sun® microsystems

NeWS™ Application Scenario

Contents

Preface

This manual serves as a companion volume to the *NeWS Manual.*

Prerequisites

Before you begin working with this manual, you should:

- have a good understanding of the POSTSCRIPT and C languages,
- have read the *NeWS Technical Overview.*

Companion Documents

You should have on hand copies of both POSTSCRIPT language books (the *PostScript Language Reference Manual* and the *PostScript Language Tutorial and Cookbook*) and the *NeWS Manual.* There are quite a few references to POSTSCRIPT language primitives and many of the NeWS operators.

Where to Start

If you haven't started up NeWS yet, the *Using NeWS* appendix of the *NeWS Manual* will tell you how.

If you want to preview some of the examples in the *PostScript Language Tutorial and Cookbook*, use the NeWS previewer, `psview`. Instructions on its use may be found on the appropriate manual page in the *NeWS Manual.*

If you have NeWS up and running and are reasonably familiar with the POSTSCRIPT language you can begin reading the rest of this manual. Chapters 1 and 2 are a quick introduction to your first steps in programming in the NeWS environment. With the material presented in those chapters, you will start drawing figures in a window almost immediately.

Structure of the Manual

This manual is organized as follows:

- A brief introduction of NeWS and a sample window for NeWS programming
- A extended example (across several chapters) which guides you through the building of an interactive client-server NeWS application (the `go` program)
- An appendix which describes NeWS programming conventions
- A second appendix which explains a number of special features that you might want to add to applications

In Chapter 4 and the succeeding chapters we take the `go` program first demonstrated in Chapter 3 and convert it into an application split across C client and NeWS server. The work in these chapters represents a gradual progression:

- Chapter 4 first splits the functionality of the `go` program and describes the general process of converting a POSTSCRIPT program for use with the CPS interface;
- Chapter 5 develops the ability to communicate with a specific process on the server side;
- Chapter 6 develops the mechanisms necessary to track mouse actions on the server side from the client side;

Source Code Listings

You will find that all the sample programs of this manual may be found either in `$NEWSHOME/clientsrc/app_guide/go` (for the `go` programs) or in `$NEWSHOME/clientsrc/app_guide/code` (for the code fragments and programs).

In addition, at the end of each chapter you will find a summary of the NeWS primitives used in that chapter.

Terminology

In this manual you will find references to procedures and routines, methods, functions:

- By *procedure* we mean a function written in the POSTSCRIPT language. It may or may not include NeWS operators.
- The term *routine* is interchangeable with *procedure.*
- By *method* we mean a procedure written in the POSTSCRIPT language that is inherited from the definition of a class.
- When you see the term *function* used it means C language functions, accessible to the client side C program.

Font Usage

The *NeWS Application Scenario* includes code and procedures from the C and POSTSCRIPT languages. In order to minimize confusion, we have used fonts to clarify which language is used:

`bold listing font`
This font indicates things that you should type at your workstation.

`listing font` This font indicates literal values such as file names and output displayed by the computer. It also indicates use of the *C* language: it is used in C program listings and C procedure names. CPS routines and code fragments such as `ps_open_PostScript()` are printed in this font.

sans serif font This font is used for *POSTSCRIPT* program listings, types and code fragments such as 300 200 createcanvas mapcanvas to distinguish them from C code. It is also used in the definition of NeWS functions (primarily in Chapter 12 of the *NeWS Manual, NeWS Operator Extensions.*

bold font Unfortunately, sans serif fonts look poor in the middle of normal text. So, as well as indicating cautions and warnings, bold font is used to indicate all NeWS names, such as **clipcanvas**, when they appear in paragraphs or the index.

italic font

This font is used as a place holder for words, numbers, or expressions that you define, for example parameters to commands, and operands of POSTSCRIPT language operators. Italics are also used in the conventional manner to emphasize important words and phrases.

1

Introduction

1 Introduction

This manual shows you some of the fundamentals of NeWS. When you finish, you should be able to develop a NeWS application that uses the basics: canvases, processes, events and interests. The application developed in this manual uses all of these.

This chapter serves as an overview, presenting brief explanations of the critical components and terminology of NeWS. Detailed explanations of these mechanisms may be found in the NeWS manual.

1.1. The NeWS Server

NeWS applications can be written completely in the POSTSCRIPT language. These applications will run within the NeWS server, which acts as an interpretive engine.

Look at the programs provided in `$NEWSHOME/demo` for examples of NeWS applications designed to run wholly within the server.

Formerly, if you wanted to use the POSTSCRIPT language you had to use a print engine. NeWS implements a large subset of the POSTSCRIPT language and creates an environment in which you can use the POSTSCRIPT language (and the NeWS extensions to it) interactively.

Most applications, because of the considerations discussed below, will have a client and a server side.

1.2. Application Functionality

This manual provides you with a model of how to split an application between the NeWS server and the client (C) side. As a general rule of thumb, POSTSCRIPT language engines are best suited to simple user interaction routines and drawing operations. Too many sophisticated calculations within the server hurts performance. Thus, you gain a significant benefit by splitting an application between the server and client sides, allowing the client side to take on the heavy computation. The client side can be written in any number of languages. Sun currently supports the use of the C language with a 'C to POSTSCRIPT' (CPS) interface[1].

1.3. NeWS Explained

The following paragraphs explains briefly some of the most vital structural elements of NeWS. If you're in a hurry (and are familiar with object-oriented programming) they may provide you with enough information that you can try programming right away.

[1] See Chapter 15 of the *NeWS Manual, Supporting NeWS From Other Languages.*

What Is a Process?

NeWS *processes* run in the server's *name space*. They are lightweight (in contrast to UNIX processes) because they all share a common address space. Thus, they need no kernel support and are very cheap to start up in terms of system resources.

When a child process is forked (for instance, whenever a menu is invoked) the child process inherits a copy its parent's dictionary stack. The child inherits certain behaviors from the parent with this dictionary stack. By manipulating the contents of this dictionary stack the child can selectively override behaviors and control how much of its name space it shares with other processes.

What Is a Class?

Classes are implemented in NeWS through the dictionary mechanism[2]. A class is a composition of both *instances* and *methods*. Instances may be likened to local variables and methods to procedures. The methods are NeWS procedures which operate on both the instance and class variables of the class. In object-oriented programming an *object* is comprised of both data and the necessary procedures to act upon that data. Objects are instances of classes in NeWS. They inherit their behavior (the methods) and data (instance and class variables) from their parent classes upon creation. An example of such an object is a NeWS *window* (see below).

What Is a Canvas?

A *canvas* is the basic mechanism for drawing on a display[3]. Canvases are surfaces upon which images may be drawn. The surface may be opaque, or transparent, and can be of *arbitrary* shape. In NeWS, you are not limited to rectangular drawing surfaces. Canvases may be virtually any shape. They may overlap on a display surface. A canvas may be created (and drawn on) while it is not displayed and then *mapped* to make it visible on the display.

Canvases are very lightweight objects – they are very easy to create and this can be done very rapidly (again with little cost in system resources).

What is a Window?

A *window* is a composition of canvases and a process called the *frame event manager*. A window is an object, created as an instance of a NeWS class. A default window is comprised of five canvases and the frame event manager. The canvases are: an opaque frame canvas (containing two buttons), two transparent button canvases (upper left and lower right), a transparent client canvas (where you normally draw images), and an opaque icon canvas (representing the closed window object). The frame event manager is a process created when the window is created. It manages operations within the context of the window (i.e., insures that something happens when you click a mouse button somewhere in the window).

All this information, in much greater detail, can be found in the *NeWS Manual.*

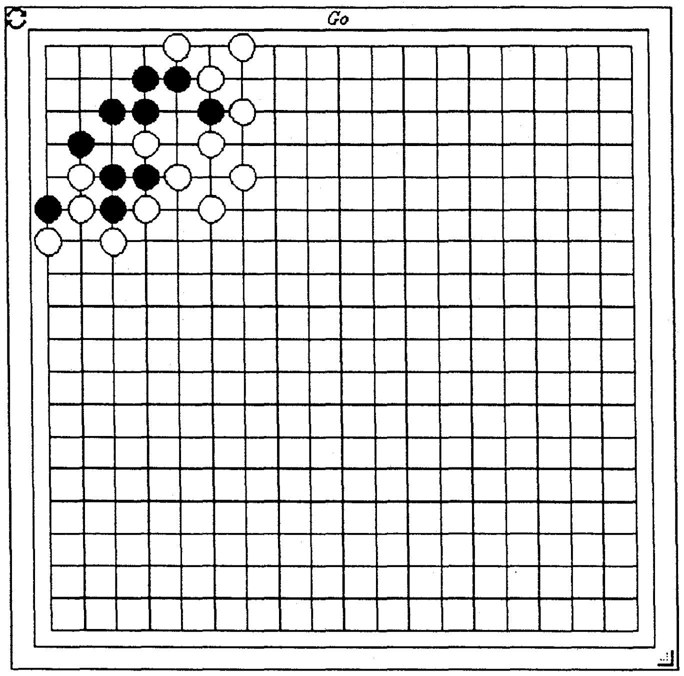
[2] For a good sample implementation, read Chapter 7 of the *NeWS Technical Overview*, *Writing a User Interface for NeWS*.

[3] We use the term *canvas* because *window* means many things to many people.

1.4. The Game of `go`

This explanation of how to write a NeWS application is developed around a simple `go` program[4]. `Go` is an ancient Japanese game of strategy. It uses a board which is 19 squares on a side. Each of the two players place pieces (stones) alternately on the intersections of the lines on the board. One player uses black stones, the other white. The objective of the game is to capture territory. This is accomplished by enclosing a group of your opponent's stones with a line of yours. There should be no open lines from within the encirclement. The edge of the board serves as an absolute boundary. The following image shows one such successful encirclement (by white).

Figure 1-1 *black loses a battle!*



The mechanisms used in implementing this application could very easily be adapted to any application that required mouse input and a display medium.

In fact, we hope that you will do just that.

1.5. Running Programs In This Manual

All the programs and program fragments in this manual have been included in this release. Each complete program may be found in either the `go` or `code` subdirectory of `$NEWSHOME/clientsrc/app_guide`. The file containing the program will have the same name as the figure[5].

[4] A good primer on the game of `go` is Volume 1 of the Elementary Go Series, *In the Beginning*, by Ikuro Ishigure. It is published by the Ishi Press of Tokyo.

[5] Only the `go` programs (`.cps` and `.c`) are in the `go` directory. All other programs are in the `code` directory.

This release of NeWS includes the psh program[6]. The NeWS server is a POSTSCRIPT language interpretive engine. psh establishes a connection to the server and allows you direct communication with the server. You may either provide psh with a file to run:

Figure 1-2 *running a program in* psh

```
babylon%  psh filename.ps
```

or start an interactive programming session with:

Figure 1-3 *starting an interactive session with* psh

```
babylon%  psh
executive
Welcome to NeWS Version 1.1
```

The **executive** command interposes an error handler between you and the NeWS server so that any errors that may occur are dealt with gracefully.

Now that you are directly connected to the NeWS server you may type in any combination of NeWS and POSTSCRIPT language commands that you want.

