints sets are deleted. You can set the starting joint either by assigning to the *startjoint* field or by calling the function `ConstraintJoints`.
- The priority of a constraint changes.

6.9 Miscellaneous Things

- The *behavior* field of the constraint tells whether the constraint is a human behavior constraint, pointed to by the human figure structure. These constraints are internal, and they are not written to environment files like constraints created by users.
- The *motion* field of the constraint tells whether the constraint belongs to a motion. These constraints are internal, and they are not written to environment files like constraints created by users.

Chapter 7

The Peabody Object Representation

7.1 The Peabody Environment

The *Peabody* environment consists of a collection of figures, segments, joints, and constraints. For convenience, figures, segments, sites, joints, and constraints are sometimes referred to collectively as peabody "constructs."

A segment is the basic geometric primitive. Typically, each segment has an associated *psurf*. The psurf describes the geometry relative to the local coordinate system of the segment.

A *site* is a coordinate frame specified relative to the segment's base coordinate system. A site represents a "handle", or a significant point on a segment. A site defines an attachment point where one segment is connected to another through a joint. Typically, the sites will lie on the surface of the segment. Each segment may have multiple sites associated with it. A joint connects two sites from different segments.

The transformation at each joint may have arbitrary degrees of freedom. Degrees of freedom may be specified by the user as a sequence of rotations and translations about arbitrary axes.

This mechanism provides flexibility in designing and manipulating articulated figures. Generally speaking, the site transformation will not be changed except as the figure is being designed. The figure is moved by adjusting the joint transformations.

Peabody represents articulated figures without imposing a predefined hierarchy upon them. From the user's point of view, the environment is a collection of segments connected by joints and constraints. However, an underlying hierarchy does exist through connections to the `world` segment. The global position of a segment or site in the environment is determined by the collective displacements across the joints which link each figure together.

Operations to be performed on the environment must have access to the global position of each segment and site. Peabody provides routines for accessing this information, as well as the connectivity of the objects.

Much of the source code in the peabody library is organized by *action* rather than by *object*. This means that routines for creating things are grouped together in one file (`new.c++`), routines for deleting things are grouped together in another file (`new.c++`), routines for writing things are grouped together in another file (`write.c++`), etc. Some of the source code is organized by *object*, though, in the case of `segment.c++`, `joint.c++`, `figure.c++`, and `constraint.c++`, each of which contain routines which specificallly operate on that particular type of data structure.

7.2 The Peabody Data Structure

The peabody data structures are declared in the include file `peabody.h`. *Peabody* encorporates a strong naming convention. Segments, sites, joints, and figures all have names associated with them. Site names are local to the segment to which they belong. Segment and joint names are local to their figure. In this way, segments belonging to different figures may have the same name. This mechanism allows easy reference in a textual description. In the peabody language, the full name of a segment is formed by appending the name of the segment to the name of the segment's figure, separated by a period. Joint names are formed similarly. Full site names are formed by concatenating the figure name, segment name and site name, all separated by periods.

Peabody maintains information about the environment which makes it easy to reference its component parts. It keeps lists of which segments, sites, and joints belong to which figure, as well as global lists of the figures, segments, sites, joints, and constraints for the entire environment. This implementation relies heavily on the list facility described in Section 9.6.

At times this information is redundant, but it provides a very simple means of accessing the various components of the environment. An example is that each figure maintains a list of the segments in the figure, and another list of all sites in the figure. The site list is redundant, since this information can be determined from the segments, but it makes the application of functions which operate on all sites in a particular figure very simple.

7.2.1 The World Segment

Peabody maintains a pseudo-segment called the *world* which is not part of any figure and which cannot be moved. This segment exist as a handle for the world coordinate frame. The world may have several sites. By default, it has one called **base**. The world has no geometry.

7.2.2 The Spanning Tree

Although peabody does not formally impose a hierarchy on the objects, a hierarchy does exist. This spanning tree fans out from the world segment across joints and constraints, through segments and sites throughout th entire environment.

Internally, the environment tree consists of special "root" fields in the peabody constructs which point to the "parent" consruct in the tree. Joints have a *rootsite* which points to the site closest to the world. Segments have a *rootsite* which is the site through which the tree "enters" the segment. Sites have a *rootjoint* field which points to the joint leading towards the world, if there is one. These fields allow access "upwards" in the tree.

7.2.3 The Segment

A synopsis of the data structure for the segment is:

```
typedef struct segment Segment;

struct segment {
    char        *name;
    char        *fullname;
    Figure      *figure;
    Site        *rootsite;
    List        sites;
    char        *filename;
    Psurf       *psurf;
    LightSource *light;
    Transform   *global;
    int         (*drawer)();
    void        *data;
    unsigned    uptodate : 1;
    unsigned    needsglobal : 1;
};
```

Each segment has a name. The *name* field gives the local name, local to the figure to which the segment belongs. The *fullname* field gives the fully qualified name of the segment, prefixed with the name of the segment's

figure. The *figure* field points to the figure to which the segment belongs. The *sites* field is a list of sites which belong to the segment.

The segment is the basic geometric primitive in the environment. Each segment may have a psurf associated with it, pointed to by the *psurf* field. The *filename* field give the name of the file from which the psurf was read. Under certain circumstances, a segment may not have a psurf. In this case, the *psurf* and *filename* fields are nil.

There is important distinction between segments and psurfs. Psurfs represent *purely geometric* information. A segment is an object which exists in the world; a psurf is the physical representation of the object.

The *rootsite* field points to the site through which this segment is attached to the world. Joints leading out of all other sites on the segment lead outwards in the environment tree.

The *light* field points to a light source, represented by the **attribute** library. This allows light sources to be manipulated as true physical objects. You may think of this field as specifying the luminance properties of the segment. If the light pointer is nil, then the segment is an ordinary, non-luminous object. No special geometry is currently associated with the light source; lights are simply directed point sources.

The *global* field stores the global transformation describing the placement of the segment in the world coordinate frame. This field is for internal bookkeeping only, and should never be accessed directly. Access to this transform should always be through the `GetSegmentGlobal` function described below.

7.2.3.1 Accessing the Segment Position

The global position and orientation of a segment may be determined with **`GetSegmentGlobal`**:

```
GetSegmentGlobal(segment,T)
Segment    *segment;
Transform  *T;
```

This routine fills in the transform T with the global transform describing the position of the segment. This routine relies on a series of internal flags to efficiently maintain the global position depending upon which joints have changed.

7.2.4 The Site

A synopsis of the data structure for the site is:

```
typedef struct site Site;
struct site {
    char        *name;
    char        *fullname;
    Segment     *segment;
    Joint       *rootjoint;
    List        joints;
    Transform   *global;
    unsigned    uptodate :  1;
    unsigned    needsglobal :  1;
    unsigned    needspush :  1;
};
```

A site is a local coordinate system specified relative to the coordinate frame of its segment. The site's segment is pointed to by the *segment* field. The primary purpose of a site it to define attachment frames for joints. The

joints field is a list of which joints reference this site, since more than one joint may be connected to a single site. The *constraints* field is a list of which constraints reference this site.

As mentioned before, a site is an attachment point, and it can represent any significant point relative to a segment. It is important to understand that a site is not just a *point*, but a complete coordinate frame, representing location and orientation. Any application which needs to access significant points on a segment should make their access through the site structure. Recall that there is a pseudo-segment representing the base of the environment, called the `world`. Significant points in world coordinates may be conveniently defined as sites on the world segment.

The *rootjoint* field points to the joint through which this segment is attached to the world. This field will be non-nil for only one site on each segment. This site is the the *rootsite* of the segment. To traverse the environment tree towards the world from a given segment, first travel from the segment to its *rootsite*, and then to the *rootjoint* of that site.

The *global* field stores the global transformation describing the placement of the site in the world coordinate frame. This field is for internal bookkeeping only, and should never be accessed directly. Access to this transform should always be through the `GetSiteGlobal` function described below.

7.2.4.1 Accessing the Position of a Site

The location of a site is a transform which specifies its position relative to the coordinate origin of its segment. This field may be accessed with the pair of functions `SetSiteLocation` and `GetSiteLocation`:

```
SetSiteLocation(site,L)
Site        *site;
Transform   *L;


GetSiteLocation(site,L)
Site        *site;
Transform   *L;
```

The function `SetSiteLocation` assigns the location transform L to the site. `GetSiteLocation` fills in the transform L with the site's location transform.

The global position of a site may be determined by the function `GetSiteGlobal`:

```
GetSiteGlobal(site,T)
Site        *site;
Transform   *T;
```

This routine relies on a set of internal flags to maintain the joint angles and efficiently determine the global position. This ensures that the minimum amount of computation is necessary to determine the position, so this routine is highly efficient.

7.2.5 The Joint

A synopsis of the data structure for the joint is:

```
typedef struct joint Joint;
```

```
struct joint {
    char        *name;
    char        *fullname;
    Figure      *figure;
    Site        *site1,*site2;
    Joint       *rootsite;
    DOF         *dofs;
    int         ndofs;
    Transform   *displacement;
    unsigned    uptodate :  1;
};
```

The *name* field gives the joint's name within its figure. The *fullname* field gives the name qualified by the figure's name. The *figure* field points to the figure to which the joint belongs.

A joint connects two sites, *site1* and *site2*. The displacement of a joint is defined as the transformation from *site1* to *site2*. This is regardless of how the segment is rooted. The *rootsite* field points to the site closer to the world in the environment tree. This field will be either *site1* or *site2*.

A joint may have a user-defined set of degrees of freedom. A degree of freedom is defined as a rotation or translation about an arbitrary axis.

A synopsis of the data structure for the degree of freedom is:

```
typedef struct dof DOF;

struct dof {
    char        type;
    float       axis[3];
    float       angle;
    float       llimit,ulimit;
    DOF         *next;
}
```

The *type* may be either 'r' for rotation or 't' for translation. The translational or rotational axis is given by *axis*. A list of DOF's represents a nested sequence of rotations and translations. The rotations are applied left to right, so at each step the transformations are with respect to the local (current) coordinate frame. The DOF specifies both the axes of rotation and translation, and the current angle or distance, in *angle*. Normally, the axis remains fixed once it is set, and only the angle ever changes.

7.2.5.1 Accessing the Joint Transformation

The transformation across the joint depends upon the degrees of freedom of the joint. If the joint has no degrees of freedom, then the transformation across the joint is an arbitrary homogeneous transform, stored in the *displacement* field. If the joint does have specific degrees of freedom, then the displacement is specified by a number of joint angles. The number of angles depends upon the number of degrees of freedom. The *displacement* field should never be accessed directly, either for reading or writing. It should only be accessed through the routines described below. The process of determining the composite joint transform from the degrees of freedom is handled internally by these routines.

The angles of a joint can be accessed through the routines `SetJointAngles` and `GetJointAngles`. The displacement transform across a joint can be accessed through `GetJointDisplacement`.

```
SetJointAngles(joint,angles,internal)
Joint     *joint;
float     angles[];
Boolean   internal;


int
GetJointAngles(joint,angles,internal)
Joint     *joint;
float     angles[];
Boolean   internal;


GetJointDisplacement(joint,D,I)
Joint       *joint;
Transform   *D,*I;
```

The functions `GetJointAngles` and `SetJointAngles` take arrays. The length of the array should at least match the number of degrees of freedom of the joint, which is never more than 6. `GetJointAngles` returns the number of degrees of freedom.

The function `SetJointAngles` assigns the joint angles regardless of the joint limits. However, it returns a boolean value specifying whether the angles are with the joint limits. This gives a method of determining when joint limits are exceeded.