Pasting Selections

The code fragments and programs in this manual have been provided to you to help you with the process of trying out the examples. You can either copy the code fragments from one file to another using a text editor or you can "cut and paste" using the following technique:

- Create two windows (psterm emulators). Display the text that you wish to copy in one; the other emulator should either display the text file into which you will insert text or it should be running psh (which requires no insert mode).
- While in the target emulator (into which you will paste), depress and hold the L6 key. Note that the cursor should be in the target emulator when you do this.
- Still holding the L6 key down, move the cursor to the emulator from which you want to copy and select the text to be copied. Click the left mouse button, with the cursor held over the beginning of the text to copy and click the middle mouse button over the end of the text to copy. Your text is now selected (signified by an underbar under the selection).
- Move the cursor back to the target emulator and release the L6 key. The selected text will be pasted into the target emulator at the insertion point (or the current line if you're working in psh).

[6] See the *Using NeWS* appendix of the *NeWS Manual* for more detail on psh .

This method of pasting should prove helpful in adding example code to the working window program.

1.6. Flexibility

In large part, this `go` application serves as a demonstration of the flexibility and versatility of NeWS. We encourage you to approach your learning experience with a willingness to experiment. There are usually many ways to accomplish the same end. NeWS is made all the more powerful by the imagination of you, its users.

2

A Sample Working Window

2

A Sample Working Window

One of the first things a new user asks when introduced to a windowing system is, "Well, how do I draw something?" This is shortly followed by, "How do I draw it in a window?" This chapter answers these often frustrating questions. The NeWS program in Figure 2-1 creates a window within which you can draw an image. This program is a skeletal form which draws a window and then calls a routine to draw something in that window. Think of this as a "working window" — an easel upon which you can sketch[7].

2.1. The Window Program

Here is the basic working window that will serve as our model throughout this manual:

This *cliché* is both useful and common and is explained in some detail in Appendix A, *Conventions*.

Figure 2-1 *code/window.ps*

```
/makewin { % - => - (builds a test window)
   /win framebuffer /new DefaultWindow send def      % cliche
   {
      /PaintClient { .4 fillcanvas } def             % draw after damage
      /FrameLabel ( Workspace ) def                  % label window
   } win send                                        % execute contents of
                                                     %   {} in win's context

   /reshapefromuser win send                         % select size/position
   /map win send                                     % make visible
} def

makewin                                              % run the program
```

[7] The directory `$NEWSHOME/clientsrc/app_guide/code` contains the programs and program fragments listed in this chapter.

2.2. Invoking the Program

To execute the window program, retrieve it from the `code` directory and either paste the program body into `psh` or use the program name as an argument to `psh`:

```
babylon%  psh window.ps
```

2.3. The Window Program Explained

Let's take a look at this code in some detail. The first line of this program:

Figure 2-2 *a cliché*

```
/win framebuffer /new DefaultWindow send def
```

is what we choose to call a *cliché*. *Clichés* are common pieces of code which perform a commonly-needed function (in this case, creating a window to work in)[8]. Some are quite simple, others quite complex. This particular *cliché* is one of the more complex ones and so is explained in detail in Appendix A. You need not understand it at this point to use it. The pair:

```
/win ... def
```

assigns the window to win.

An object's class hierarchy is placed on the dictionary stack (creating a context) so that any methods used will evaluate references correctly.

We can use this variable to send instructions to be executed in the new window's class *context*.

/PaintClient is an instance variable within the **DefaultWindow** class hierarchy. NeWS uses this variable when re-drawing the client canvas after damage[9]. Whenever damage occurs, the procedure body identified with **/PaintClient** is executed[10]. Hence, we can use this by specifying our own procedure to be drawn:

Figure 2-3 *the* **/PaintClient** *routine*

```
{.4 fillcanvas}
```

In this case, painting the window a shade of gray in the event of damage.

/FrameLabel is another instance variable used by NeWS to determine what label to give a newly-created window. The contents of the **/FrameLabel** variable should be an ASCII text string. You may provide it by either placing a string in

[8] These windows are built using the LiteWindow package. Strictly speaking, these are not low-level NeWS operations.

[9] The interior drawing surface of the window is called the *client canvas*.

[10] For example, *damage* occurs when part of the window is obscured, or when a canvas is first mapped to the display. See the discussion in Chapter 2, *NeWS Extension Overview* of the *NeWS Manual*.

parentheses (as we have done):

```
/FrameLabel ( Workspace ) def
```

or by placing a procedure in braces (which delays execution). The procedure has to leave a text string on the operand stack after evaluation for this to work. The final:

```
{ ... } win send
```

sends the contents of the {}'s to the window win to be executed in its context.

The **/reshapefromuser** method is "sent" to win's class context and executed therein. Note that the **send** has to do with setting up the correct context, not sending a message to a process.

This method allows the user to shape a window at creation by positioning the cursor and clicking the left mouse button to fix the window corner. Then drag the cursor – the wire frame of the window will follow it. Click the left mouse button again to fix the window size.

The **/map** method simply makes the window visible after you have sized it. It is sent to win and executed in the same manner as **/reshapefromuser**.

2.4. Drawing An Image In the Sample Window

At this point, you should have some understanding of how our sample window works.

Now consider the task of drawing an image in the sample window.

Changing /PaintClient

The sample window can fill an image in the canvas rather than the whole canvas. To do this, you need to change two things in the sample program.

First, replace the procedure body (the code in braces following the **/PaintClient** variable) with a call to the POSTSCRIPT language routine that you would like executed. This is more complex than just calling the NeWS **fillcanvas** utility, so you will require a procedure rather than a simple operator. For example, say you wish to draw a simple arc in this window. Call this new routine /draw_arc:

```
/PaintClient {100 100 draw_arc} def
```

When **/PaintClient** variable is used it places 100, 100 on the stack and then executes /draw_arc.

Now you must actually add the /draw_arc routine to the window program body. The new procedure is as follows:

Figure 2-4 *code/sample2.ps*

```
/draw_arc { % x y => - (draws an arc at the specifed location)
      100 0 300 arc                  % radius start finish
      gsave                          % save graphics context
         .5 setgray fill             % set the path color and fill it
      grestore                       % restore graphics context
      stroke                         % draw the arc
} def
```

This code is relatively straightforward. You can see from the comments that the `100, 100` passed to `/draw_arc` on the stack determines the center point of the arc in the client canvas.

The `gsave ... grestore` pair saves then later restores the graphics context[11]. In this case, this means that the pair saves the path of the arc for later stroking. The:

```
.5 setgray fill
```

fills the current path with half-tone gray. The path of the arc is closed by the `fill` operator before being filled[12]. The `grestore` restores the path (because the `fill` consumes it) and then you `stroke` it with the default color (black); drawing the path in the client canvas of the window.

[11] See the *PostScript Language Reference Manual, Section 4.3* for a complete explanation of all the contents of the graphics state stack saved by this operation.

[12] See the *PostScript Language Reference Manual, Section 4.6* for an explanation of painting operations.

2.5. Sample Program Plus /draw_arc

Now, composing the new drawing routine with the sample window progam results in:

Figure 2-5 *code/arc.ps*

```
/draw_arc { % x y => - (draws an arc at the specifed location)
    100 0 300 arc                                   % define arc path
    gsave                                           % save the context
       .5 setgray fill                              % fill path with gray
    grestore                                        % restore context
    stroke                                          % draw the arc
} def

/makewin { %  - => - (builds a test window)
    /win framebuffer /new DefaultWindow send def    % cliche
    {
       /PaintClient { 100 100 draw_arc } def        % invoke draw_arc
       /FrameLabel ( Workspace ) def                % label the window
    } win send

    /reshapefromuser win send                       % size it
    /map win send                                   % make it visible
} def

makewin                                             % run the program
```

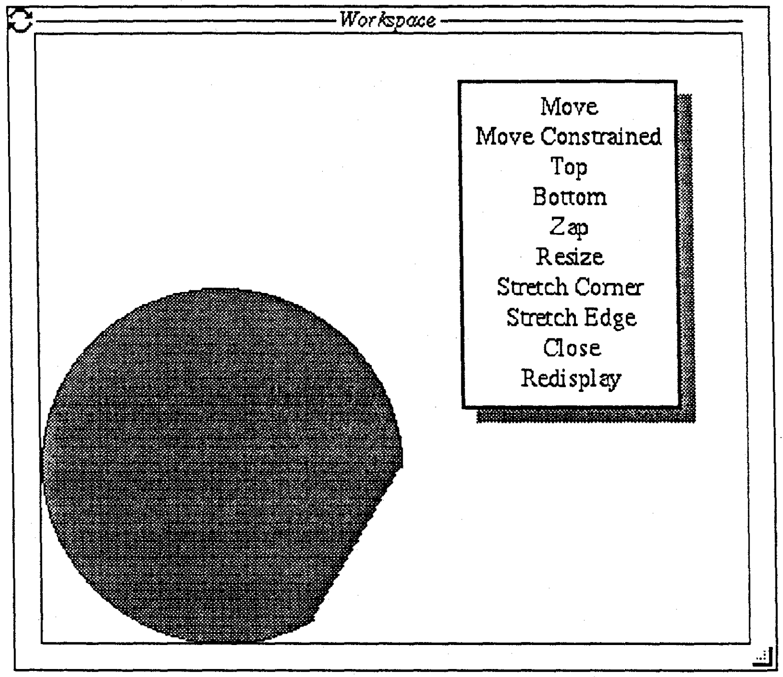
This change was an easy one. At the end of this chapter are a pair of examples of more complicated drawing procedures that have been integrated with this window.

Again, invoke `psh` to run the window program:

```
babylon% psh arc.ps
```

After you have sized the window, the following image should be drawn:

Figure 2-6 *drawing an arc in a window*



Note that this window has a menu. When you click the right mouse button a menu appears that you did not explicitly create. Because the client canvas didn't express interest in the menu button, the event generated by the mouse click was passed through to the frame canvas beneath the client canvas[13].

2.6. Establishing A Coordinate Space

Coordinates specified in a NeWS program refer to locations within an ideal coordinate system that are independent of the coordinate system of whatever device displays the image.

Following the implementation specification of the POSTSCRIPT language, NeWS defines a default user space which programs may modify with transformation operators such as **translate, scale** and **rotate**[14].

Using clippath pathbbox

The *cliché*:

```
clippath pathbbox    % - => x y w h     % return dimensions
```

sets the current path to the one describing the current clipping path and then returns the bounding box of the current path. The coordinates it returns are the lower left x, lower left y, upper right x, and upper right y[15].

[13] Menus are discussed in more detail in Appendix B, *Tailoring an Application*. Events and interests are discussed in Chapter 3, *Input* of the *NeWS Manual*.

[14] This mapping from user space to device space is *always* applied before any drawing operation.

[15] See clippath and pathbbox in the *PostScript Language Reference Manual* for a more detailed

This *cliché* can be very useful in initially establishing a coordinate space It is frequently coupled with the translate and scale operators to set up a "worldspace" before drawing an image:

Figure 2-7 *setting up a worldspace*

```
clippath pathbbox           % x y w h
4 2 roll                    % w h x y
translate                   % w h
scale                       % -
```

The roll changes the order of the elements on the operand stack (as per the comments)[16]. translate before the scale because the scale will change the current transformation matrix and you must be sure of the location of the origin[17].