Both of these functions take a flag *internal* which determines the ordering of angles within the array. If this flag is **FALSE**, then the angles are returned in the order in which they are defined by joint, i.e. *from site1 to site2*. This happens regardless of how the figure is rooted, so it is possible that the ordering of the angles will be ***upwards*** in the object hierarchy. This is sometimes objectionable, so if the *internal* is **FALSE**, the angles are returned in the direction of the object hierarchy. This ordering is illustrated in Figure 7.1 and is described in greater detail in Section 7.2.5.2. For practically all operations, it is best to pass this argument as **FALSE**.

The function `GetJointDisplacement` returns the composite transformation between the two sites of a joint. It takes two transform arguments for internal efficiency reasons. The *D* arguments refer to the transformation *from site1 to site2*. The *I* arguments refer to the transformation *from site2 to site1*, which is the inverse of *D*. Only one of these arguments need be supplied. Either of these arguments may be nil, in which case the necessary displacement is determined from the one provided. This is purely for efficiency sake to avoid needlessly inverting a transformation before supplying it for the joint displacement.

The displacement across a joint with no degrees of freedom may be set with the function `SetJointDisplacement`. This function should not be called with a joint which does have degrees of freedom.

```
SetJointDisplacement(joint,D,I)
Joint       *joint;
Transform   *D,*I;
```

7.2.5.2 The Joint Transformation

The transformation across a joint is always defined as *from site1 to site2*. This also defines the sequence in which the degrees of freedom should be interpreted.

There is a correspondence between the ordering of the *dof* list of the joint structure and the terms in the degree of freedom expression in the peabody language definition of the joint. The first element in the *dof* list is the leftmost term and the last list element is the rightmost term. The conventional interpretation for the primitive rotational and translational transformation terms which compose the joint transformation is left to right in local coordinates. Therefore, the transformation across a joint from *site1* to *site2* is composed of a primitive transformation by the rightmost term in the **type** field, which corresponds to the *last* element in the joint's *dof* list, followed by a transformation by the term second from the right in the **type** field, which corresponds to the next-to-last element in the joint's *dof* list, etc.

7.2.6 The Figure

A synopsis of the data structure for the figure is:

```
typedef struct figure Figure;

struct figure {
    char        *name;
    char        *filename;
    List        segments;
    List        sites;
    List        joints;
    Site        *root;
    Transform   location;
};
```

The *segments* and *sites* fields are lists of all of the segments and sites in the figure. The *joints* fields is a list of all the joints in a figure. Note that a joint may not belong to two different figures. If it did, the figures would not be separate[1]! Using these lists, functions can easily reference those segments, sites, and joints.

7.2.6.1 Accessing the Figure Location

The location of a figure is defined through the location of its root site, pointed to by the *root* field. The global location of this site is given by the figure's *location* field, and the location of all other sites and segments within the figure are subsequently defined in terms of this.

The location of the figure can be controlled by the following functions:

```
SetFigureRoot(figure,site)
Figure  *figure;
Site    *site;
```

[1] Siamese twins are one figure, not two!

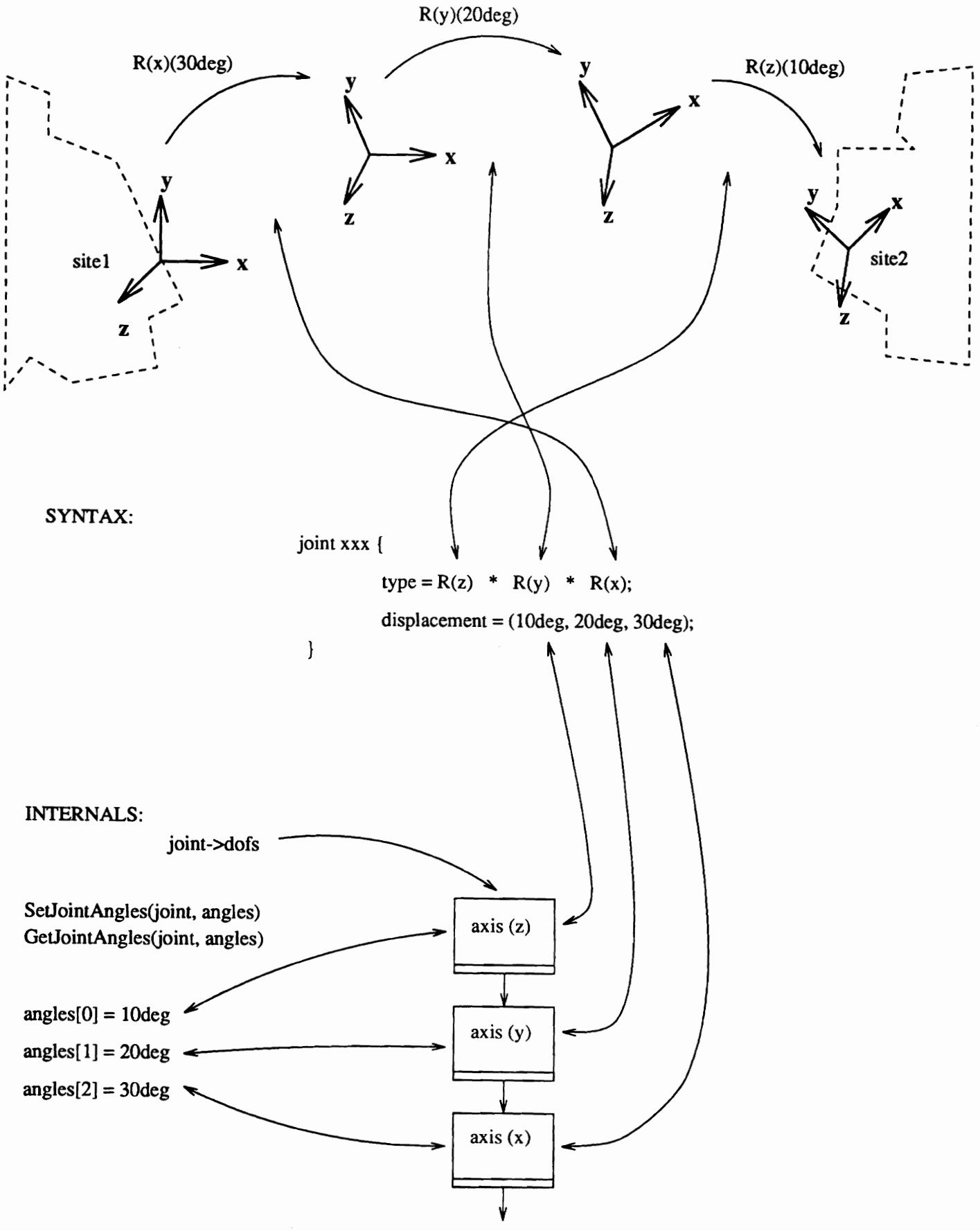

Figure 7.1: The Joint Transformation

```
SetFigureLocation(figure,L)
Figure      *figure;
Transform   *L;


GetFigureLocation(figure,L)
Figure      *figure;
Transform   *L;


SetSiteGlobal(site,L)
Figure      *figure;
Transform   *L;
```

The function `SetFigureRoot` takes pointers to a figure and a site and makes the site the root for the figure. The site must belong to the figure. This function automatically adjusts the figure's `location` field so that the figure doesn't "move."

The function `GetFigureLocation` returns with the global location of the figure's root site. This is identical to calling `GetSiteGlobal` with the figure's root site. `SetFigureLocation` takes a new transform for the root site.

The function `SetSiteGlobal` is an alternative way of setting the location of a figure. It takes a pointer to a site and global transform for that site. It sets the site's figure's location so that that site lies at the given transform.

7.2.7 The Environment

A synopsis of the data structure for the environment is:

```
typedef struct environment Environment;

struct environment {
    List        figures;
    List        segments;
    List        sites;
    List        joints;
    List        constraints;
    List        constraintsets;
};
```

An environment is a complete graph of a geometric world. For convenience, all the figures, segments, sites, joints, and constraints are referenced here. The root of the environment graph is the a pseudo-segment `world`. The world segment has no geometry, but it has sites through which constraints attach each figure to it.

7.2.7.1 The Environment Variable

There is only one environment structure in *Jack*, and is is pointed to by the global variable called `env`.

7.3 The Peabody Hierarchy

This section describes the internal workings of the peabody hierarchy. It describes the set of flags in each of the structures which maintain information about the hierarchy. This information is transparent to the casual user of the routines in the peabody environment. It is included here only as an explanation of how it works.

The maintenance of the peabody tree is at the heart of *Jack*, because it is through the tree that information about the geometric relationships of the environment is available. It is vitally important that this representation be as efficient as possible. *Jack*'s relatively simple access routines described above are designed to be easy to use from a programmer's point of view but still function efficiently.

Jack runs on Silicon Graphics IRIS workstations, and it is designed to take advantage of the IRIS hardware, particularly the matrix stack. The implementation of the peabody library does not depend heavily on the IRIS hardware except for the matrix stack, which is accessible through five simple subroutines: **pushmatrix**, **popmatrix**, **loadmatrix**, **multmatrix**, and **getmatrix**. The peabody library can function independently of the IRIS hardware by simulating the function of these routines in software. Such a library has been implemented in **gen/src/lib/glng** (named for IRIS GL with "no graphics")

Although it is important to ensure the portability of the peabody library, it is also beneficial make sure that its primary implementation, on the IRIS, is as efficient as possible. This means taking advantage of the matrix stack and the ability to multiply 4 × 4 matrices in hardware.

This facility has some peculiar features, however, which have a significant impact on the efficiency of the code. In particular, matrix multiplications in hardware are many times faster than they can be performed in software, but the function **getmatrix** is very expensive. A single call to **getmatrix** can take as long as 15 matrix multiplications!

The peabody library is designed with the following principles:

- When only two matrices need to be multipled, it is faster to do it in software, using the function **matmult**.
- Calls to **getmatrix** should be minimized. In other words, only matrices which are absolutely essential should be stored internally.

The matrix stack is 32 levels deep, but there is no maximum level to the peabody data structure. The peabody structure is very general in the way it allows joints to branch off of sites. A niave traversal of a moderately complex human figure model in peabody can reach 50 levels! It is important to make sure that the pushes and pops of the matrix stack are performed only when absolutely necessary.

7.3.1 The *cleantree* Flag

The environment structure has a flag called *cleantree*. This flag tells whether the peabody tree is valid. Every internal peabody routine examines this flag to make sure the tree is valid before proceeding to access the tree. If it is not, it calls the function **MkPeabodyTree**, which remakes the tree. Technically, this need only be done when a new object is read in, or when a figure is rerooted, or when the connectivity of a joint is changed. In practice, it is called more frequently because it is the best way of guaraneeing the validity of the other flags in the data tree.

Any routine which disturbs the peabody tree can signify that the tree needs to be rebuilt simply by setting the **env->cleantree** flag to 0.

7.3.2 The *uptodate* Flags

The segment and site structures have flags called *uptodate* which specify whether their *global* fields are currently valid. These are maintained by the functions **SetJointAngles**, **GetSiteGlobal** and **GetSegmentGlobal**. The function **SetJointAngles** traverses the tree from the given joint outwards and sets the *uptodate* flags of every site and segment to false. **GetSegmentGlobal** and **GetSiteGlobal** first examine the flag to determine if the segment or site is uptodate. If it is, then it returns with the already-computed transform. Otherwise, it recurses upwards in the tree until it either reaches a site or segment which is up to date or it reaches the figure's root. It then loads that transform onto the matrix stack, and as the resursion unravels downward in the tree, it multiples by the successive transformations, across joints and sites.

7.3.3 The *needsglobal* Flags

The update procedures described above compute global transformations for all segments between the world and the segment for which the information has been requested, but because of the expensive nature of the `getmatrix` subroutine, it does not automatically retrieve and store these transforms. To do so would cause excessive unnecessary retrievals for segments for which the information will not be needed.

The peabody library uses a form of heuristics to predict which segments and sites will need global information by recognizing that in the *Jack* environment, many things happen in loops and many operations are performed repeatedly with slight changes to the positions of certain figures or joints. Therefore, peabody maintains a *needsglobal* field in the segment and site which is set by the routines `GetSegmentGlobal` and `GetSiteGlobal`. Whenever peabody computes a transformation for a segment or site which has this flag set, it retrieves it off the stack and stores in internally, under the assumption that if it was requested before, it is likely to be requested again.

This flag is reset when the peabody tree is remade with `MkPeabodyTree`. This ensures that the flags do not remain set long after they stop being needed.

7.3.4 The *needspush* Flags

The site and segment data structure also maintain a *needspush* flag to define where the tree branches. This allows the tree traversal routine to selectively push and pop the matrix stack only where it is absolutely necessary. The peabody data structure is very general in the way it represents joints connected to sites; segments may branch to several sites, and sites may branch to several joints, but in practice, many parts of commonly used figures like the human figure, consist of long chains that don't require pushing and poping. A niave traversal of a typical human figure model can easily overflow the matrix stack. Thus the *needspush* flags determine when there are branches in the figure tree where the stack must be pushed or popped. These flags are determined by the function `settreetraverslflags`. They are used by the traversal functions `traverseenvfromroot` and `traverseenvfromsite`, all defined in the source code file `gen/src/lib/pea_tree.c++`.

7.4 Accessing the Spanning Tree

The primary way of accessing the spanning tree of the environment is through the function `JointPath`, which returns the path of joints from one site to another.

```
int
JointPath(site1,site2,joints)
Site   *site1,*site2;
Joint  *joints[];
```

This function fills the array *joints* with pointers to the joints in the path from *site1* to *site2*, and it returns the number of joints in the path. Notice that since every site in the environment must be rooted, so there will always be a path between the two sites. Some of the joints may be constraints, as determined by their *type* field. Remember that the direction of a joint is defined by its *site1* and *site2* fields, so the direction of the individual joints in the *joints* array not all be the same.

For convenience, the function `JointPathList` performs the same function but instead constructs a `List` out of path rather than returning the joints in an array.

```
List
JointPathList(site1,site2)
Site  *site1,*site2;
```

7.5 Reading the Peabody Language

The syntax of the peabody language is described in the *Jack User's Guide*. There are two distinct types of peabody files. Environment files contain any legal peabody syntax. Figure files contain "template" definitions for figures. Files are read primarily with `ReadEnvironment` and `ReadFigure`.

```
VALUE *
ReadEnvironment(filename)
char     *filename;


VALUE *
ReadFigure(name,filename)
char     *name;
char     *filename;


VALUE *
ReadPeabodyString(string)
char     *string;



VALUE *
ReadPeabodyStatement(file)
FILE     *env;
```

Environment files may be read with `ReadEnvironment`. The file may contain any valid peabody input. It may be figure definitions, constraints, joint displacements, etc. It need not be a "complete" environment. **ReadFigure** reads a figure file and constructs a figure from it. It returns a pointer to the figure, or nil if the figure could not be read. The *name* argument specifies the name of the figure. The function **ReadPeabodyString** reads peabody from a internal character string.

The function **ReadPeabodyStatement** reads a peabody statement from an open stream. Exactly one statement will be read from the file. A peabody statement is any simple single statement terminated by a semicolon, or a block of simple statements enclosed in curly braces.

The peabody language resembles "definitions" for figures, segments, etc. Actually, peabody constructs are defined, or created, when they are first referenced, whether inside a block or not. Subsequent occurrences of constructs refer to the original ones, instead of creating new one. This allows the same peabody file to be read twice. The first time will cause the creation of all referenced constructs. The second time will only have the effect of re-initializing the fields declared in the file.

7.6 Peabody Values

At the heart of the peabody language is a data structure defining a *value*, which has a type and a data component. This structure can be used to represent a kind of abstract data type, that is, a quantity that may optionally have one of several types. The peabody parser is based heavily on this notion o `VALUE` types. The routines for reading the peabody language return pointers to structures of these types. Since these values can represent many differen things, including numbers, strings, matrices, and arithmetic expressions, it allows the peabody parser to be used to read expressions of a quite arbitrary nature.

The `VALUE` structure is defined as:

```
typedef struct value    VALUE;
typedef enum valuetype  VALUETYPE;


enum valuetype {
    V_UNDEF, V_UNSUPPLIED,
    V_VARIABLE, V_OP, V_FUNCALL,
    V_NUMBER, V_MATRIX, V_STRING, V_DOF, V_VECTOR,
    V_FIGURE, V_SEGMENT, V_LIGHT, V_SITE, V_JOINT, V_CONSTRAINT,
    V_NODE, V_EDGE, V_FACE, V_ATTRIBUTE
};

struct value {
    VALUETYPE   type;
    union {
        float       number;
        Matrix      matrix;
        char        string[64];
        DOF         *dof;
        VALUE       *vector;
        Figure      *figure;
        Segment     *segment;
        Site        *site;
        Joint       *joint;
        Constraint  *constraint;
        Attribute   *attribute;
        struct {
            Segment     *segment;
            short       n;
        } item;
        struct      {
            char        *name;
            VALUE       *args;
        } funcall;
        struct      {
            char        type;
            VALUE       *left,*right;
        } op;
    }           v;
    VALUE       *next;
};
```

The types are:

V_UNDEF Undefined type. This means that none of the union fields are valid.

V_UNSUPPLIED A special type which corresponds to a the special symbol $ in the peabody language. This is used by *Jack* to signal that a parameter has been left unsupplied.

V_NUMBER A numerical value. There is no distinction between integers and floating point values. The *number* field stores the number, as a float.

V_MATRIX A homogeneous transform. In the language, this value is generated as a product of `xyz`, `trans`, and `scale` expressions. The *matrix* field contains the transform.

V_STRING A character string. In the language, this is anything inside double quotes. The *string* field contains the string, which is limited to 64 characters (the size of the rest of the union).

V_VARIABLE A character string, generally assumed to be referring to a value stored in the peabody symbol table. In the language, this is any character string *not* inside double quotes. The *string* field contains the name of the variable. When this type of value is evaluated by `evalval` described below, it looks up the value in the symbol table. Values are placed in the symbol table in the language through assignment statements to variables which are not peabody keywords. An exception to this is the interpretation of variables as strings. When a character string occurs in a place where a string is expected, peabody assumes that the variable *is* the character string and it uses the variable's name for its value.

V_DOF A degree of freedom list. In the language, this value is a product of `R` and `T` operators. The *dof* field points to the `DOF` structure.

V_VECTOR An arbitrary list of values. In the language, this is a comma-separated list of values inside parentheses. The list may be of arbitrary length. The list of values is stored in the *vector* field. The *next* field points to the next element of the list.

V_FUNCALL A "function call." Syntactically, this is an identifier followed by an arglist. This is returned in the `funcall` structure, with *name* as the name of the "function" and *args* its arguments. No function is evaluated.

V_OP An arithmetic operation. Syntactically, this is any sequence of numbers or variables, together with `+`, `-`, `*`, `/`, or `^`. The type is a character giving the symbol. The operation operates on the *left* and *right* values. The operands may in turn be values of this type, thus forming an expression tree. The evaluation of these values is discussed below.

V_FIGURE A figure. The *figure* field points to the figure.

V_SEGMENT A segment. The *segment* field points to the segment.

V_LIGHT A light. The *light* field points to the light.

V_SITE A site. The *site* field points to the site.

V_JOINT A joint. The *joint* field points to the joint.

V_CONSTRAINT A constraint. The *constraint* field points to the constraint.

V_NODE An node. The *item* structure holds the reference to the node. The field *item.segment* points to the node's segment; the *item.n* field gives the node's index.

V_EDGE An edge. The *item* structure holds the reference to the edge. The field *item.segment* points to the edge's segment; the *item.n* field gives the edge's index.

V_FACE An face. The *item* structure holds the reference to the face. The field *item.segment* points to the face's segment; the *item.n* field gives the face's index.

V_ATTRIBUTE An attribute. The *attribute* field points to the attribute.

The following functions operate on value structures. They are most useful in manipulating the values returned by the peabody parser.

```
VALUETYPE
evalval(result,val)
VALUE       *result;
VALUE       *val;
```

```
int
valtovec(vec,max,val)
float  vec[];
int    max;
VALUE  *val;


valtostringvec(strings,max,val)
char   *strings[];
int    max;
VALUE  *val;


char *
valtostring(val)
VALUE   *val;


VALUE *
dupval(val)
VALUE    *val;
```

The function **evalval** evalutes a value. If the value is a **V_OP**, it recursely evaluates the left and right operatands and applies the operation to them. If it is a variable, it looks the value up in the symbol table and returns it. Otherwise, the value doesn't need evaluating, and the function just copies *val* into *result*.

The function **valtovec** converts a value into an array of numbers. It expects the value to be of type **V_VECTOR**. It loops over the elements of the vector and evaluates them in turn, placing the result in the *vec* array. The function returns the number of values it filled in. It will not fill in more than *max* elements. If the value passed in is a **V_NUMBER**, it places it in the first element and returns 1. **valtovec** first evaluates the value passed in, so if the passed in value is of type **V_VARIABLE**, the conversion will be done on the looked-up value of the variable.

The function **valtostringvec** converts a value into an array of strings. It expects the value to be of type **V_VECTOR**, and it expects that each element of the vector to be of type **V_VARIABLE** or **V_STRING**. It loops over the elements of the vector, placing the result in the *strings* array. The function returns the number of values it filled in. It will not fill in more than *max* elements. Unlike **valtovec**, this function does *not* evaluate the value first. If the value passed in is a **V_VARIABLE**, or if an element of the vector is a **V_VARIABLE**, it uses the name of the variable as the value.

The function **valtostring** converts a single value to a character string. This function returns **NULL** if the value is not a **V_VARIABLE** or **V_STRING**.

The function **dupval** duplicates a value, including any sup-elements, as in the case of a **V_OP**. This is critical in the case of the values returned by the peabody parser, since the parser uses a static internal storage area which is overwritten each time the parser processes a line. If you need to store an expression tree internally which has been generated by the parser, you must duplicate its value and store the duplicated value.

7.7 Creating Parts of the Environment

Parts of the environment are created with the "new" functions.

```
Figure *
NewFigure(name)
char       *name;
```

```
Segment *
NewSegment(name,figure)
char        *name;
Figure      *figure;


Site *
NewSite(segment,name)
Segment   *segment;
char      *name;


Joint *
NewJoint(name,figure)
char      *name;
Figure    *figure;
```

NewSegment requires a pointer to the figure to which the segment belongs. **NewSite** requires a pointer to the segment to which the site belongs. **NewJoint** requires a pointer to the figure to which the joint belongs, along with a *type*, which must be one of the enumerated types **internal** or *type*. These functions allocate the necessary memory and update the necessary lists, but they do not fill in any of structure the values.

The function **NewJoint** is not much use by itself, since it doesn't specify the sites which the joint connects. This is done by **AssignJointConnectivity**.

```
AssignJointConnectivity(joint,site1,site2)
Joint   *joint;
Site    *site1;
Site    *site2;
```

At a higher level, it is frequently necessary to create figures and segments and initialize them in the process. The following two routines read psurfs and construct things out of them.

```
Figure *
CreateFigure(filename,name)
char       *filename;
char       *name;
```

CreateFigure reads a psurf from *filename* and creates a figure with a single segment, assigning the psurf to the segment.

7.8 Dealing with Names

The "find" routines search for constructs with specific names. Each function searchs a list of constructs for the one with the given name. The *name* argument should be a identifier, not a compound name.

```
Figure *
FindFigure(figures,name)
List      figures;
char      *name;


Segment *
FindSegment(segments,name)
List       segments;
char       *name;


Site *
FindSite(sites,name)
List    sites;
char    *name;


Joint *
FindJoint(joints,name)
List     joints;
char     *name;



Constraint *
FindConstraint(constraints,name)
List          constraints;
char          *name;
```

Each function takes a name and a list to search. If the named part is not in the list, then the function returns nil. `FindJoint` may, of course, be used to find constraints as well.