2.7. The Next Step

The next chapter takes the program `arc.ps` and alters it to make use of NeWS classes and includes the first steps to create the `go` application from this framework.

2.8. NeWS Operators, Methods and Keys

The following are all the NeWS features that have been introduced in this chapter:

/new

canvas **/new** window
Creates a new window. Sent to a window class, generally **DefaultWindow**, to create an instance of the class. The canvas is the parent canvas for the window and is generally **framebuffer**. After creating a window, a client may want to modify the window by changing its drawing routines, adding a client menu, changing its frame or icon label, etc. The client makes these modifications by changing instance variables in the new window (typically, by sending the window an executable array as a method).

/map

– **/map** –
Make the window/icon visible. Fork the window's event manager if that has not already been done. **/map** is initially called by the client, but is then handled by the window's user interface.

explanation of their behavior.

[16] See Appendix A, *Conventions* for notes on comment conventions in NeWS.

[17] See *Transformations* in *Section 4.4, Coordinate Systems* of the *PostScript Language Reference Manual* for a detailed explanation of the transformation operators.

/reshape

x y width height **/reshape** –

Note: This does *not* force the shape to be rectangular, just to fit *within* the bounding rectangle.

Reshape the window to have the given bounding box.

/reshapefromuser

– **/reshapefromuser** –

Reshape the window to have a new bounding box. The user is prompted for a bounding box, and the results are passed to **/reshape**. **/reshapefromuser** is initially called by the client, but is then handled by the window's user interface.

send

<optional args> method object **send** <optional results>

Establishes the object's context by putting it and its class hierarchy on the dictionary stack, executes the method, then restores the initial context. The method is typically the keyword of a method in the class of the object, but it can be an arbitrary procedure.

3

A First Use of Subclassing

3

A First Use of Subclassing

This chapter, develops a subclass of the class **DefaultWindow**. One of the great strengths of NeWS is in the ease with which objects can inherit behaviors from classes. The process of defining a mechanism where an object inherits these default behaviors is called *subclassing*.

Subclasses are very useful in providing newly-created windows with a specialized behavior[18]. With subclasses, we can alter or extend the class methods (which are inherited by every window of the subclass) to perform exactly as desired.

The first step is to rewrite `arc.ps` to create a window, an *instance*, of the class GoWindow[19]. The next step is to alter the drawing procedure again to create the first example of the `go` board.

3.1. Defining a Class

Here is a "pseudo-code" listing of a class definition:

Figure 3-1 *a pseudo-class*

```
/name OBJECT
   dictbegin
      instance variables
   dictend
   classbegin
      class variables
      class methods
   classend
def
```

Briefly, the variables in the class dictionary (those lines bounded by the **dictbegin** and **dictend** in `arc2.ps` below) are instance variables. When you alter one of these variables it will have a purely local effect. Only the behavior of the individual window will be altered.

[18] Classes and developing a subclass are thoroughly explained in the *NeWS Manual*, Chapter 6, *Classes*. A sample class is implemented in Chapter 7 of the *NeWS Technical Overview*.

[19] An *instance* refers to a specific member of the class. The window is an object created from the class definition.

Conversely, class methods and variables (found between the **classbegin** and **classend** statements) are inherited by each member of a class (or subclass) at the time of creation. Should you alter or extend these methods, this change will have an effect on all instances of the class created after the change was made.

Consider the use of **/PaintClient** and **/FrameLabel** in the following implementation of the sample window program, rewritten to use the GoWindow subclass:

Figure 3-2 *code/arc2.ps*

```
/draw_arc { % x y => - (draws an arc at the specifed location)
    100 0 300 arc                                   % locate the arc
    gsave                                           % save the context
      .5 setgray fill                               % fill path with gray
    grestore                                        % restore context
    stroke                                          % draw the arc
} def

/makewin { %  - => - (builds a test window)
    /GoWindow DefaultWindow                         % create a subclass
    dictbegin                                       % begin local dict
      /FrameLabel ( Workspace ) def
    dictend                                         % end local dict
    classbegin                                      % begin class defs
      /PaintClient { 100 100 draw_arc } def         % draw gray arc
      /PaintIcon {                                  % drawing icon image
            .5 fillcanvas                           % fill with gray
            0 strokecanvas                          % stroke the perimeter
      } def
     classend def                                   % end class definition

     /win framebuffer /new GoWindow send def        % cliche
     /reshapefromuser win send                      % resize window
     /map win send                                  % make it visible
} def

makewin                                             % run the program
```

Note that while **/PaintClient** and **/FrameLabel** have been used as local (instance) variables they are now separated. **/PaintClient** is generalized across the entire subclass because it provides us with common behavior (that is the same for all go boards). **/FrameLabel**, however, remains an instance variable because the different instances of the subclass may well bear different names. In the event that more than one window (of the subclass GoWindow) is created, using /**PaintClient** as a window method saves resources (memory). This is at the cost of generalizing its behavior across all members of the class.

Generally, you should create a window as a subclass rather than altering a method with the **super send** pair (another subclassing technique)[20]. There are

[20] The use of the **super send** pair is discussed in detail in Chapter 6 of the *NeWS Manual, Classes.* We use it in Chapter 6 and Appendix B of this manual.

slight speed advantages to subclassing and a significant savings in memory use. You will notice that another method, **/PaintIcon** has been added to the code listing. It is very easy to paint an icon. In this case:

Figure 3-3 *painting an icon*

```
/PaintIcon {                    % draw icon image
        .5 fillcanvas           % fill with gray
        0 strokecanvas          % stroke perimeter
} def
```

All this code does is fill the current canvas (the icon) with a shade of gray (.5 **fillcanvas**) and then stroke the perimeter of the canvas with a zero-width line (1 pixel). Like **/PaintClient**, **/PaintIcon** can draw much more complex images.

3.2. A First Go Board

Now, replace the /draw_arc routine with the /draw_board routine. This new routine (with the further addition of code to describe the path) will draw a 19x19 go board in the working window.

The following code (**/ClientPath**) sets the coordinate space up for the client canvas and thus for the /draw_board routine. It alters the scale of the client canvas to a 19x19 coordinate system. /ClientPath also sets up the current path to be the shape of the canvas.

Figure 3-4 *the* **/ClientPath** *and* **/IconPath** *methods*

BOARD_SIZE is a constant (defined below) with a value of 19.

```
/ClientPath { % x y w h => - (define a client canvas path)
        4 2 roll translate
        BOARD_SIZE div exch BOARD_SIZE div exch scale
        .5 .5 translate
        -.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
} def
/IconPath {ClientPath} def          % define an icon canvas path
```

The **/ClientPath** method is called by the **/reshapefromuser** method when you change the window's shape. **/reshapefromuser** provides the bounding box coordinates that **/ClientPath** receives on the operand stack. The same is true of the **/IconPath**[21].

This routine is a typical approach to the issue of translating and scaling a new canvas.

The **/ClientPath** method rolls the operand stack so that the x and y coordinates are on the top and then does a translate. The translate pops these coordinates off the stack, leaving just w and h. This translate fixes the client canvas origin to be x, y (relative to its parent). BOARD_SIZE is placed on the stack and h is divided by it, leaving the quotient on the stack. This value is exch'ed with the w and the same operation is performed on the w. The end result is that this scale operation yields a 19x19 coordinate space. This means that if you address a

[21] Because a **/reshape** operation is done when the icon is displayed.

location between 0 and 19 that that location will fall within the boundaries of the client canvas.

The sequence:

```
.5 .5 translate
-.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
```

moves the origin to .5 .5 of the client canvas' 19x19 coordinate space. This leaves the new origin slightly offset from the lower left corner. The **/rectpath** method adds a rectangle to the current path using the coordinates x, y, width and height. So the statement:

```
-.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
```

defines the path to be the entire client canvas.

Thus, when the /draw_board routine draws from 0,0 to 0,18 it leaves a .5 unit space around all four edges (because 0,0 is now .5 units inset from the bounds of the client path). This little trick simply isolates the board within the window with a margin.

The /draw_board routine itself is much simpler to understand:

Figure 3-5 *code/sample4.ps*

```
/BOARD_MAX   18  def                              % BOARD_SIZE - 1
/BOARD_SIZE  19  def                              % number of lines drawn

/board_color  .9 .69 .28 rgbcolor def             % wood
/line_color   0 0 0 rgbcolor def                  % black

/draw_board { % - => - (draw the playing surface)
   board_color setcolor                           % set the board color
   clippath fill                                  % fill the window w/wood
   line_color setcolor                            % set the line color
   0 1 BOARD_MAX {                                % draw the lines
      dup 0 moveto 0 BOARD_MAX rlineto
      0 exch moveto BOARD_MAX 0 rlineto
   } for
   stroke                                         % stroke the path
} def
```

The clippath operation clips the current path to the current clipping path. Since it hasn't drawn anything yet, that makes the current path the interior area of the window — the entire client canvas. The fill operation fills the current path with the specified color or shade of gray (in this case wood)[22].

[22] On a monochrome system the board appears as white.

3.3. The Sample Window Program Plus

/draw_board

All the routines composed together give us a complete sample window program which draw a go board.

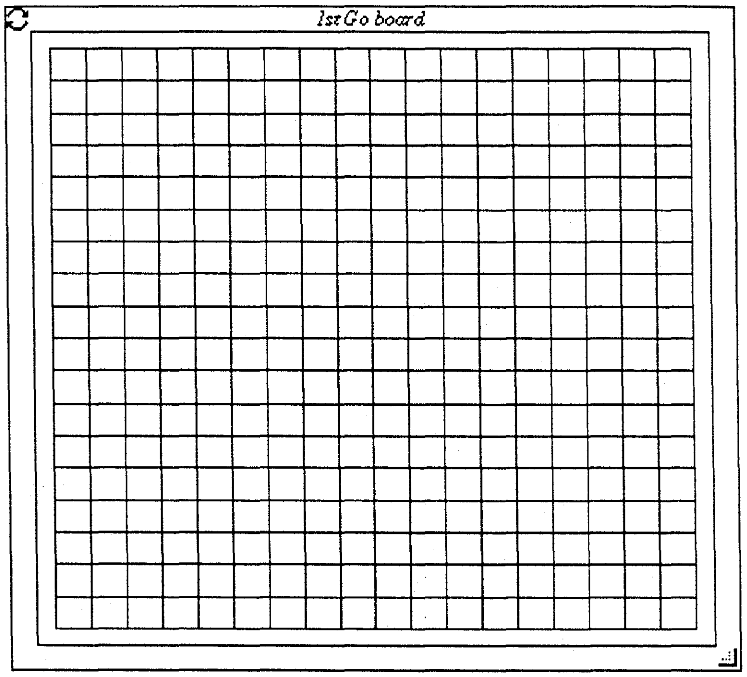
Figure 3-6 *code/go_window.ps*

Create an instance of the class GoWindow.

```
% constant definitions
/BOARD_SIZE 19 def
/BOARD_MAX 18 def

/board_color  .9 .69 .28 rgbcolor def           % wood
/line_color   0 0 0 rgbcolor def                % black

/draw_board { % - => - (draw the playing surface)
   board_color setcolor                         % playing surface color
   clippath fill                                % fill the client canvas
   line_color setcolor                          % set line color
   0 1 BOARD_MAX {                              % draw board (0,0->0,18)
      dup 0 moveto 0 BOARD_MAX rlineto
      0 exch moveto BOARD_MAX 0 rlineto
   } for
   stroke                                       % stroke the path
} def

/makewin { %  - => - (builds a test window)
    /GoWindow DefaultWindow                     % create a subclass
    dictbegin                                   % begin instance defs
       /FrameLabel ( 1st Go board ) def         % label the window
    dictend
    classbegin                                  % begin method defs
       /PaintClient { draw_board } def
       /PaintIcon { draw_board } def
       /ClientPath { % x y w h => -
          4 2 roll translate
          BOARD_SIZE div exch BOARD_SIZE div exch scale
          .5 .5 translate                       % move origin
          -.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
       } def
       /IconPath {ClientPath} def               % define icon image
    classend def

    /win framebuffer /new GoWindow send def     % cliche
    /reshapefromuser win send                   % resize window
    /map win send                               % draw it
} def

makewin                                         % run the program
```

running this program with psh creates the following window:

Figure 3-7 *the sample window*



3.4. Painting an Icon

The following example program demonstrates how easy it is to paint an icon:

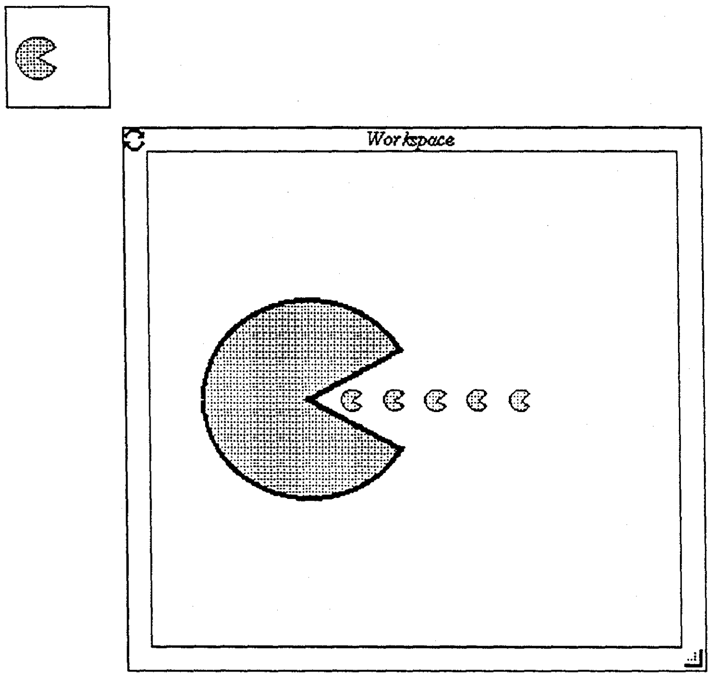
Figure 3-8 *code/muncher.ps*

```
/setup {                                   % set up world
   clippath pathbbox                       % obtain dimensions
   scale                                   % scale to fit
   pop pop                                 % -

   .3 .5 translate                         % move origin
   0 0 moveto
   0.01 setlinewidth                       % set stroke width
} def

/muncher {                                 % draw muncher
   newpath
   0 0 .2 30 330 arc
   0 0 lineto
   closepath
   gsave                                   % save graphics context
       .8 setgray
       fill
   grestore                                % restore graphics contex
   stroke                                  % stroke outline
} def
```

```
/makewin { % - => - (builds a test window)
  /GoWindow DefaultWindow                      % create a subclass
  dictbegin                                    % begin instance dict
    /FrameLabel ( Workspace ) def
  dictend                                      % end instance dict
  classbegin
    /PaintClient {                             % draw client canvas
      setup                                    % set up world
      muncher                                  % draw muncher
      .1 .1 scale
      1 1 5 {                                  % draw baby munchers
        .8 0 translate
        0 0 moveto
        muncher
      } for
    } def
    /PaintIcon {                               % draw the icon
      erasepage
      setup
      muncher
      currentcanvas setcanvas                  % set a path
      0 strokecanvas                           % stroke border
    } def
  classend def

  /win framebuffer /new GoWindow send def      % cliche
  /reshapefromuser win send                    % resize window
  /map win send                                % make it visible
} def

makewin                                        % run the program
```

The following image shows both the iconic form of the window and the image drawn in the window:

Figure 3-9 *muncher*



3.5. The Next Step

We expect that you have gained a familiarity with the NeWS environment in reading the companion volumes: the *NeWS Manual* and the *NeWS Technical Overview*.

CAUTION **If you have not done so, please read the chapter on classes in the** *NeWS Manual* **and the section on lightweight processes in the** *NeWS Technical Overview* **before reading further. It is essential that you have a good grasp of the NeWS process model and class hierarchy.**

The next chapter will expand on the work you've already done, writing a simple program which will communicate between the client and server sides.

The client side will contain code for calculations and the server side the user interface.

3.6. NeWS Operators, Methods and Keys

The following are all the NeWS features that we have introduced in this chapter:

classbegin

classname superclass instancevariables **classbegin** –
Creates an empty class dictionary that is a subclass of *superclass*, and has *instancevariables* associated with each instance of this class. The dictionary is put on the dictionary stack. *Instancevariables* may be either an array of keywords, in which case they are initialized to null, or a dictionary, in which case they are initialized to the values in the dictionary.

classend

– **classend** classname dict
Pops the current dictionary off the dictionary stack (put on by **classbegin** and presumably filled in by subsequent **defs**), and turns it into a true class dictionary. This involves compiling the methods and building various data structures common to all classes.

currentcanvas

– **currentcanvas** canvas
Returns the current value of the canvas parameter in the graphics state.

dictbegin

– **dictbegin** –
Combined with **dictend**, creates a dictionary large enough for subsequent **defs** and puts it on the dictionary stack. Avoids guessing what size dictionary to create.

dictend

– **dictend** dict
Returns the dictionary created by a previous **dictbegin**.

rectpath

x y width height **rectpath** –
Adds a rectangle to the current path with *x*, *y* as the origin.

setcanvas

canvas **setcanvas** –
Sets the current canvas to be *canvas*. Implicitly executes **newpath initmatrix**.

strokecanvas

int/color **strokecanvas** –
Strokes the border of the canvas with a one-point edge with the gray value or color. Currently only works for rectangular canvases.

4

Connecting to the Client Side

4

Connecting to the Client Side

In this chapter, we will discuss how to develop an application that will work across the gap between NeWS server and client program. Then you will apply this knowledge to the window program developed in Chapter 3, creating an application that splits its functionality between a C client and the NeWS server. Keep in mind that this application will allow interaction between the NeWS server and the user. For the remainder of this manual we will enhance this application (which draws a `go` board) adding more and more sophisticated features.

4.1. Principles of Application Development

As discussed in the *Introduction* to this manual, a major objective in splitting an application between server and client sides is to obtain optimal performance. So, in approaching the question of just what to put into the client side program and what to put into the server side, you should use as a measure the degree of complexity of the task. The NeWS server is best suited to dealing with simple mathematical operations and the rendering of shapes. Complex mathematical operations, or operations requiring a large number of iterations should be folded into the client side.

For instance, say your application is one which reads data from a file, builds matrices with this data (after performing mathematical operations on it), and then uses these matrices to produce a number of two-dimensional mappings displayed in a window. While all these operations *could* be done in a POSTSCRIPT language engine (and thus, in the NeWS server) you would probably find that such an application wasn't particularly speedy.

There are a couple of ways you could deal with the issue of splitting this application's functionality. At a first approximation, you could simply isolate the file manipulation and mathematical functions in a C client and pass the graph-rendering information to the NeWS server. You would be using the server as you would use a POSTSCRIPT language engine, off-loading (into client side code) those tasks that would impede its performance. This is a perfectly reasonable approach.

However, you would not being taking advantage of the real strength of NeWS: *it is interactive and programmable*. Let us say that you wish to be able to dynamically alter the coordinate system of the images displayed (perhaps to demonstrate a particular relationship). In the above model, you would recalculate the matrices on the client side and re-display the information.

With NeWS you have an alternative. A menu in the window would provide a good mechanism for letting the user change scales (or axis, or rotation in a plane).

Instead of directing your request back to the client side, you decide that it will be directed to a lightweight process within the NeWS server. This process performs a simple matrix multiplication operation and redisplays your graph (using cached data) with the new coordinate system (or orientation). Never having had to cross the client-server bridge, you may have gained a measurable benefit in real-time performance.

4.2. Other Considerations

There are a number of other considerations to be reviewed during the development of a client-server application:

- If you plan on running the application in a very low speed communication environment then you will want to reduce network bandwith,
- You may want to use a client-side user interface toolkit which hides the server side.
- You may be more comfortable writing in a language of your choice (other than the POSTSCRIPT language). Splitting the application across client and server allows you to use the language of your choice on the client side.

4.3. A New Set of Tools

This is your introduction to a new set of tools. The CPS interface consists of the `cps` program, which will convert a POSTSCRIPT program of the correct format to a form that a C program can use[23]. This program provides the material for the bridge that will span the gap between client and server halves of an application. CPS provides provides a number of C functions in the library `libcps.a`. They perform the following functions:

- open and close communication with the NeWS server,
- transmit POSTSCRIPT language code from the C client side to the NeWS server,
- receive data sent from the server side to the client side[24].

4.4. Conversion to CPS Format

Converting a POSTSCRIPT program for use with CPS is relatively simple. In general the conversion of a POSTSCRIPT language application to the client-server model is relatively easy. The functions isolated on the client side will need to be re-written in a client-side language[25]. Having separated your application into component halves, you will have to add the necessary utilities to allow the two halves to interact.

NeWS provides you with `cps` program that will do the low-level work of converting a POSTSCRIPT program (with a few minor changes) to a form that a C

[23] This converted form is not strictly the POSTSCRIPT language.

[24] For more detailed information on the this interface, read Chapter 9 of the *NeWS Manual, C Client Interface*. See the manual page on `cps(1)`.

[25] Currently the C language. See Chapter 15 of the *NeWS Manual, Supporting NeWS From Other Languages* for more information.

program can use.

We will now take the window program introduced in Chapter 3 and change it to be in CPS format. Initially, these additions will be few in number.