In the peabody language, environment parts are referenced using the "dot" notation, by using a period between the names of the part and its "parent" part. For instance, `foo.bar.x` may refer to site `x` on segment `bar`, which is part of figure `foo`. Identifiers of this syntax may be parsed using the "FindNamed" functions. Each function take a pointer to a character string name and an optional pointer to the "parent" part. If the parent pointer is not nil, then the function acts like the corresponding "Find" function. If the parent pointer *is* nil, then the identifier is assumed to give the name of the parent in the dot notation. In this case, the *bufp* argument is set to point to the remaining part of the string. Therefore, the *bufp* argument must be a modifiable variable!

```
Site *
FindNamedSite(segment,bufp)
Environment  *env;
Segment      *segment;
char         **bufp;


Segment *
FindNamedSegment(figure,bufp)
Figure     *figure;
char       **bufp;
```

```
Joint *
FindNamedJoint(figure,bufp)
Environment  *env;
Figure       *figure;
char         **bufp;
```

These functions work in tandem to parse a string which may contain multiple dots.

7.8.1 Generating Unique Names

The names are local to the environment part to which they belong, but they must be unique within that part. In an interactive setting, sometimes names aren't commonly used, so determining unique names can be difficult. To avoid this, there are routines which generate unique names for figures, segments, sites, and joints. These routines take a "base" name and append it with an integer to generate a unique name. Each function takes a list of parts to search through in determining whether the name is unique.

```
UniqueFigureName(name,base,figures)
char  *name;
char  *base;
List  figures;

UniqueSegmentName(name,base,segments)
char  *name;
char  *base;
List  segments;

UniqueSiteName(name,base,sites)
char  *name;
char  *base;
List  sites;

UniqueJointName(name,base,joint)
char  *name;
char  *base;
List  joint;
```

The *base* argument is the initial name, and *name* is the generated unique name. If there is no conflict with the name *base*, then it is copied directly to *name*.

These functions must generate legal peabody names, which means that the names must be legal peabody identifier, consisting of legal symbols but also not conflicting with peabody keywords. An example of when this conflict may easily occur is in attempting to name a figure "light." `light` is a peabody keyword! The function `ispeakey` returns whether or not a string is a peabody keyword.

```
Boolean
ispeakey(word)
char     *word;
```

```
legalizepeaname(legal,word,suffix)
char  *legal;
char  *word;
char  *suffix;
```

The function `legalizepeaname` takes a string and makes it into a legal peabody name, by converting all illegal symbols to underscores and ensuring that the name is not a keyword. If the original *word* is a keyword, the string *suffix* is appended to it. The legal name is returned in *legal*.

7.9 Writing the Environment

Parts of the environment may be written to files with the "write" functions. The results of these functions may then be read with the peabody parser.

```
WriteEnvironment(filename,doconstraints)
char     *filename;
Boolean  doconstraints;


WriteEnvironmentFile(file,doconstraints)
FILE     *file;
Boolean  doconstraints;
```

The function `WriteEnvironment` writes out a complete copy of the environment to the named file. The function `WriteEnvironmentFile` writes out a complete copy of the environment to an opened file. If the *doconstraints* flag is set, it writes the constraints, too.

The following functions are useful for printing out the current values of figure locations and joint displacements. The *level* argument is either 0 or 1, specifying the indentation level. The *figurename* flag specifies whether the figure name should be printed. If it is not, then it is important to make sure that this is printed inside of a block in which the figure name *is* printed.

```
WriteFigureLocation(figure,file,level,figurename)
Figure   *figure;
FILE     *file;
int      level;
Boolean  figurename;


WriteFigureRoot(figure,file,level,figurename)
Figure   *figure;
FILE     *file;
int      level;
Boolean  figurename;
```

```
WriteJointDisplacement(joint,file,level,figurename)
Joint    *joint;
FILE     *file;
int      level;
Boolean  figurename;
```

7.9.1 Printing Things to Strings

The function `sprintdisplacement` prints the angles of a joint into the character string *buf*. The return value of the function is the length of the resulting string.

```
int
sprintdisplacement(buf,dof)
char  buf[];
DOF   *dof;
```

Homogeneous transformations may be formatted with `sprinttransform`, which formats the transform as a rotation component times a translation component. The return value of the function is the length of the resulting string.

```
int
sprinttransform(buf,t)
char       buf[];
Transform  *t;
```

Most of the numerical values in the peabody language have physical significance, in terms of angles, distances, and masses. The peabody language allows various units of measure to be used in inputting the values, and the printing routines allow the values to be formatted in the same way. The following functions format dimensioned values with the unit identifier, using the "current" units. The *precision* argument is number of digits to the right of the decimal point. The return value of the function is the length of the resulting string.

```
int
sprintangle(buf,angle,precision)
char   buf[];
float  angle;
int    precision;


int
sprintdistance(buf,d,precision)
char   buf[];
float  d;
int    precision;
```

```
int
sprintmass(buf,m,precision)
char   buf[];
float  m;
int    precision;
```

VALUE structures may be formatted using **sprintvalue**. Numbers are printed without decimals if their values are integral. Vectors are formatted with parentheses and commas. Strings are formatted in double quotes. Peabody constructs are printed as typecasts. The return value of the function is the length of the resulting string.

```
int
sprintvalue(buf,value)
char   buf[];
VALUE  value;
```

7.10 Collision Detection

Peabody has a collision detection mechanism that works efficiently on convex objects. It is an implementation of the algorithm of Gilbert and Johnson, which measures the minimum distance between to convex point sets and returns the points on either point set closest to the other. It uses frame to frame coherence to achieve near-linear speed. The principle behind this approach is to maintain information about the closest points between two objects as the routine is invoked over and over again. It uses the asumption that the previously-closest points will be a good first guess at the closest points for the current iteration.

The peabody implementation of this facility measures the distance between between two segments with the function **SegmentSegmentDistance**.

```
Couple *
SegmentSegmentDistance(segment1,segment2)
Segment   segment1;
Segment   segment2;
```

This routine returns a pointer to a couple structure which records information about the distance between the segments.

```
typedef struct {
    Segment *seg1;
    Segment *seg2;
    Vector  pt1,pt2,dist;
    char    type1,type2;
    short   k1,k2;
} Couple;
```

This structure maintains in *pt1* and *pt2* the closest points on *seg1* and *seg2* respectively. These points may lie at a node of the psurf, along an edge, or in the interior of a face. The fields *type1* and *type2* specify which with a value of `'n'`, `'e'`, or `'f'`, respecively. *k1* and *k2* give the corresponding index of the psurf item.

Peabody maintains an internal list of these `Couple` structures so that when the distance between two segments is requested, it will first look to see whether a couple structure exists for these segments.

7.11 The Peabody Parser

The peabody parser is generated with Yacc. The yacc input specification defines the basic structure of the language, including the blocks delimited by curly braces, the references to figure files, the assignment statements, and arithmetic expressions. It does not, however, define the peabody keywords and functional primitives. The peabody keywords are stored in internal tables. The parser consults these tabless to determine if a given character string is a keyword. The tables pair functions with the keywords. These functions perform the basic operations of the language.

Since the tables are independent of the parser, it is relatively easy to extend the language by adding new keywords. In previous versions of *Jack*, these tables were static and adding to them required modifying the code which defined them. There is now an algorithmic procedure for creating and adding things to the tables, through the pair of functions `DefinePeabodyBlock` and `DefinePeabodyAssign`. Peabody blocks are the things delimited by curly braces. There are currently blocks for figures, segments, sites, joints, and constraints. Peabody assignment statements go inside of blocks, and each assignment statement can be local to a particular type of block.

```
int
DefinePeabodyBlock(char *name,int (*blockfunc)())


DefinePeabodyAssign(char *name,int block,int (*assignfunc)())
```

The function `DefinePeabodyBlock` returns with a block type. This can be used as the block argument to `DefinePeabodyAssign` to define assignment statements inside of that block. The *name* argument to `DefinePeabodyBlock` defines the name of the block (i.e. **`figure`**, **`segment`**, **`site`**, etc. The `blockfunc` function argument sets the state for the peabody parser. To see how to do this, consult one of the examples where blocks are created. The `blockfunc` is invoked by the parser with the name of the object (the figure name, segment name, etc.), and the function must determine whether the object needs to be created or whether it is a reference to an existing one. It then sets the parser's current state indicator with the block type.

The `DefinePeabodyAssign` accepts the name of the assign statement (the identifier left hand side), the name of the block inside of which it should occur, and a pointer to the function to execute when it occurs. The block identifier allows the same character string to be used in different blocks to mean different things. This particular function will only be invoked when the assignment statement occurrs inside of this type of block. The structure of the assignment functions is described below.

An example of theis procedure is shown in Figure 7.2, which illustrates the peabody figure block definition and the definition for the **`archive`** assignment statement.

There are separte keyword tables for the constructs, the functions, and the assignment statements. Most extensions come in the form of new assignment statements, since these correspond to fields in figures, segments, sites, and joints.

The file `gen/src/lib/pea_keyword.c++` contains the source code for the keyword table for the peabody parser. This file declares arrays of type `Peakey`. The first entry is the character string name of the keyword, which must contain only alpha-numeric characters, and must begin with a letter. The second argument is an internal numerical identifier. These identifiers are defined in the include file `gen/include/parse.h`. The

```
peastate_figure(key,name)
Peakey  *key;
char    *name;
{
    Figure      *figure;

    if (peastate->block != PK_UNDEF) {
        error("invalid figure block\n");
    }

    if ((figure=lookupfigure(name,TRUE))) {
        setpeastate(key->key,figure,0);
    }
}

    ...

    PK_FIGURE = DefinePeabodyBlock("figure",peastate_figure);

    DefinePeabodyAssign("archive",PK_FIGURE,assign_archive);
```

Figure 7.2: **The Peabody Figure Block Definition**

numerical values are arbitrary, but each one must be unique. The third field in the **Peakey** structure is the address of a function which is invoked when the keyword occurs.

The functions for the assignment statements in the peabody language are defined in the source code file **gen/src/lib/pea_assign.c++**. Each function is invoked with the following arguments:

```
Peakey      *peakey;
int         block;
void        *val;
VALUE       *expr;

(*assign_func)(peakey,block,val,expr);
```

Each function must accept these arguments. The first argument *peakey* is a **Peakey** structure pointing the to the current keyword structure. It is passed here mostly for reference. The second *block* argument is the type of the current block. It is passed so that each assignment statement can check to make sure that it occurs in the proper block. Note that some statements are legal in several different types of blocks, (e.g. **location** may appear in either a site or a figure). The third argument is a generic pointer to the data element of the current block. Thus, if the current block is a segment, then *val* will be a pointer to a segment. The final argument *expr* is a **VALUE** structure giving the right hand side of the assignment statement. This expression is un-evaluated, so it should be evaluted before being used.

Chapter 8

The Psurf Geometric Primitive

The basic geometric primitive used in the Graphics Lab is called the *psurf*, for *Polygonal SURFace*. It represents the boundary of a geometric object as a graph of nodes, edges, and faces. The psurf library has functions for reading psurfs from files in both text and binary form.

The psurf data structure and library represent objects as simple geometric primitives. There is no hierarchy, and all coordinates are in local coordinate systems. The psurf library is used in conjunction with the peabody environment representation. Each segment in the peabody environment is a single primitive, and has a psurf associated with it. There is no explicit modeling transform built into the psurf structure. It is the responsibility of peabody to represent the modeling transformation which place a segment in the world coordinate system.

The structure definitions for a psurf are declared in the include file `psurf.h`. `psurf.h` includes `attribute.h`, so `attribute.h` and the files which it includes (`vec.h`, `gl.h`, `math.h`, `stdio.h`) need not be included explicitly.

8.1 The Psurf Data Structure

A psurf represents the boundary of an object as a graph of nodes, edges, and faces. Each element of the graph points to each other element, so complete connectivity information is available. Basically, a psurf has a table of nodes, a table of edges, and a table of faces. The references from nodes to faces, etc. are represented as indices into the corresponding table. No pointers are used. Since the table indices are stored as short integers (two bytes), much space in conserved.

For convenience, the nodes, edges, faces, curves, and patches of a psurf are sometimes referred to collectively as "items".

A synopsis of the psurf data structure is:

```
typedef struct psurf Psurf;

struct psurf {
    short       nnodes;
    short       nedges;
    short       nfaces;
    struct {
        Vector      *coords;
        Vector      *normals;
        FaceIndex   **faces;
        short       *nfaces;
        EdgeIndex   **edges;
        short       *nedges;
    } nodes;
```

```
    struct {
        VertexIndex *heads;
        VertexIndex *tails;
        FaceIndex   *lefts;
        FaceIndex   *rights;
        DLEntry     *displaylist;
    } edges;
    struct {
        short       *nvertices;
        VertexIndex **vertices;
        short       *nedges;
        EdgeIndex   **edges;
        Vector      *normals;
        Vector      *centers;
        short       *attributes;
        DLEntry     *displaylist;
    } faces;
    Attribute       **attributes;
    float           min[3],max[3];
    float           span[3],extent;
    float           sx,sy,sz;
    Csurf           *csurfs;
    int             ncsurfs;
};
```

Each of the fields in the above structures is a pointer to an array of the appropriate length. The length of the fields in the nodes structure is given by *nnodes*. The length of the fields in the **edges** structure is given by *nedges*. The length of the fields in the faces structure is given by *nfaces*.

8.1.1 Lazy Evaluation and Psurf Fields

The psurf library uses a form of lazy evaluation to maintain the data structure. This means that only information which is absolutely necessary is computed and stored internally. This is the reason that each field in the data structure has its own array: each array will be allocated and filled in only if it needs to be.

This organization facilitates storing and retrieving information in binary psurf files. A binary psurf file is basically a binary "dump" of the fields in the data structure, each preceeded by an identifying header. This file is generated by a program, *bps* or *Jack*, which generates all these fields and writes them to a file. Subsequently, when *Jack* reads information from the psurf file, it can read only the information which it absolutely needs, saving space and time.

When accessing the data structure or performing operations on it, it is important to make sure that the needed fields exist beforehand. This is done through the routine **getpsurffields**.

```
Psurf *
getpsurffields(psurf,fields)
Psurf         *psurf;
unsigned int  fields;
```

The *fields* argument is a bit-string which is formed by OR-ing together the values:

```
PSURF_DIMENSIONS   NODE_COORDS       NODE_NORMALS
NODE_FACES         NODE_EDGES        EDGE_HEADS
EDGE_TAILS         EDGE_LEFTS        EDGE_RIGHTS
EDGE_DISPLAY       FACE_VERTICES     FACE_NORMALS
FACE_EDGES         FACE_ATTRIBUTES   FACE_DISPLAY
FACE_CENTERS
```

Each value corresponds directly to the field in the data structure. This function examines the state of the psurf to make sure that the requested fields exist. If it does not, then if the psurf came from a binary psurf file, it opens the file and reads the information from it. Otherwise, it computes the information from the existing fields.

For example, the following call ensures that the node coordinates and face vertices fields have been read and filled in.

```
getpsurffields(psurf,NODE_COORDS|FACE_VERTICES);
```

Figure 8.1: `getpsurffields`

8.1.2 Psurf Nodes

Each of the arrays in the *nodes* structure has *nnodes* entries, one per node. The description of the k-th node is taken from the k-th entry in each of the arrays.

Each node represents a point in space as a vector *coords*, along with its surface normal *norm*, which is an average of the normals of the neighboring faces. Each node also "points" to the faces and edges which contain it. There are *nfaces* faces which reference the node. The *faces* field is a dynamic array of indices into the psurf's face table. There are *nedges* edges which reference the node. The *edges* field is a dynamic array of indices into the psurf's edge table. The ordering of the entries in the *faces* and *edges* arrays is not currently significant.

The example code segment in Figure 8.2 prints out the coordinates of each node in the psurf.

```
for (i=0; i<psurf->nnodes; i++) {
    printf("(%f,%f,%f)\n",
        psurf->nodes.coords[i][0],
        psurf->nodes.coords[i][1],
        psurf->nodes.coords[i][2]);
}
```

Figure 8.2: `Psurf Nodes`

There is a minor difference in the terminology used to describe the nodes in a psurf. The *nodes* are the actual points in space. When references to the nodes occur in the faces and edges, they are called *vertices*. Therefore, faces and edges have *vertices*, which are references to nodes.

8.1.3 Psurf Edges

Each of the arrays in the *edges* structure has *nedges* entries, one per edge. The description of the k-th edge is taken from the k-th entry in each of the arrays.

An edge points to two vertices, one at the *head* and one at the *tail*. It also points to the faces on its *left* and *right* side. If you were walking along the surface from the *tail* to the *head*, the *left* face would lie on your left and *right* face would lie on your right. This orientation is significant.

The edges are not stored directly in the psurf text file. The file lists only sequences of nodes which form faces. The edges are defined implicitly by the sets of nodes which are adjacent some face. Computing the edges is very time consuming: the psurf library uses a linear-time algorithm to compute the edges from the face vertex information, but the algorithm is $O(n^2)$ in space, which can make it very slow. This is the benefit of using a binary psurf file: the edges can be calculated once and then stored directly in the psurf file. The routine **psurfedges** which computes the edges is in the source code file **gen/src/lib/psurf_edge.c++**.

The edges are important because they are what *Jack* draws as the image of the psurf in wireframe. Since the speed of *Jack* is many times bounded by the speed with which the graphics hardware and draw the objects, it is important that the drawing mechanism be as efficient as possible. For this reason, it is not acceptable to draw objects by outlining each face. This would draw each edge twice.

8.1.4 Edge Display Lists

The psurf library has another optimizing mechanism to speed up the drawing process, in the form of display lists. The display list field is a preprocessed array containing information necessary to draw the edges of the psurf. This array exists to accelerate the drawing of psurfs in *Jack*. The array consists of vertex and color information, ordered in such a way that it is very efficient to loop over the array and pass its contents to the IRIS Graphics Library. In particular, the IRIS Graphics Library can draw connected line sequences much more efficiently than drawing each line segment individually. The edge display list generation routine, **genedgedl**, searches through the psurf for long sequences of edges which it can draw as single connected line sequences. The display list then consists of sets of sequences of vertex and color data.

The displaylist is an array of **DLentry** structures. This type is a union of a single float and a single integer. This allows the contents of the array to be stored accessed as either a float or an integer.

```
typedef union {
    float   f;
    int     n;
} DLentry;
```

The edge and face display lists have different formats. The format of the edge display list is described here. The format of the face display list is described in the next section.

The first entry in the edge display gives the length of the data portion of the list, not including the first entry, or the last, which is a 0 in it signifying the end of the list. Therefore, if the first entry in the list is **n**, then the actual length of the chunk of memory storing the list is **n+2**. After the first entry, the display list is a number of sets of vertex data:

nvertices
color
half intensity color
x_0
y_0
z_0
x_1
y_1
z_1
⋮

The **nvertices** field gives the number of (x, y, z) coordinates in the vertex set. The **color** field gives the the color in the form of the GL subroutine **cpack**. In addition, the uppermost byte of the color field gives the index of the attribute which this line is drawn in. This is necessary only for redoing the color of the display list when the properties of the display list change. The half intensity color gives the color required to draw the coordinate axis projections in *Jack*. The display list is terminated by an entry with **nvertices=0**.

The display list stores actual color values in RGB coordinates, although the colors of psurf actually come from the attributes which the faces refer to. If the parameters of the attributes change, the display list must be updated to reflect the change in color. This is handled by the *timestamp* field in the attribute structure and the *edges.timestamp* field in the psurf. The attribute timestamp tells when the attributes have last changed values. The edge timestamp tells when the edge displaylist was last updated. *Jack* draws the display list with the function **drawedgedl**, which first compares the two timestamps to see if the display list is out of date with respect oo the attributes. These functions are in the file **gen/src/lib/psurf_display.c++**.

Therefore, whenever the attribute parameters change, the *timestamp* of the attribute should be updated, using **gettimestamp**, described in Section 9.7. This will cause the display lists to be updated the next time *Jack* draws the psurf.

The edge timestamp should be set to zero if the the attribute information in the psurf changes, for example, if a face is given a new attribute.

8.1.5 Psurf Faces

Each of the arrays in the *faces* structure has *nfaces* entries, one per face. The description of the k-th face is taken from the k-th entry in each of the arrays.

The vertices of the face are specified in the *vertices* field, which contains indices into the psurf's node table. There are *nvertices* of them. The edges of the face are specified in the *edges* field, which contains indices into the psurf's edge table. There are *nedges* of them. The ordering of the edges is not currently significant; the ordering of the vertices *is* significant. The vertices are specified in counter-clockwise direction, so that traversal according to the right-hand rule yields an outward-pointing normal. Typically, the traversal of faces is done by looping through the vertices, not the edges.

The surface normal of the face is specified by *norm*. If the face is not planar, then *norm* will be as close as possible to the true normal. The center of the face is maintained in *center*. This is merely the average of the positions of the vertices.

8.1.6 Face Display Lists

The display list field is a preprocessed array containing information necessary to draw the faces of the psurf. This array exists to accelerate the drawing of psurfs in *Jack*, when the object is shaded. The array consists of vertex and material property information, ordered in such a way that it is very efficient to loop over the array and pass its contents to the IRIS Graphics Library.

The display list is an array of **DLentry** structures, but its format is different from the edge display list described above.

The first entry in the face display gives the length of the data portion of the list, not including the first entry, or the last, which is a 0 in it signifying the end of the list. Therefore, if the first entry in the list is **n**, then the actual length of the chunk of memory storing the list is **n+2**. After the first entry, the display list is a number of sets of vertex and surface normal data. There can be one normal per face or one normal per vertex. This is signified in the list by the sign of the **nvertices** entry: if **nvertices** is positive, there is one normal per vertex; if **nvertices** is negative, there is one normal for the entire face.

The vertex section for a face with a normal for each vertex is:

nvertices
material
n_{x_0}
n_{y_0}
n_{z_0}
x_0
y_0
z_0
n_{x_1}
n_{y_1}
n_{z_1}
x_1
y_1
z_1
⋮

The vertex section for a face with one normal for the whole face is:

- nvertices
material
n_x
n_y
n_z
x_0
y_0
z_0
x_1
y_1
z_1
⋮

The `material` field gives the IRIS GL library lighting index of the face, for the GL subroutine `lmbind`. The uppermost byte of the `material` field gives the index of the attribute which this face is drawn in. This is necessary only for redoing the color of the display list when the properties of the display list change. The display list is terminated by an entry with `nvertices=0`.

Unlike the edge display list, the face display list makes a reference to the properties of the attribute, through the material index, so it is not necessary to compare the time stamps of the attributes and faces when the attribute parameters change. However, the face timestamp should be set to zero if the the attribute information in the psurf changes, for example, if a face is given a new attribute.

8.1.7 Attributes

Each face refers to an attribute indirectly, through an index into the psurf's attribute table, the *attributes* field in the psurf structure. This field is an array of pointers to surface attributes. Each face has an attribute field which is an array of indices into this table.