Take another look at the window program of Chapter 3:

Figure 4-1 *code/go_window.ps*

```
% constant definitions
/BOARD_SIZE 19 def
/BOARD_MAX 18 def

/board_color  .9 .69 .28 rgbcolor def              % wood
/line_color   0 0 0 rgbcolor def                   % black

/draw_board { % - => - (draw the playing surface)
   board_color setcolor                            % playing surface color
   clippath fill                                   % fill the client canvas
   line_color setcolor                             % set line color
   0 1 BOARD_MAX {                                 % draw board (0,0->0,18)
      dup 0 moveto 0 BOARD_MAX rlineto
      0 exch moveto BOARD_MAX 0 rlineto
   } for
   stroke                                          % stroke the path
} def

/makewin { %  - => - (builds a test window)
     /GoWindow DefaultWindow                       % create a subclass
     dictbegin                                     % begin instance defs
        /FrameLabel ( 1st Go board ) def           % label the window
     dictend
     classbegin                                    % begin method defs
        /PaintClient { draw_board } def
        /PaintIcon { draw_board } def
        /ClientPath { % x y w h => -
           4 2 roll translate
           BOARD_SIZE div exch BOARD_SIZE div exch scale
           .5 .5 translate                         % move origin
           -.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
        } def
        /IconPath {ClientPath} def                 % define icon image
      classend def

      /win framebuffer /new GoWindow send def      % cliche
      /reshapefromuser win send                    % resize window
      /map win send                                % draw it
} def

makewin                                            % run the program
```

What do you need to do to it to convert it for use with CPS? Well, note first that there is no interaction with the user in this simple program beyond the initial sizing of the window. That doesn't change in this chapter. Thus, all that need be done is to modify `go_window.ps` to make it acceptable to CPS and to create a simple C client to invoke the routines defined in the modified `go_window.ps` (within the context of the NeWS server).

First, tackle the issue of making this program acceptable as `cps` input. The `cps` program expects that the file passed to it will have POSTSCRIPT language definitions and procedures encapsulated in **cdef** statements. A **cdef** statement is created by placing **cdef** on a newline, followed by some identifier in the form of a C function name[26]. A **cdef** statement is terminated either by the end of the file or another **cdef** statement. Thus, it is quite possible to take an entire POSTSCRIPT program and encapsulate it within one **cdef** statement. This statement then becomes a function, which may be called from the C client, as if it were any other function in a C library.

We need to do a little more with our window program. First, we will encapsulate the body of the program with the statement except for the concluding invocation of makewin:

```
cdef initialize()                           % begin program body
```

Then we will create a separate function `execute()` to explicitly call the routine makewin:

```
cdef execute()
        makewin                             % execute procedure
```

These additions alone (with a simple C program) would be sufficient to allow us to invoke the drawing of the `go` board from outside the NeWS server. Our C client need only invoke `initialize()` to download the body of the POSTSCRIPT program to the server, and then invoke `execute()`. However, we need to make two more additions to insure that we don't leave our C program hanging when we terminate the NeWS process within the server.

The pair:

Figure 4-2 *a first* **tag**

```
#define DONE_TAG 1              % define tag value
cdef done() => DONE_TAG()       % define return value
```

define a value to be returned to the client side from the server side[27]. The

[26] A complete explanation of syntax for both **cdefs** may be found in Chapter 9, *C Client Interface* of the *NeWS Manual.*

function `done()` will be called from the client side. When called it looks at the input file (`PostScriptInput`) to determine whether the DONE_TAG has been sent to the client[28].

The second addition is to alter the /DestroyClient method in the class GoWindow so that it returns the **tag** value (by means of a **tagprint** statement) to the client-side C program. The frame event manager (the lightweight process managing the window) calls this method in the event of a choice of 'Zap' from the window menu. The value (DONE_TAG) is returned from the server side when the process controlling the window is destroyed:

```
/DestroyClient {                          % override class method
    DONE_TAG tagprint                     % return 1 to client
}
```

The disappearance of the window processes on the server side severs the connection to the client side (the file `PostScriptInput` receives an `eof`). The receipt of the DONE_TAG value on the input stream insures that the client-side program terminates gracefully.

4.5. The Sample Window Program Converted

Here is the go program converted for use with CPS. It is now capable of simple communication between client and server.

Figure 4-3 *go/go3.cps*

```
% PostScript constants
#define DONE_TAG    1                       % see function call defs..

cdef initialize()

/BOARD_SIZE    19 def                       % number of lines drawn
/BOARD_MAX     18 def                       % BOARD_SIZE - 1

/black_color  0 0 0 rgbcolor def            % black
/white_color  1 1 1 rgbcolor def            % white
/board_color  .9 .69 .28 rgbcolor def       % wood
/line_color   black_color def               % line color

/draw_board { % - => - (draw the playing surface)
   board_color setcolor
   clippath fill
   line_color setcolor
   0 1 BOARD_MAX {                          % draw the lines
      dup 0 moveto 0 BOARD_MAX rlineto
      0 exch moveto BOARD_MAX 0 rlineto
   } for
```

27 The syntax of **tags** is complicated. See Chapter 9, *C Client Interface*, of the *NeWS Manual* for a complete explanation.

```
  stroke
} def

/makewin { %  - => - (builds a go window)
    /GoWindow DefaultWindow                    % create a subclass
    dictbegin
       /FrameLabel ( 1st Go board ) def
    dictend
    classbegin
       /PaintClient { draw_board } def
       /PaintIcon {                            % define icon image
            .5 fillcanvas
            0 strokecanvas
       } def
       /DestroyClient {                        % override method
            DONE_TAG tagprint
       }
       /ClientPath { % x y w h => - (define the client path)
            4 2 roll translate
            BOARD_SIZE div exch BOARD_SIZE div exch scale
            .5 .5 translate
            -.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
       } def
       /IconPath {ClientPath} def              % define the icon canvas path
     classend def

     /win framebuffer /new GoWindow send def   % cliche
     /reshapefromuser win send                 % resize window
     /map win send                             % draw it
} def                                          % end initialize()

% function call definitions
cdef done() => DONE_TAG()                      % notify client side
cdef execute()
      makewin                                  % execute program
```

This conversion took very little effort. Of course, the above program is simply one-half (the server side) of our equation. It defines the information to be passed to the client (and the actions to take place on the server side). To have any measure of sophisticated control over the process of execution you need to write a program on the client side.

4.6. Creation of the Client Side

In general, the client side receives information on user interaction from the NeWS server. A program on the client side can do many things with the information it receives. Commonly, it coordinates the execution of functions on the server side. The following C program is the complement to the CPS program we've just written. It both passes the routines to the server and waits for notification of completion before exiting.

[28] See Chapter 9 of the *NeWS Manual* for a complete explanation of the client-server interface.

The CPS interface provides several functions for the maintenance of connections with the NeWS server. These functions are `ps_open_PostScript()`, `ps_flush_PostScript()` and `ps_close_PostScript()`[29]. We use two of these functions to open and close a connection to the NeWS server (`ps_open_PostScript` and `ps_close_PostScript`). These functions are available through the CPS library, `libcps.a`. As explained in Section 4.7, *Using CPS*, these functions are defined, along with a number of others[30], in the header file created by `cps`.

Figure 4-4 *go/go3.c*

```
#include "go3.h"        /* go3.cps => go3.h */

main ()
{
    /* open connection to server */
    if (ps_open_PostScript() == 0) {
        fprintf (stderr,"Cannot connect to NeWS server\n");
        exit (1);
    }

    /* load routines and execute */
    initialize ();
    execute ();

    /* let us know when done */
    while (!psio_error(PostScriptInput)) {
        if (done() || (psio_eof(PostScriptInput)) ) {
            printf ("Task completed\n");
            break;
        }
    }

    /* close the connection to the server */
    ps_close_PostScript();
    exit(0);
}
```

The `while` loop executes until an error condition is noted on the input stream (which the server writes on) or a DONE_TAG is received. The first step within the loop is a call to the `done()` function, which checks to see whether the specified tag (in the `.cps` file) has been placed on the stream. If the connection with the server has been broken, an `eof` is placed on the input stream. The call to the `psio_eof()` function checks for this eventuality. If either of these conditions are true the client side C program terminates, closing the connection to the NeWS server (if it is not closed already).

[29] For a complete explanation of these functions, see Chapter 9 of the *NeWS Manual.*

[30] These functions (provided by `cps`) are detailed in Chapter 9 of the *NeWS Manual.*

In effect, all this simple C client does is to open a connection to the NeWS server, send over the body of POSTSCRIPT language code, execute it within the context of the server, and await an indication that execution has completed.

4.7. Using CPS

Building a version of go from the sample programs presented is a relatively easy task. First, you need to run cps against the target .cps file:

```
babylon% cps test.cps
```

This creates a .h file (called test.h) which should then be included by the C program at compile time (using a #include statement). You will need to add the CPS library, libcps.a, to the list of libraries searched by the linker. This may be done at compile time with the following command line form:

```
babylon% cc -I$NEWSHOME/include g3.c $NEWSHOME/lib/libcps.a
```

So Far

What have you accomplished so far?

- you have a window within which to work,
- you have a method for drawing in the window and have drawn the go board pattern and,
- you have a metric for deciding how to divide functionality within a CPS-based application,
- you have a mechanism which defines the interaction between server and client halves.

4.8. The Next Step

In this chapter we communicated with the NeWS server: downloading a POSTSCRIPT program and executing it. We have not, as yet, given any consideration to communicating with a specific process within the server (there can be very many). In the next chapter we will address the problems that arise when there is a need to communicate with a specific process within the NeWS server.

4.9. NeWS Operators, Methods and Keys

The following CPS directives were introduced in this chapter:

cdef

cdef label <POSTSCRIPT language code>
Defines a body of code on the NeWS server side which may be called from the client side with *label()* (as if it were a C function). Definition is terminated either by an eof or by another **cdef**.

#define

#define name constant
Defines *name* as *constant* in a cps file. Commonly used for defining tags.

5

Communication With a NeWS Process

5

Communication With a NeWS Process

While communicating with the NeWS server from a client-side application opens up many new possibilities, we still have not completely explored the capabilities of the server. In the last chapter, we passed NeWS procedures to the server and requested their execution on the server. However, one very real distinction between the NeWS server and a simple POSTSCRIPT language engine is that the server is capable of maintaining a set of concurrently executing *lightweight processes* (*lwp*).

These *lwp* are at the core of the design of the NeWS server. Each process executes independently until it **pauses** and only one process is active at a time. Each process carries with it its own graphics context, dictionary stack, execution stack, and operand stack.

In order to take full advantage of the server's capabilities you must be able to communicate with a specific process within the server. Communicating with one process limits the client side program to very simple interactions with the server. More complex interactions are only possible when you gain the ability to communicate with a specific process within the server.

To this end, there are a number of utilities which should prove very useful. These are the Client IDentification (CID) utilities.

5.1. A Review of NeWS Input

The NeWS server distributes input events from a single queue, which is ordered on the events' timestamps. Events are inserted in this queue in response to physical actions (such as key presses); they are distributed as they come to the head of the queue.

The NeWS input system is described in full in Chapter 3, *Input* of the *NeWS Manual.*

Processes receive events in which they have expressed interest; they do so by providing a template event which the distributed events must match.

Processes may generate events procedurally, and insert them in the input queue. These events are then handled exactly like physical events: they may be distributed back to the process that sent them, or to any other, as long as the recipient has expressed a matching interest.

Events have various fields, including a **Name** and **Action** which describe the event, and the **TimeStamp** mentioned above, which identifies when it happened. There is also a field called **ClientData**, which is ignored by the input distribution mechanism. It can filled with any information desired by the event's creator.

5.2. The CID Utilities

The CID package provides 3 procedures which are used by the sample program:

uniquecid

Returns a new value every time it is called. A match on this value can be used by client- and server-side code to recognize a synchronization rendezvous.

cidinterest

Takes a value such as the one returned by **uniquecid**, and constructs an interest identified by that value. Calling **forkeventmgr** with this interest as argument creates a process which will wait to receive an event identified by this value, and then execute a procedure included in that event.

sendcidevent

Takes a value such as the one returned by **uniquecid**, and a procedure to be executed. It constructs an event identified by that value, includes the procedure, and gives it to NeWS' input system to be distributed. If some process has invoked **cidinterest** on the same value, and passed the result to **forkeventmgr**, it will receive this event and execute the procedure.

5.3. The Event Manager

The NeWS process mentioned above is an event manager which waits dormant to receive an event from the input queue mechanism. An event manager is created by the following fragment of POSTSCRIPT language code:

Figure 5-1 *an event manager*

```
/repair { % - => - (repair the board)
   DAMAGE_TAG tagprint uniquecid dup typedprint
   [exch cidinterest] forkeventmgr
   waitprocess pop
} def
```

The statement:

```
DAMAGE_TAG tagprint
```

sends the tag DAMAGE_TAG to the client side C program. As you will see, upon the receipt of this tag the client side program will in return make a call to the `draw_board()` function, asking it to draw the go board in the window. The statement:

```
uniquecid dup typedprint
```

passes the value of uniquecid to the client side (using the **typedprint** statement). uniquecid provides a unique number with which to identify events. The dup simply insures that a copy of this unique number remains on the operand stack for use by the next statement. The statement:

```
[exch cidinterest] forkeventmgr
```

creates an interest identified with the same value passed to the client side and hands this interest to the forkeventmgr method.

Remember that '[' is the mark object in the POSTSCRIPT language.

The exch swaps the '[' and the uniquecid on the stack, so that uniquecid gets passed to **cidinterest** as an argument.

forkeventmgr then creates a lightweight NeWS process. This lwp is created with an interest in any events bearing this unique number as an identifier. The statement:

```
waitprocess pop
```

insures that the lwp pauses until the just-forked event manager process exits. Since **waitprocess** returns the value that was on the top of its stack when it exits, the pop simply serves to clean up the operand stack.

This procedure is called by the **/PaintClient** method (as our /draw_board used to be) when the window suffers damage:

```
/PaintClient { repair } def
```

This is a reasonably complex set of events to occur within such an abbreviated procedure. In summary, all that is really occurring is that when the /repair procedure is called, it notifies the client side that damage has occurred and creates a process which waits for an event directed at it.

5.4. Why Synchronize?

Why does the event manager process wait? The answer to this is the key to the use of the CID utilities. The process pauses in its execution until it receives an event. When it receives that event (that matches the interest it has specified) it executes the code that it finds in the **/ClientData** field of that event.

So, upon the receipt of the DAMAGE_TAG from the /repair procedure the client side program invokes `draw_board()`. The invocation of `draw_board()` from the client side inserts an event into the distribution queue. The definition of this call gives us a clue to understanding this:

```
cdef draw_board(int id)
   id {draw_board} sendcidevent                    % launch event with id
```

If you look up **sendcidevent** in the Chapter 4 of the *NeWS Manual* you will find that it takes two arguments, an *id* and a *proc*. The *id* becomes a label for the event which it inserts into the input distribution mechanism. The *proc* becomes the contents of the /ClientData field of the event, and is thus carried by it.

The portion of the client side code that deals with this looks like:

```
/* let us know when done */
while (!psio_error(PostScriptInput)) {
    if (get_damage(&id)) {
       draw_board(id);
       repaired(id);
    } ...
```

The `get_damage()` call simply checks the input stream to determine whether the DAMAGE_TAG has been received. If it has, it calls the `draw_board()` and `repaired()` functions in succession.

WARNING **It is important to note that all functions which look for an argument associated with a tag value need to pass that value by address. You will note, in the above example, that** `get_damage()` **passes back the id value as an address:** `&id`.

What we have created may be likened to a communication channel between the client and server sides. We now have a dedicated lwp running in the context of the NeWS server, waiting to be sent code to execute. We can direct code at this particular process because it has expressed interest in events with an id that the client-side program knows. This process will continue to run within the server until we send it an event which causes it to exit.

Looking at the function `repaired()` we have just such a mechanism:

```
cdef repaired(int id)
    id {exit} sendcidevent
```

The waiting process receives an event with the correct id and executes the exit statement, terminating itself.

5.5. Adding CID To the Server Side

Now that you have some understanding of both how the CID utilities work and what you want them to do let's add them to our POSTSCRIPT program. We have discussed almost all the additions that need to be made. The only additional changes will be to change the **/FrameLabel** variable and the **/PaintClient** class variable. **/PaintClient** now invokes the /repair procedure.

With the addition of the synchronization procedures the complete server side code now looks like this:

Figure 5-2 *go/go4.cps*

```
% PostScript constants (see function call defs..)
#define DONE_TAG     1
#define DAMAGE_TAG   2
```

```
cdef initialize()

/BOARD_SIZE    19 def                                % number of lines drawn
/BOARD_MAX      18 def                               % BOARD_SIZE - 1

/black_color   0 0 0 rgbcolor def                    % black
/white_color   1 1 1 rgbcolor def                    % white
/board_color   .9 .69 .28 rgbcolor def               % wood
/line_color    black_color def                       % line color

/repair { % - => - (repair the board)
   DAMAGE_TAG tagprint uniquecid dup typedprint
   [exch cidinterest] forkeventmgr
   waitprocess pop
} def

/draw_board { % - => - (draw the playing surface)
   board_color setcolor
   clippath fill
   line_color setcolor
   0 1 BOARD_MAX {                                   % draw the lines
      dup 0 moveto 0 BOARD_MAX rlineto
      0 exch moveto BOARD_MAX 0 rlineto
   } for
   stroke
} def

/makewin { %  - => - (builds a go window)
     /GoWindow DefaultWindow                         % create a subclass
     dictbegin
        /FrameLabel ( 2nd Go board ) def
     dictend
     classbegin
        /PaintClient { repair } def
        /PaintIcon {                                 % define icon image
            .5 fillcanvas
            0 strokecanvas
        } def
        /DestroyClient {                             % override method
            DONE_TAG tagprint
        }
        /ClientPath { % x y w h => - (define the client path)
            4 2 roll translate
            BOARD_SIZE div exch BOARD_SIZE div exch scale
            .5 .5 translate
            -.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
        } def
        /IconPath {ClientPath} def                   % define the icon canvas path
      classend def

      /win framebuffer /new GoWindow send def        % cliche
      /reshapefromuser win send                      % resize window
      /map win send                                  % draw it
} def                                        % end initialize()

% function call definitions
cdef done() => DONE_TAG()                    % notify client side
cdef draw_board(int id)
    id {draw_board} sendcidevent             % draw the go board
cdef get_damage(int id) => DAMAGE_TAG(id)
cdef repaired(int id)
    id {exit} sendcidevent                   % close channel
cdef execute()
        makewin                              % execute program
```

5.6. Adapting the Client Side

The changes necessary on the client side are much less extensive. Since you can't yet place stones on the board, damage repair will consist of simply redrawing the board upon the receipt of a specific tag value (in the event of a board re-scaling). All that you have effectively added to the client side are the calls to `get_damage()`, `draw_board()`, and `repaired()`. The client side C program now looks like:

Figure 5-3 *go/go4.c*

```
#include "go4.h"

main ()
{
    int id;

    /* open connection to server */
    if (ps_open_PostScript() == 0) {
        fprintf (stderr,"Cannot connect to NeWS server\n");
        exit (1);
    }

    /* load routines into buffer */
    initialize ();

    /* execute them */
    execute ();

    /* let us know when done */
    while (!psio_error(PostScriptInput)) {
        if (get_damage(&id)) {
           draw_board(id);
           repaired(id);
        } else if (done() || (psio_eof(PostScriptInput))){
           printf ("Task completed\n");
           break;
        }
    }

    /* close the connection to the server */
    ps_close_PostScript();
    exit(0);
}
```

When the client-side C program receives the information that the board has been damaged the `draw_board()` function is called and then the `repaired()` function. Because of the work done on the server side of the application, these changes are all that is required.

5.7. A Cautionary Note

The CID mechanism outlined here is a clever mechanism for communications with an arbitrary NeWS lwp. However, you should be aware that there is some performance penalty for sending an event from the lwp that takes commands off the wire to another desired lwp. For example, using this mechanism in a situation where you wish to pump thousands of vectors to the server would result in an unacceptable slowdown.

One alternative would be to create a POSTSCRIPT language routine that renders a single graphic, download the routine once, and send the command to invoke the routine repeatedly.

Another alternative is to make sure that the transformation, graphics context and current canvas are set up in the lwp receiving commands from the connection. Then send the vectors to the server. This is the most efficient way to send a large number of rendering commands (the vectors) to the server[31].

5.8. So Far

So far in this chapter we have explained:

- a new mechanism for maintaining multiple communication channels, and
- you have applied this knowledge to give the client side control over damage repair on the server side

5.9. The Next Step

In this chapter we learned how to use the CID utilities to communicate with a single process within the NeWS server. However, this interaction has been both simple and on a low-level – with little of the mechanism of choice (the input mechanism) exposed. In the next chapter we will expose more of the NeWS process model and provide you with a means to track mouse actions on the server side.

5.10. NeWS Operators, Methods and Keys

The following NeWS features were introduced in this chapter:

[31] We haven't described this approach but hope to do so in the future.

cidinterest

id **cidinterest** interest
Creates an interest appropriate for use with **forkeventmgr**. The callback procedure installed in this interest simply executes the code fragment stored in the event's **/ClientData** field.

forkeventmgr

interests **forkeventmgr** process
Forks a process that expresses interest in *interests*, which may be either an array or a dictionary whose values are interests. Each interest must contain, in its **/ClientData** field, a dictionary having an entry (/proc) which is executed by the event manager process. This procedure is called with the event on the stack.

NOTE *The event manager uses some entries of the operand stack; do not use* **clear** *to clean up the stack in your* 'proc' *procedure.*

sendcidevent

id proc **sendcidevent** –
Sends a code fragment *proc* packaged in an event with a **/Name** of *id* to a process created by the use of the **cidinterest** — **forkeventmgr** pair.

typedprint

object **typedprint** –
Returns the object in an encoded form on the current output stream. Encoding is discussed in detail in Chapter 14 of the *NeWS Manual*, *Byte Stream Format.*

uniquecid

– **uniquecid** integer
Generates a unique identifier (*integer*) for use with the rest of the package.

waitprocess

process **waitprocess** value
Waits until *process* completes, and returns the value that was on the top of its stack at the time that it exited.

6

Tracking Mouse Actions

6 Tracking Mouse Actions

This chapter develops a mechanism for handling user input. This may be as simple as clicking a mouse button to place a stone or as complex as cycling an item button[32]. Up to this point, the user has had little interaction with the application beyond that allowed by the LiteWindow mechanisms.

The additions presented in this chapter will allow the user to place either a black or white stone on the go board. This, while seeming quite simple, is actually made possible by a non-trivial interaction between server and client program.

While this application could be kept very simple by only making additions on the server side, we have chosen to do record-keeping on the client side. Doing so leaves us with the option of someday hooking up this program with a go playing algorithm written in C.