8.1.8 Psurf Dimensions

Several useful pieces of information are maintained about the psurf primitive. The bounding box of the psurf is maintained in the *min* and *max* fields. Also, the field *span* is the difference between *min* and *max*, and *extent* is the magnitude of the *span*. This information gives an approximation of where the psurf is in space, and how big it is. These fields are accessible by calling `getpsurffields` with the argument `PSURF_DIMENSIONS`.

8.1.9 The Psurf Scale

The psurf has a scale parameter, maintained in the fields *sx*, *sy*, and *sz*. Each is a scale along the specified axis. This scale represents the scale factor applied to the psurf's coordinates as specified in the psurf file to yield the coordinates as they are stored internally in the node *coords* field.

This scale factor typically comes from the reference to the psurf in the peabody language. The peabody statement which references the file may specify an optional scale:

```
psurf = "arm.pss" * scale(5.6,7.3,30.1);
```

These numbers are store in the psurf's scale parameter, and this scale is applied to the coordinates as they are read from the file.

When the coordinates of a psurf are written out to a file, the node coordinates are multiplied by the inverse of this scale.

8.2 Syntactic Representation of Psurfs

Psurfs may be described syntactically in text files. A psurf file is a textual representation for the node, edge, and face tables of the psurf. By convention, these files have the suffix `.pss`. The file lists a set of nodes, edges, and faces.

```
0   0   0
0   1   0
1   1   0
1   0   0
0   0   1
0   1   1
1   1   1
1   0   1
; {end of nodes}
; {end of edges}
1 2 3 4;                 { back }
1 4 8 5;                 { bottom  }
3 7 8 4;                 { right side }
1 5 6 2 [attribute 1];   { left side }
5 8 7 6;                 { front }
2 6 7 3;                 { top }
```

Figure 8.3: An example psurf

A psurf file is a list of nodes, which are specified as triplets of real numbers. There is 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 a semicolon.

Following the semicolon which ends the nodes are the edges, which are pairs of indices into the nodes. The pairs of numbers may be separated by an optional comma. The edges are terminated by a semicolon.

Following the edges are the faces, which are lists of indices into the nodes. Each lists 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 no predefined limit on the number of vertices in each face.

Between the last vertex of the list 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.

Comments may appear anywhere in the file and are delimited by curly braces. An example psurf is shown in Figure 8.2.

The indices listed in the psurf file all start at 1 for historical reasons. Beware that this can causes some confusion, since internally the indices start at 0, since they are stored in arrays.

8.3 Csurfs

Psurfs files may contain several "psurfs" of the above syntax. You can achieve this by concatenating several psurf files into one. Each separate sub-psurf is called a *csurf*. This name is a historical relic.

The advantage of doing this is that each the faces in each csurf refer to nodes only in that psurf. Sometimes, it can be much easier to generate psurfs in this format. It is also more efficient to do so, because the psurf edge genertation routine takes advantage of the fact that each csurf is a separate, disconnected unit.

The csurf information is store internally in the psurf in the *csurfs* field. The csurf data structure is:

```
typedef struct csurf Csurf;

struct csurf {
    short       nnodes;
    short       nedges;
    short       nfaces;
};
```

Each structure gives the number of nodes, edges, and faces which that csurf contains. The ordering of the array of csurf structures determines which nodes, edges, and faces belong to which csurf.

This information is kept only in the form of these tables. The face vertices and other indices are resolved to the global arrays when they are read in, to the indices themselves do not reflect the face that they came from different csurfs. It is the responsibility of any routine which needs the csurf information to adjust the indices accordingly.

8.4 How Psurfs are Read from Files

The principal routine for reading psurfs from files is `readpsurf`, defined in `gen/src/lib/psurf_rp.y`.

```
Psurf *
readpsurf(filename,archive,parentdir,scale)
char     *filename;
char     *archive;
char     *parentdir;
float    scale[];
```

In *Jack*, this routine is called only from within the peabody parser, which is where the elements of the peabody data structure are created, as they are read in from a file. The *filename* argument gives the file name as it occurred in the peabody line

```
psurf = "cube.pss";
```

The *archive* is the archive for the segment's figure, in which the psurf can reside. The *parentdir* argument is the directory of the peabody file in which the psurf reference was made. The scale parameter is the scale given on the psurf line. The scale is discussed below.

The `readpsurf` routine first looks for the file in the current directory, i.e. the one in which the *Jack* program was run. It looks for a binary psurf file first. If it finds both a binary psurf file and a text file, it compares the date of modification to make sure that the binary psurf file was generated since the last modification of the text file. If the binary psurf is out of date, it issues a warning message and reads the text psurf.

If the file is not in the current directory, then it looks for the archive in the current directory, doing the same search for the binary and text forms. If no archive exists, then it looks in the parent directory, first for the file, then for the archive. If it cannot be found there, it looks to see if the file is installed.

8.5 Psurf Utilities

Psurfs may be transformed by a homogeneous transformation with `transformpsurf`. This routine transforms the coordinates of each node and rotates the face and node normals by the rotation part of the transformation. This function is defined in `gen/src/lib/psurf_util.c++`.

```
transformpsurf(psurf,M)
Psurf   *psurf;
Matrix  M;
```

The function `PsurfInertia` computes the inertial properties of the shape of the psurf. These operations work on any shape of psurf, convex or not. The *density* is an input parameter, in grams per cm^3. The output mass is in grams, and the volume is in cm^3. The center of mass is in centimeters, and the inertia tensor is in gcm^2. This function is defined in the source code file `gen/src/lib/psurf_inertia.c++`.

```
PsurfInertia(psurf,density,volume,mass,centerofmass,inertia)
Psurf   *psurf;
double  density;
double  *volume;
double  *mass;
double  centerofmass[3];
double  inertia[3][3];
```

8.6 Distance Measuring Utilities

The following functions measure distances from points, lines, and planes in space to parts of a psurf.

```
int
pt_to_face_distance(psurf,f,p,d,x,item)
Psurf   *psurf;
short   f;
Vector  p;
float   *d;
Vector  x;
short   *item;


int
line_to_face_distance(psurf,f,p,v,d,x,item)
Psurf   *psurf;
short   f;
Vector  p;
Vector  v;
float   *d;
Vector  x;
short   *item;


int
plane_to_face_distance(psurf,f,r,n,d,x,item)
Psurf   *psurf;
short   f;
Vector  r;
Vector  n;
float   *d;
Vector  x;
short   *item;


int
pt_to_edge_distance(p,head,tail,d,x)
Vector  p;
Vector  head;
Vector  tail;
float   *d;
Vector  x;


int
line_to_edge_distance(p,v,head,tail,d,x)
Vector  p,v;
Vector  head,tail;
float   *d;
Vector  x;
```

The function names describe the distances they measure. `pt_to_face_distance` returns the distance to the point p. `line_to_face_distance` returns the distance to the line defined by reference point p and direction v. `plane_to_face_distance` returns the distance to the plane defined by reference point r and normal n.

Each of these functions operates in the local coordinate frame of the psurf. If you use the functions on psurfs associated with peabody segments, transform the coordinates of the point, line, or plane to the local coordinate frame of the segment before measuring the distance. This is more efficient than transforming the coordinates of the psurf to the coordinate frame of the point, line, or plane.

The distances are along the shortest path, and the point on the face which lies closest is returned in x. The distance is returned in d. x may lie in the interior of the face, along an edge of the face, or at a vertex of the face. This is specified by the return value of the function, together with the *item* argument.

The return values are as follows:

0 The point, line, or plane lies inside the face, so $d = 0$.

1 The closest point x lies in the interior of the face.

2 The closest point x lies along an edge of the face. The pair of vertices is (*item*,*item*+1)%nvertices). Note that the *item* field is *not* an index of a psurf edge, but is rather an index into the face's vertex array.

3 The closest point x is a vertex. referenced by *item*.

The `_to_edge` functions return the distance to a line segment, defined by the two vectors *head* and *tail*. They return 0 if the closest point is the *head* endpoint, 1 if the closest is the *tail* endpoint, and 2 if the closest point in in the interior of the line segment. In each case, the functions place the coordinates of the closest point in x and the distance in d.

The function `rayfaceintersection` tests whether a ray intersects with a face of a psurf.

```
int
rayfaceintersection(x,p,v,psurf,face)
Vector  x;
Vector  p;
Vector  v;
Psurf   *psurf;
short   face;
```

The ray is defined with starting point p and direction v If it intersects, it returns the intesection point in x, and the return +1 or -1 depending upon whether the ray intersects the face in the $+v$ or $-v$ direction from p. Otherwise, the function returns 0.

Like the distance routines above, this function operates in the local coordinate frame of the psurf. If you use this function on psurfs associated with peabody segments, transform the coordinates of the ray to the local coordinate frame of the segment before doing the intersection test. This is more efficient than transforming the coordinates of the psurf to the coordinate frame of the ray.

8.7 Writing Psurfs

Psurfs may be printed in a format readable by `readpsurf` in several different ways. The routines for printing psurfs are defined in the file `gen/src/lib/psurf_print.c++`.

```
xwritepsurf(psurf,filename,verbose,offset)
Psurf    *psurf;
char     *filename;
Boolean  verbose;
short    offset;
```

```
fwritepsurf(psurf,file,verbose,offset)
Psurf    *psurf;
FILE     *file;
Boolean  verbose;
short    offset;
```

`xwritepsurf` opens a file and writes the psurf to it. `fwritepsurf` writes the psurf to a standard I/O stream. The *verbose* and *offset* arguments control the format of the printing. Sometimes, it is helpful to print information about a psurf, such as the connectivity information and surface normals. This is called a *verbose* printing, and is produced when *verbose* is `TRUE`. The extra information is placed in comments, so the resulting file is readable by `readpsurf`.

Also, the discrepancy between the internal indices of a psurf and its textual description can cause great confusion during debugging. Sometimes it is convenient to print a psurf *as is*, with the indices starting at 0 instead of 1. This can be controlled with the *offset* argument, which specifies the offset index of the nodes. 1 is the default; 0 is helpful for debugging. Printing a psurf with nodes starting a 0 is only good for diagnostics, since it is not readable!

For convenience, there are macros for calling these two functions with the different arguments:

```
#define printpsurf(p)        fwritepsurf(p,stdout,FALSE,1)
#define printpsurf0(p)       fwritepsurf0(p,stdout,FALSE,0)
#define vprintpsurf(p)       fwritepsurf(p,stdout,TRUE,1)
#define vprintpsurf0(p)      fwritepsurf(p,stdout,TRUE,0)
#define writepsurf(p,f)      xwritepsurf(p,f,FALSE,1)
#define writepsurf0(p,f)     xwritepsurf(p,f,FALSE,0)
#define vwritepsurf(p,f)     xwritepsurf(p,f,TRUE,1)
#define vwritepsurf0(p,f)    xwritepsurf(p,f,TRUE,0)
#define fwritepsurf0(p,f)    fwritepsurf(p,f,FALSE,0)
#define vfwritepsurf(p,f)    fwritepsurf(p,f,TRUE,1)
#define vfwritepsurf0(p,f)   fwritepsurf(p,f,TRUE,0)
```

When a psurf is written out, it is written as a number of csurfs, in the form in which it was read in. If you want to collapse all csurfs into now, just change the counters in the first csurf to include all nodes, edges, and faces in the psurf.

8.8 Modifying Psurfs

Modifying psurfs by adding or deleting nodes, edges, or faces is difficult because of the tabular internal representation. The psurf structure was originally designed as a space-efficient representation for geometric objects, without too much concern for the ease of modifying data structure once it has been read in from a file.

There is, however, a simple mechanism for changing psurfs. This involves making changes only to the basic set of fields in the psurf and then using the routine `zappsurf` to erase the other fields which depend on the basic ones. It is important to use this mechanism whenever you write an application which changes a psurf internally.

```
zappsurf(psurf,docsurfs)
Psurf    *psurf;
Boolean  docsurfs;
```

The psurf data structure maintains information about its current status in several internal flags. These flags make the storage and retrieval of information from binary psurf files very efficient. It is important that changes in the psurf structure be reflected in these internal flags, it is important to change the psurf structure only through this mechaism.

The psurf is completely defined in terms of the following fields:

nnodes The number nodes. This gives the length of the arrays in the *nodes* structure.

nodes.coords The coordinates of the nodes.

nodes.flags The array of flags associated with the nodes.

nfaces The number faces. This gives the length of the arrays in the *faces* structure.

faces.nvertices The array specifying the number vertices in each face.

faces.vertices The array pointing to the vertex arrays for each face.

faces.attributes The array specifying the attribute index of each face.

faces.flags The array of flags associated with the faces.

Any changes to the psurfs, in terms of creating or deleting nodes, edges, or faces, should explicitly modify these fields and ensure that they are properly allocated and filled in. After doing so, you should call **zappsurf**. The other fields of the psurf will then be recomputed as they are requested by **getpsurffields**.

Chapter 9

The VEC Library

The ^library implements many simple mathematical operations which are common to many applications. It implements operations on vectors, 4 × 4 matrices, and quaternions, as well as some miscellaneous functions which don't really belong anywhere else.

The type declarations and macros for this facility are defined in the include file `vec.h`. `vec.h` includes `stdio.h`, `math.h`, and the IRIS Graphics Library file `gl.h`. Source code files which include `vec.h` thus need not include these files.

9.1 Macros

The following macros are defined in `vec.h`:

`DOT(u,v)`	Dot product of 3-vectors u and v.
`DOT4(u,v)`	Dot product of 4-vectors u and v.
`ABS(x)`	Absolute value of x.
`MIN(x,y)`	Minimum of x and y.
`MAX(x,y)`	Maximum of x and y.
`SIGN(x)`	Sign of x: $+1$ or -1.
`MAG(u)`	Magnitude of 3-vector u.
`RTOD(r)`	Convert radians r to degrees.
`DTOR(d)`	Convert degrees d to radians.
`INTERP(x,y,t)`	Interpolated value between x and y, according to t.
`CLAMP(x,min,max)`	x, clamped to interval (min,max): $\begin{cases} min & \text{if } x < min \\ x & \text{if } min \le x \le max \\ max & \text{if } max < x \end{cases}$

`INTERP` returns x if $t = 0$, or y if $t = 1$. To remember this, think of a "t" numberline with x on the left at 0 (it's the argument on the left) and y on the right at 1, (it's the argument on the right).

The following macros are defined for "fuzzy" comparison:

`FUZZ`	Standard fuzz factor: 0.0001
`EQ(x,y)`	Fuzzy equal: true if x and y are within $FUZZ$
`LEQ(x,y)`	Fuzzy less than or equal

`GEQ(x,y)`	Fuzzy greater than or equal
`VECEQ(u,v)`	Fuzzy comparison of 3-vectors
`FEQ(x,y,f)`	Fuzzy equal, given fuzz factor *f*
`FLEQ(x,y,f)`	Fuzzy less than or equal, given fuzz factor *f*
`FGEQ(x,y,f)`	Fuzzy greater than or equal, given fuzz factor *f*.
`FVECEQ(u,v,f)`	Fuzzy comparison of 3-vectors, given fuzz factor *f*

9.2 Vectors

The `vec.h` include file defines the type `Vector`

```
float Vector[3];
```

This type is used interchangably with floating point arrays of length three. It is not always used consistently in the *Jack* source code, but it is sometimes convenient.

Vectors are arrays of floating point numbers. There are routines in the library for adding, subtracting, multiplying, and crossing 3-vectors. There is no explicit type declaration for vectors: the routines operate on floating point arrays. The following operations are defined in the file **gen/src/lib/vec_vector.c++**.

```
crossproduct(r,u,v)
float  r[3];
float  u[3],v[3];


vecadd(r,u,v)
float  r[3];
float  u[3],v[3];


vecsub(r,u,v)
float  r[3];
float  u[3],v[3];


vecinterp(r,u,v,t)
float  r[3];
float  u[3],v[3];
float  t;


vecscalarmult(u,v,f)
float  u[3],v[3];
float  f;
```

```
unitize(u)
float  u[3];


cpvector(u,v)
float  u[3],v[3];


Boolean
zerovec(u)
float    u[3];
```

Each of these functions takes arguments in the order of an assignment statement: the first argument is the result and the remaining arguments are taken left to right. `unitize` makes a vector into a unit vector. `cpvector` copies the 3-vector v to u. It is a macro, not a function. `zerovec` returns whether or not u is the zero vector, using fuzzy comparison. `vecinterp` interpolates between the two vectors. If $t = 0$, then $r = u$; if $t = 1$, then $r = v$. This is the same sense as the `INTERP` macro. In the actual arguments to `crossproduct`, r should not be the same vector as u or v.

The following routines for transforming vectors are defined in **gen/src/lib/vec_vector.c++**.

```
vecmult(r,u,M)
float   r[3];
float   u[3];
Matrix  M;


vecmult0(r,u,M)
float   r[3];
float   u[3];
Matrix  M;


vecmult4(r,u,M)
float   r[4];
float   u[4];
Matrix  M;


vecrot(p,q,axis,refpt,angle)
float  p[3];
float  q[3];
float  axis[3];
float  refpt[3];
float  angle;
```

`vecmult` multiplies u by M, using a 1 in the fourth position, thereby performing translation. It performs the following computation:

$$[r_0\ r_1\ r_2\ *] = [u_0\ u_1\ u_2\ 1] \begin{bmatrix} x_0 & x_1 & x_2 & 0 \\ y_0 & y_1 & y_1 & 0 \\ z_0 & z_1 & z_2 & 0 \\ p_0 & p_1 & p_2 & 1 \end{bmatrix}$$

`vecmult0` performs the multiplication using a 0 in the fourth position of u, thus it does *not* perform translation. Remember that vectors are represented here as *rows*, so the matrices have the translational part in the bottom row! None of the vector multiplication routines make any assumptions about the contents of the matrices. In the actual arguments to each of these functions, r may be the same vector as u. Hence, to transform a vector "in place," it is legal to say:

```
vecmult(a,a,M);
```

`vecrot` rotates the vector q around the axis defined by the direction *axis* and the reference point *refpt*, through an angle *angle*, given in radians. If *refpt* is null, then the origin $(0,0,0)$ is used. In the actual arguments, p and q may be the same vector.

9.3 Matrices

The ^library uses the IRIS Graphics Library's typedef for a matrix, `Matrix`, which is a 4×4 array of floating point numbers, represented with the translation component along the bottom row:

$$\begin{bmatrix} x_0 & x_1 & x_2 & 0 \\ y_0 & y_1 & y_1 & 0 \\ z_0 & z_1 & z_2 & 0 \\ p_0 & p_1 & p_2 & 1 \end{bmatrix}$$

This representation requires the interpretation of a product of matrices right to left in local coordinates, or left to right in absolute coordinates. Thus, the transformation `M` defined as

$$\mathbf{M} = \mathbf{T_1} * \mathbf{T_2} * \mathbf{T_3}$$

may be interpreted as:

1. A transformation by $\mathbf{T_1}$ with respect to the base coordinate frame, followed by a transformation by $\mathbf{T_2}$ w.r.t. to the base frame, followed by a transformation by $\mathbf{T_3}$ w.r.t. to the base frame.
2. A transformation by $\mathbf{T_3}$ with respect to the base coordinate frame, followed by a transformation by $\mathbf{T_2}$ w.r.t. to the *transformed* frame, followed by a transformation by $\mathbf{T_1}$ w.r.t. to the *doubly* transformed frame.

This sense of the ordering of the transformations is maintained internally as well as externally in the syntax of the transformations in the peabody language.

The ^library has a declaration of an identity matrix, called `idmat`:

```
extern  Matrix  idmat;
```

This matrix may be used, but should *never* be modified!

The following operations are defined in the file `gen/src/lib/vec_matrix.c++`.

```
matmult(r,a,b)
Matrix  r;
Matrix  a,b;
```

```
invertmatrix(r,m)
Matrix  r,m;


cpmatrix(a,b)
Matrix  a,b;


eqmat(a,b)
Matrix  a,b;


printmat(r)
Matrix  r;


fprintmat(file,r)
FILE    *file;
Matrix  r;
```

`matmult` performs the matrix product `r = a * b`. `cpmatrix` copies *b* to a. It is defined as a macro, not a function. `invertmatrix` inverts an arbitrary 4 × 4 matrix. `eqmat` returns whether or not the two matrices are equal, using fuzzy comparison. The right column of the matrix is ignored, since it is typically 0. `printmat` prints a matrix to standard output, as 4 lines of 4 floating point numbers. `fprintmat` does the same thing but prints to a file.

There are also some miscellaneous routines for constructing matrices. These functions are also defined in the file `gen/src/lib/vec_matrix.c++`.

```
xyztomatrix(M,x,y,z)
Matrix  M;
float   x,y,z;


transtomatrix(M,x,y,z)
Matrix  M;
float   x,y,z;


axisangletomatrix(M,axis,angle)
Matrix  M;
float   axis[3];
float   angle;
```

`xyztomatrix` constructs a matrix given by a rotation of *x* around the *x*-axis, followed by a rotation of *y* around the rotated *y*-axis, followed by a rotation of *z* around the rotated *z*-axis. `transtomatrix` constructs a translation matrix out of *x*, *y*, *z* components. `axisangletomatrix` constructs a matrix which rotates around *axis* through an angle *angle*, given in radians.

There are also routines for converting from a matrix to other representations:

```
mattorot(angles,type,M)
float   angles[3];
char    type[3];
Matrix  M;
```

`mattorot` converts the rotation part of M to euler angles. The *type* argument specifies the euler axes. It is a character string which should be three characters long. Each character must be an `x`, `y`, or `z`. The legal values are: `"XYZ"`, `"ZXY"`, `"YZX"`, `"XZX"`, `"ZXZ"`, `"YXZ"`, and `"ZYX"`. The values filled into the *angles* array are the rotations around each of the given axes such that the product of the three rotations gives M. The results will be incorrect if M is not homogeneous.

9.3.1 Homogeneous Transformations

A homogeneous transformation is a product of simple rotations and translations. Homogeneous transforms have some very nice properties. `vec.h` defines the type `Transform` as:

```
typedef union   transform   Transform;

union transform {
    Matrix  matrix;
    struct {
        float   x[4];
        float   y[4];
        float   z[4];
        float   p[4];
    } v;
};
```

The structure of vectors overlaps the matrix so that the vectors correspond to the rows of the matrix. When a transform is considered as a local coordinate frame, the x vector lies along the x axis of the local coordinate frame, the y vector lies along the y axis of the local coordinate frame, the z vector lies along the z axis of the local coordinate frame, and the p vector lies at the origin of the local coordinate frame. This structure is defined so that these vectors may be easily associated with the transform.

There are also several routines which deal with matrices as homogeneous transformations, which are matrices formed by the product of simple rotations and translations. These functions are defined in the file `gen/src/lib/vec_matrix.c++`.

```
hmatmult(r,a,b)
Matrix  r;
Matrix  a,b;


inverthomomatrix(r,m)
Matrix  r,m;
```

```
ishomo(r)
Matrix  r;


homogenize(x,y,z)
float  x[3],y[3],z[3];


linterpmatrix(r,a,b,t)
Matrix  r;
Matrix  a,b;
float   t;
```

hmatmult multiplies two homogeneous transforms. This is a highly-optimized routine which takes advantage of the special structure of the homogeneous transform, and it is much more efficient than **matmult**. **inverthomomatrix** computes the inverse of a homogeneous transform. If the matrix is not a product of simple rotations and translations, the inverse will not be correct, but no warning will be issued. The function **invertmatrix** inverts an *arbitrary* 4 × 4 matrix, not just a homogeneous one. However, if you know a matrix to be homogeneous, use **inverthomomatrix**, since it is many times more efficient.

ishomo returns **TRUE** if the matrix is a homogeneous transform, **FALSE** if not.