This record-keeping function is one likely to be used by many different C clients (albeit in different forms). While we don't perform any calculations on the client side data, this would be easily done.

6.1. Following Mouse Actions

In concluding this model of application development, our final objective is to have the ability to follow mouse actions as they occur on the server side. We would like to know when a button is depressed, which button it is, and where it was when the transition occurred.

The movement of the mouse generates events. As does the depression and release of its buttons. This application is limited to simply reacting to the clicks.

6.2. What Do We Need?

What procedures do we need to add to the server-side program? The procedures to be added may be roughly divided into two groups: those necessary to draw and remove the stones; and, those necessary for mouse interaction.

In the first group, we will need procedures to:

- draw a stone of an arbitrary color
- place the stone according to the mouse event's x and y coordinates
- erase the stone

[32] See Appendix B of the *NeWS Manual* for a number of excellent examples of button implementations.

In the second group, we will need a number of changes to support a single complex function. This function will both create a process to wait for **/DownTransitions** of the mouse buttons, and will notify the client-side program of where it placed (or removed) the stone.

6.3. Server Side Changes

First, we'll look at the procedures necessary to draw and remove stones. After that we'll undertake a detailed explanation of the new complex function, the /ButtonMgr. As we encounter the need for new **cdefs** we'll talk about them. We conclude with a discussion of the changes needed on the client side to track the placement of stones on the server side.

drawing a stone

This first procedure (/stone) draws a stone of specified color and then pauses to allow other NeWS processes to execute.

STONE_SIZE is defined to be .80.

Figure 6-1 *drawing a stone*

```
/stone { % outline_color stone_color x y => - (draw stone)
   STONE_SIZE 2 div 0 360 arc % set stones path
   gsave                      % save path
     setcolor fill            % fill with stone_color
   grestore                   % restore path
   setcolor stroke            % stroke path w/outline_color
   pause                      % allow others to execute
} def
```

The next pair of procedures are a logical couple, /checkloc being used only by /placestone. /placestone has a fairly complex task:

Figure 6-2 *placing the stone*

```
/checkloc { % float => int (convert location to legal board location)
   0 max BOARD_MAX min round
} def

/placestone { % event tag => - (place stone at event's x,y)
   ClientCanvas setcanvas                         % set current canvas
   tagprint uniquecid dup typedprint              % send tag & id to client
   exch                                           % uniquecid event
   begin                                          % begin local dictionary
     XLocation checkloc YLocation checkloc        % round location
   end                                            % end local dictionary
   typedprint typedprint                          % send x,y to client
   [exch cidinterest1only] forkeventmgr           % create eventmgr
   waitprocess pop                                % wait for events
} def
```

First, it sets the current canvas to be the **ClientCanvas**. This insures that when the event manager process is created by the call to **forkeventmgr** it will be attached to the correct canvas. The line:

```
tagprint uniquecid dup typedprint
```

informs the client that a stone has been placed (the **tagprint** uses the tag passed to the routine) and then passes the unique identifier that future calls to the server side should use (**uniquecid** dup **typedprint**). We want to use the identifier returned by **uniquecid** within this routine, so the dup insures that a copy of it remains on the stack.

System events are labeled with the cursor location when they are generated[33]. The begin ... end pair places the event (which had been on the operand stack) on the dictionary stack. Now /checkloc can access the fields of the event dictionary, **XLocation** and **YLocation**. Once these two values have been rounded they are passed to the client-side program with the **typedprint** statements. You have seen the statement:

```
[exch cidinterest1only] forkeventmgr
```

before. The last time we used it we left the channel open (using **cidinterest** instead of **cidinterest1only**). The exch statement exchanges the '[' and the value returned from **uniquecid** on the stack.

This time only a single event is to be returned so this process will exit upon executing the *proc* carried by that event.

/placestone provides the client with the location the stone is placed at, and awaits an event which will contain the instructions to draw it on the server side. Let's look at the **cdefs** that allow this routine to be called from the client side:

```
cdef black_stone(int id, int x, int y)
    id {outline_color black_color x y stone} sendcidevent
cdef white_stone(int id, int x, int y)
    id {outline_color white_color x y stone} sendcidevent
```

As you can see, the event returned to the waiting process contains an invocation of the /stone procedure. The waiting process invokes /stone after placing the outline color, the stone color, and the x, y coordinates of the stone on the stack. Now let's take a look at the code necessary to remove a stone after we've placed it.

[33] See Chapter 5 of the *NeWS Manual* for an explanation of the structure of events, especially those generated by the system.

drawing a cross

In effect, this routine draws a cross in the region where a stone is present. A small path is defined which encompasses the entire stone and that path is filled with the board color. That serves to erase the stone.

However, drawing a cross is a somewhat more difficult issue than drawing a stone. The lines with which crosses are drawn is the narrowest width possible (one pixel). The imaging model defined by the POSTSCRIPT language leaves the particular instance of drawing a minimum-width line in a precise area undefined[34]. It has been called a "fence-post" problem. On which side of an arbitrary line does one draw a one-pixel line? How does one predict what the choice made by the POSTSCRIPT language interpreter will be?

There are other solutions but this one is reasonably quick, if not pretty. It offers the additional benefit of being easy to understand (if you're practiced with POSTSCRIPT language syntax). It doesn't attempt to anticipate the choice the NeWS server will make, it merely verifys that the result is acceptable.

Figure 6-3 *drawing a cross*

```
/cross { % x y => - (draw cross)
10 dict begin                                   % define a local dict
   /y exch def                                  % save the y value
   /x exch def                                  % save the x value

   % clear the stone:
   x .5 sub y .5 sub 1 1 rectpath               % define a path
   board_color setcolor                         % set color
   fill                                         % erase stone

   % draw the two cross strokes, carfully adjusting for edge locations:
   x .5 sub 0 max y moveto x .5 add BOARD_MAX min y lineto  % hrz stroke
   x y .5 sub 0 max moveto x y .5 add BOARD_MAX min lineto  % vrt stroke
   line_color setcolor                          % set color
   stroke                                       % stroke cross
   pause                                        % allow other processes
                                                %  to execute
end                                             % end dict
} def
```

This routine isolates the values of x and y in a local dictionary. This allows us to use them several times in the routine, confident that as long as we don't def the statments their value will remain unchanged:

```
x .5 sub y .5 sub 1 1 rectpath
```

draws a bounding path around the stone of unit dimension 1 (remember our STONE_SIZE is .80). When we fill this path we effectively erase whatever was

[34] It would be antithetical to the philosophy of display independence.

there before. The key line (which draws the cross) is:

```
x .5 sub 0 max y moveto x .5 add BOARD_MAX min y lineto
```

perhaps it could be more clearly written:

```
x .5 sub 0 max y moveto                % set start location
x .5 add BOARD_MAX min y lineto        % set end location
```

In the first line, we subtract .5 from the value of x and then take the **max** of the two values (x-.5, 0). This insures that we don't go over the edge of the go board. Then we moveto that location.

The lineto creates a path from the location we moved to to another point .5 units on the other side of x, once again checking to see that we don't exceed the bounds of the board.

This is repeated with a path along the vertical axis, and the path is then stroked. Finally, the routine **pauses** to allow other NeWS routines to execute.

The **cdef** that allows us to invoke this routine is:

```
cdef cross(int id, int x, int y)
  id {x y cross} sendcidevent                % draw a cross
```

quite similar to the **cdefs** for placing the stones.

These routines, /stone, /checkloc, /placestone, /cross allow us to both draw and remove stones from the board. Before we progress to a discussion of the additions necessary to support tracking mouse actions, we should briefly consider the changes necessary to the client side (to this point).

client side tag functions

The client-side program watches the input stream for specific tag values[35]. The tag value is an integer defined as either WHITE_TAG or BLACK_TAG. The client-side program needs two functions (defined by **cdefs**) to determine which tag it has received:

```
cdef  get_black(int id, int x, int y) => BLACK_TAG(id, y, x)
cdef  get_white(int id, int x, int y) => WHITE_TAG(id, y, x)
```

These, in company with the already-defined black_stone and white_stone, allow the client to both test the input stream and initiate a response on the server side:

[35] The values received by /placestone depend on which mouse button is depressed.

The ellipses mark code which is used to record the position of the stones. This will be discussed in greater detail later in this chapter.

```
/* loop until error on input stream */
while (!psio_error(PostScriptInput)) {
    if (get_black(&id, &x, &y)) {
          ...
       black_stone(id,x,y);
    } else if (get_white(&id, &x, &y))
          ...
       white_stone(id,x,y);
}
```

6.4. Accepting Input

The following sections describe the manner in which the server-side NeWS code tracks mouse (button) actions.

tracking mouse actions

As explained in Chapter 3 of the *NeWS Manual*, manipulation of the mouse generates events with the names: **/MouseDragged, /LeftMouseButton, /MiddleMouseButton, /RightMouseButton**. Right now, we're only interested in the effect of depressing a mouse button. If a mouse button is depressed or released, the **Name** of the event identifies which button is affected and the **Action** is one of the keywords **/DownTransition** or **/UpTransition.**

tracking button transitions

Now that we have the necessary procedures to both draw a stone and inform the client side of its location, we need to create an event manager to watch for the /DownTransitions of the mouse buttons. These transitions mark where stones are to be placed or removed. The event labels will be:

Figure 6-4 *event labels*

```
#define BLACK_EVENT    /LeftMouseButton
#define WHITE_EVENT    /MiddleMouseButton
```

building a button manager

Our event manager process (the /ButtonMgr) is a very simple construction. When /startinput is called it creates a process which waits for an event with the **Name /LeftMouseButton** or **/MiddleMouseButton** and an **Action** of **/DownTransition.**

Upon receiving an event the /ButtonMgr calls the procedure /placestone which informs the client side as described above.

Figure 6-5 *a button manager*

Remember the lwp is waiting for an event generated within the ClientCanvas

```
/downeventinterest {/DownTransition ClientCanvas eventmgrinterest} def

/startinput { % - => - (Wait for input)
   /ButtonMgr [
      BLACK_EVENT {BLACK_TAG placestone} downeventinterest
      WHITE_EVENT {WHITE_TAG placestone} downeventinterest
   ] forkeventmgr store
} def
```

The two routines /downeventinterest and /startinput form another logical couple; /downeventinterest only being called from /startinput.

eventmgrinterest takes four arguments and creates an interest suitable for use by **forkeventmgr**. An interest is created with the label of either BLACK_EVENT or WHITE_EVENT. It has an **Action** of **/DownTransition** and is aimed at the ClientCanvas. When the /ButtonMgr process receives this event (which will be sent to it by a call from the client side program) it will draw the appropriate color of stone (using the invocation of /placestone).

altering the frame event manager

Our final task is to insure that the frame event manager automatically starts up the /startinput procedure when it in turn starts up. This way, whenever our window is displayed the /ButtonMgr will be looking for **/DownTransitions** of mouse buttons:

To do this, we take advantage of NeWS' class hierarchy[36].