The function **homogenize** is useful for constructing the three perpendicular vectors necessary to describe a homogeneous transform. The function takes x and y as inputs, computes z as their cross product, then computes y as the cross product of z and x. Finally, all three vectors are unitized. Thus, the direction of x remains unchanged, z is the cross product of x and y, and y is "corrected" to be perpendicular to x and z.

The function **linterpmatrix** linearly interpolates between the two matrices a and b, according to the parameter t, which is between 0.0 and 1.0. Like the **INTERP** macro, If t=0, then r=a; if t=1, then r=b. This function linearly interpolates the translation component, and it linearlyy interpolates the orientation component around the single axis of revolution which rotates a into b.

9.4 Quaternions

Quaternions are useful for representing and manipulating rotational transformation. **vec.h** defines a quaternion as a 4-vector of floats, (w, x, y, z), where w is the angle part and (x, y, z) is the axial part. The operations here are an implementation of the equations described in "Animating Rotations with Quaternion Curves," by Ken Shumake[1].

```
typedef float Quaternion[4];
```

The following operations on quaternions defined in the source code file **gen/src/lib/vec_quaternion.c++**.

```
qmult(q,a,b)
Quaternion  q;
Quaternion  a,b;
```

[1] Ken Shumake, "Animating Rotations with Quaternion Curves," *Computer Graphics*, Vol. 19, No. 3. (1985) pp. 245-254.

```
qinverse(q,a)
Quaternion  q;
Quaternion  a;


slerp(q,a,b,t)
Quaternion  q;
Quaternion  a,b;
float       t;


qdouble(q,a)
Quaternion  q;
Quaternion  a;


qbisect(q,a,b)
Quaternion  q;
Quaternion  a,b;


matrixtoq(q,M)
Quaternion  q;
Matrix      M;


qtomatrix(M,q)
Matrix      M;
Quaternion  q;
```

`qtomatrix` and `matrixtoq` convert between rotation matrices and quaternions.

9.5 Intersections of Planes and Lines

These routines compute simple intersections of planes and lines. They are defined in the source code file `gen/src/lib/vec_intersect.c++`.

```
Boolean
planelineint(x,n,r,p,v)
float    x[3];
float    n[3],r[3];
float    p[3],v[3];


Boolean
commonnormal(x,y,n,p,u,q,v)
float    x[3],y[3],n[3];
float    p[3],u[3];
float    q[3],v[3];
```

planelineint determines the intersection of the plane defined by normal n and reference point r with the line defined by direction v and reference point p. The intersection is returned in x. The function returns **FALSE** if the line is parallel to the plane. **commonnormal** determines the common normal between the two lines defined by reference points p and q with directions u and v, respectively. The intersection of the common normal with the p line is returned in x, and the intersection with the q line is returned in y. The common normal itself is returned in n, as a unit vector. u and v need not be unit vectors. If the lines are parallel, then $x = p$, and y is set accordingly. The function returns TRUE if the lines are parallel, FALSE otherwise. If the lines intersect, then x and y are equal.

9.6 Lists

Lists are collections of objects. There are routines in the library for manipulating circular linked lists which contain arbitrary pointers to objects. The fact that the lists are circular should have no impact on the applications (except that insertion and appending are both $O(1)$ operations). A list is defined by the type **List**. The actual contents of the structure is unimportant. Notice that the use of the **List** type does *not* require an asterisk, except to refer to the *address* of list list.

The following functions construct lists. They are defined in the source code file **gen/src/lib/vec_list.c++**.

```
void
appendcirclist(p_list, p_data)
List   *p_list;
int    *p_data;


void
insertcirclist(p_list, p_data)
List   *p_list;
int    *p_data;


void
addcirclist(p_list, p_data)
List   *p_list;
int    *p_data;
```

appendcirclist appends *p_data* to the list pointed to by *p_list*. An appended item becomes the last item in the list. Note that a pointer to the list must be passed as well as a pointer to the data. **insertcirclist** inserts *p_data* in the list pointed to by *p_list*. An inserted item becomes the first item in the list. The item that was previously the first in the list becomes the second item. Note that a pointer to the list must be passed as well as a pointer to the data. The function addcirclist appends the data element *p_data* to the list ***only if it isn't already there.*** This is convenient when you need to make sure a list doesn't contain duplicate entries for the same data element.

The function **circlistiterator** is used to loop over a list. It is used in conjuction with macro **LISTDATA**, which casts the data field of the list to the appropriate type. This function, when called repeatedly, successively returns each node in *list*. It must be called with the list to be iterated over and the last return value. The first time this function is called, *lastreturn* should be NULL. This function will return NULL when all items have been returned. The macros **L_NEXT** and **L_PREV** return the next and previous elements in a list, for cyling over the list explicitly.

```
List
circlistiterator(list, lastreturn)
List  list;
List  lastreturn;

L_NEXT(list)

L_PREV(list)

LISTDATA(type,list)
```

Figure 9.1 shows an example of the list operations. It is important to make sure that you initialize the list variable, both before inserting information into the list, and before looping over the list. The diagram at the bottom of Figure 9.1 shows the layout of the list as it would be constructed by the operations in the code segment. It also shows where in the list an appended or inserted value will go, and which element is returned first by `circlistiterator`.

```
Segment *segment,*A,*B,*C;
List    l;

list = 0;
appendcirclist(&list,A);
appendcirclist(&list,B);
insertcirclist(&list,C);

l = 0;
while ((l=circlistiterator(env->segments,l))) {
    segment = LISTDATA(Segment,l);
    printf("%s\n",segment->fullname);
}
```

Figure 9.1: `circlistiterator`

The function `circlistlength` returns the length of a list. The function `sortcirclist` sorts a list, according to a comparion function *cmp*. This function takes a pointer to two list data elements and returns -1 if the first is less than the second, 0 if the two are equal, and 1 if the second is greater than the first.

```
int
circlistlength(list)
List  list;

sortcirclist(list,cmp)
List  list;
int   (*cmp)();
```

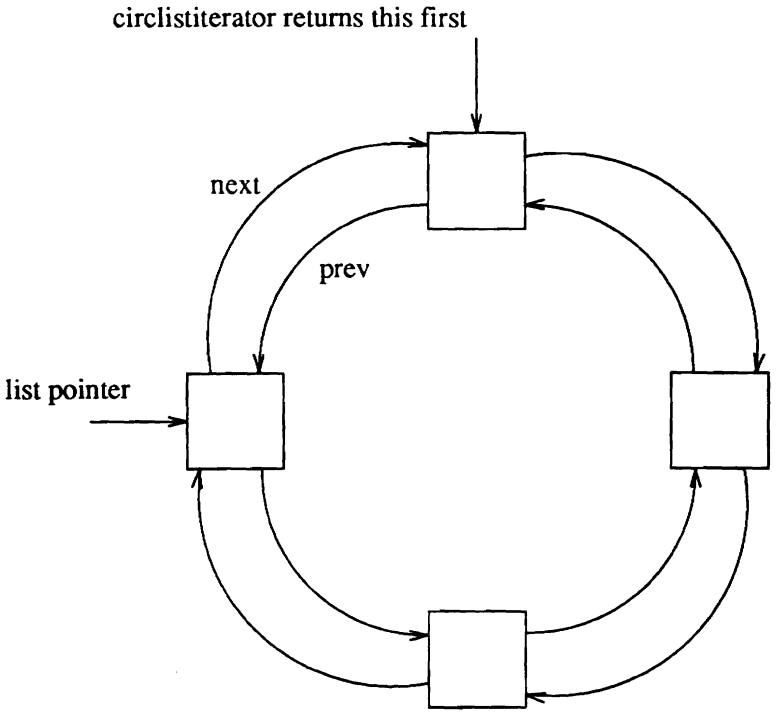

Figure 9.2: Circular Lists

There are also routines for deleting entire lists and single elements from lists.

```
deletecirclistdata(p_list,p_data,p_cmp_f,p_del_f)
List   *p_list;
int    *p_data;
int    (*p_cmp_f)();
int    (*p_del_f)();


killcirclist(p_list, p_del_f)
List   *p_list;
int    *p_data;
int    (*p_del_f)();
```

For the function `deletecirclistdata`, the argument *p_cmp_f* is a comparison function used to locate the node to delete. It takes a pointer to two list elements, and should return 1 if the element matches *p_data*, 0 otherwise. Note that it need not compare all fields of the list data. If this function is `NULL`, then the function just searchs for the list element whose data pointer is *p_data*. The argument *p_del_f* is a function to invoke on the data element as the list entry is deleted. This function typically frees the data associated with the entry. If this function is `NULL`, then it is not executed.

9.7 Timestamps

Jack uses the idea of a *timestamp* to record when certain events have occurred. Several of the data structures contain *timestamp* fields. The values of these fields are set by the function `gettimestamp`.

gettimestamp()

This function returns an integer. Each time the function is invoked in returns a value greater than the previous invokation. When stored in the appropriate places, these values may be compared to see when certain events took place relative to other events. This is not a true notion of *time* because it has no relation to a clock, but it is a useful mechanism for determining when certain values need to be updated.

An example of how this is used is with the psurf display lists, described in Section 8.1.4. The display lists contain information about the psurf geometry which can be passed quickly to the IRIS Graphics Library. This information includes color information, although the actual color of the psurf is kept inside an **attribute** structure pointed to by the psurf. If the attribute parameters change, the psurf display list must be updated. When the parameters of an attribute change, *Jack* sets the *timestamp* field in the attribute structure, rather than searching for psurfs which refer to the attribute and updating them immediately. The psurf display list also has a timestamp, which is updated when the display list is generated or modified. Setting the timestamp for the attribute forces the psurf display list to be "out of date" with respect to the attribute. The routine which actually draws the display list compares the timestamps of the display list and the attributes to which the psurf refer. If the display list is out of date, it updates it before drawing it.

9.8 Miscellaneous Utilities

The **vec** library also contains some miscellaneous routines and declarations of general usefulness which don't properly belong anywhere else.

9.8.1 Strings

The following macros are defined for string manipulation:

isvowel(c)	true if *c* is a vowel
CONTROL(c)	control character corresponding to *c*
SLENGTH(s)	the length of a string formatted with **sprintf**

The **isvowel** macro determines whether a character is a vowel. The **CONTROL** macro provides a way of nicely specifying a control character, in terms of its "partner" alphabetic character. For example, **CONTROL('C')** is **^C**.

The **SLENGTH** macro is useful since the System V (IRIS) version of the function **sprintf** returns the number of characters formatted, while the BSD version returns a pointer to the resulting string. This macro makes it convenient to advance a pointer through a buffer while printing into it, and have the same code work in both environments. An example of the use of **SLENGTH** is shown in Figure 9.3.

9.8.2 Memory Allocation

Most memory allocation is done in allocating space for structures, particularly for linked lists. There is a macro **talloc** which allocates memory and casts it as a pointer to a given type:

```
char    buf[1024];
int     i;

n = SLENGTH(sprintf(buf,"hello, world:  "));
for (i=0; i<3; i++) {
    n += SLENGTH(sprintf(buf+n,"%d ",i));
}
n += SLENGTH(sprintf(buf+n,"good bye %d now\n",4));
```

Figure 9.3: SLENGTH

```
#define talloc(type,n) (type *) calloc(n,sizeof(type))
#define trealloc(type,ptr,n) (type *) realloc(ptr,n,sizeof(type))
```

Index