Figure 6-6 *altering the frame event manager*

```
/ForkFrameEventMgr {              % alter class method
   /ForkFrameEventMgr super send
   startinput                     % invoke /ButtonMgr
} def
```

We use **super send** in this routine to alter the class method. The use is quite simple, the underlying context quite complex[37]. In brief, the instruction:

```
/ForkFrameEventMgr super send
```

executes the **/ForkFrameEventMgr** method from the superclass of the current class. The superclass is **DefaultWindow**. The current class, GoWindow,

[36] As we did in Chapter 3, when we redefined the **/PaintClient** method of the class.

[37] For a detailed explanation, see the explanation of classes and **super send** in Chapter 6 of the *NeWS Manual, Classes.*

redefines the **/ForkFrameEventMgr** to add **/startinput** to the method, insuring that it is executed whenever the **/ForkFrameEventMgr** method is called.

cleaning up

Where the change to **/ForkFrameEventMgr** adds to the method defined in class **DefaultWindow**, the following change defines a method specific to class GoWindow.

Every time a new go window is created a /ButtonMgr process is launched, tied to that window. We must insure that that process dies when the window dies. So, we add to the **/DestroyClient** method as follows:

```
/DestroyClient {                        % override method
    ButtonMgr killprocess               % kill ButtonMgr
    DONE_TAG tagprint                   % inform client side
} def
```

/DestroyClient is null-defined in class **DefaultWindow**. As such, using **super send** would not gain us anything.

6.5. The Complete Server Side Code

Here is the complete listing of the modified server side code.

Figure 6-7 *go/go5.cps*

```
% Constants needed in both C & PostScript:
C: #define BOARD_SIZE  19

% tag values (see function call defs...)
#define DONE_TAG          1
#define DAMAGE_TAG        2
#define BLACK_TAG         3
#define WHITE_TAG         4

cdef initialize()
/BOARD_SIZE  19 def                              % number of lines drawn
/BOARD_MAX   18 def                              % BOARD_SIZE - 1
/STONE_SIZE  .80 def                             % stone diameter
/BLACK_EVENT /LeftMouseButton  def               % place black stone
/WHITE_EVENT /MiddleMouseButton  def             % place white stone

% define colors
/black_color    0 0 0 rgbcolor def           % black
/white_color    1 1 1 rgbcolor def           % white
/board_color    .9 .69 .28 rgbcolor def      % Wood color
/line_color     black_color def              % line color
/outline_color  black_color def

/repair { % - => - (repair the board)
   DAMAGE_TAG tagprint                       % send tag to client
   uniquecid dup typedprint                  % send id to client
   [exch cidinterest] forkeventmgr           % launch waiting process
   waitprocess pop                           % clear stack
```

```
} def

/draw_board { % - => - (draw the playing surface)
   board_color setcolor clippath fill
   line_color setcolor
   0 1 BOARD_MAX {                              % draw the lines
      dup 0 moveto 0 BOARD_MAX rlineto
      0 exch moveto BOARD_MAX 0 rlineto
   } for
   stroke                                       % stroke board path
   pause
} def

/stone { % outline_color stone_color x y => - (draw stone)
   STONE_SIZE 2 div 0 360 arc                   % set stones path
   gsave                                        % save context
      setcolor fill                             % fill with stone_color
   grestore                                     % restore context
   setcolor stroke                              % stroke restored stone path
   pause                                        % allow other processes
                                                %  to execute
} def

/cross { % x y => - (draw cross)
10 dict begin                                   % begin local dictionary
   /y exch def                                  % save as local var
   /x exch def                                  % save as local var

   % clear the stone
   x .5 sub y .5 sub 1 1 rectpath
   board_color setcolor fill

   % draw the two cross strokes, carfully adjusting for edge locations:
   x .5 sub 0 max y moveto x .5 add BOARD_MAX min y lineto    % horiz stroke
   x y .5 sub 0 max moveto x y .5 add BOARD_MAX min lineto    % vert stroke
   line_color setcolor
   stroke                                       % stroke cross
   pause                                        % allow other processes
                                                %  to execute
end                                             % end local dictionary
} def

/checkloc { % float => int (convert location to legal board location)
   0 max BOARD_MAX min round
} def

/placestone { % event tag => - (place stone at event's x,y)
   ClientCanvas setcanvas                       % set current canvas
   tagprint uniquecid dup typedprint            % send tag & id to client
   exch                                         % uniquecid event
   begin                                        % begin local dictionary
      XLocation checkloc YLocation checkloc     % round location
   end                                          % end local dictionary
   typedprint typedprint                        % send x,y to client
   [exch cidinterest1only] forkeventmgr         % create eventmgr
   waitprocess pop                              % wait for events
} def

/downeventinterest {/DownTransition ClientCanvas eventmgrinterest} def

/startinput { % - => - (Wait for input)
   /ButtonMgr [
      BLACK_EVENT {BLACK_TAG placestone} downeventinterest
      WHITE_EVENT {WHITE_TAG placestone} downeventinterest
   ] forkeventmgr store
} def

/makewin { %  - => - (builds a go window)
    /GoWindow DefaultWindow                     % create a subclass
    dictbegin                                   % begin instance variables
       /FrameLabel ( 3rd Go board ) def         % label window
       /ButtonMgr null def                      % create instance
    dictend                                     % end instance variables
    classbegin                                  % begin class definitions
       /PaintClient { repair } def              % define client canvas image
       /PaintIcon { draw_board } def            % define icon image
       /DestroyClient {                         % override method
           ButtonMgr killprocess                % kill ButtonMgr
           DONE_TAG tagprint                    % inform client
       } def
       /ForkFrameEventMgr {                     % alter class method
           /ForkFrameEventMgr super send
           startinput                           % start ButtonMgr
       } def
       /ClientPath { % x y w h => - (define client path)
           4 2 roll translate
           BOARD_SIZE div exch BOARD_SIZE div exch scale
           .5 .5 translate
           -.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
       } def
       /IconPath {ClientPath} def
     classend def                               % end class definitions

     /win framebuffer /new GoWindow send def    % cliche
     /reshapefromuser win send                  % resize window
     /map win send                              % draw it
} def
% end initialize()

% function call definitions
cdef done() => DONE_TAG()                                  % inform client
cdef get_damage(int id) => DAMAGE_TAG(id)                  % inform client
cdef get_black(int id, int x, int y) => BLACK_TAG(id, y, x) % inform client
cdef get_white(int id, int x, int y) => WHITE_TAG(id, y, x) % inform client
cdef draw_board(int id)                                    % draw the go board
    id {draw_board} sendcidevent
```

```
cdef black_stone(int id, int x, int y)                          % draw  black stone
    id {outline_color black_color x y stone} sendcidevent
cdef white_stone(int id, int x, int y)                          % draw  white stone
    id {outline_color white_color x y stone} sendcidevent
cdef cross(int id, int x, int y)                                % draw  cross
    id {x y cross} sendcidevent
cdef repaired(int id)                                           % close channel
    id {exit} sendcidevent
cdef execute () makewin                                         % execute program
```

Note that we have defined BOARD_SIZE using the `cps` macro:

```
    C: #define BOARD_SIZE               19
```

This is a macro which allows the definition of a C constant to take place on the server side. We take advantage of this in the client side code below.

Further, note the addition of the necessary **#define** and def statements to support our enhanced use of tags and the expanded drawing routines.

6.6. Client Side Changes

The interaction between client side and server side will now be as follows:

- a mouse button is clicked in the `go` window (server side),
- a lwp is launched to listen for an event directed at it (server side),
- the client side is informed with a tag and any additional information it may need (server side),
- the client side, watching the input stream, decides upon an action based on the received tag value and information (client side),
- if a stone or cross is to be placed it updates the stone array and returns the requisite information to the server side with a function call (client side),
- otherwise, it simply makes one of a number of function calls to functions within the server (client side),
- the waiting lwp, upon receipt of an event generated by the client side call to a function, executes the *proc* carried by that event.

This is a reasonably lengthy interaction but also a well-defined one.

The client side code must undergo some fairly severe changes to keep up with the revisions that we have made to the server side. We must add a number of things:

- An array to keep track of stone color and location.
- Calls to the functions on the server side (to draw the stones).

Here is where we discuss the code marked by ellipses in Section 6.3.3, *client side tag functions*.

```
            } else {
                board[x][y] = WHITE;
                white_stone(id,x,y);
            };
        } else if (get_damage(&id)) {
            draw_board(id);
            for (x = 0; x < BOARD_SIZE; x++)
                for (y = 0; y < BOARD_SIZE; y++)
                    switch(board[x][y]) {
                    case BLACK:  black_stone(id,x,y);
                                 break;
                    case WHITE:  white_stone(id,x,y);
                                 break;
                    }
            repaired(id);
        } else if (done() || (psio_eof(PostScriptInput)) )
            break;
    }

    /* close the connection to the server */
    ps_close_PostScript();
    exit(0);
}
```

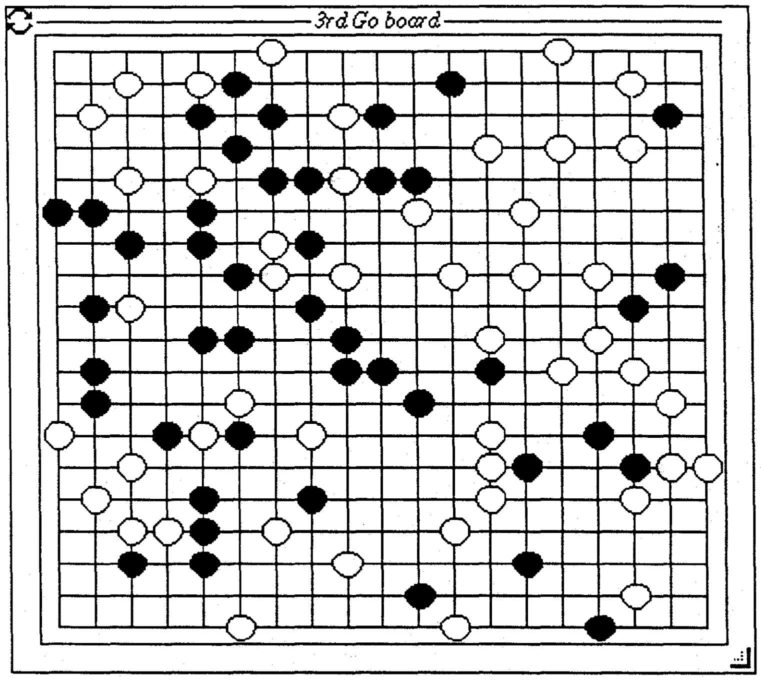
6.7. The Next Step

This chapter concludes this explanation of how to write a NeWS application. To this point, we've developed many of the skills necessary to write a useful application. Appendix B shows you how to add some features to this application which should prove very useful. It describes how to add a menu of your own design (using the **tag** feature) and how to alter the icon. Both these mechanisms will make use of NeWS classes in an effort to demonstrate their relative power and ease of use.

6.8. The Final Board

This program can serve as a playing aid to the game of go. Our final board looks as follows:

Figure 6-10 *the go program*



6.9. Conclusion

This demonstration program has been designed to illustrate *one* way to design the interaction between client and server. As such, it is not necessarily the most efficient, nor even the best way to do it.

The model of interaction presented by the design of the client and server sides should prove relatively easy to generalize to more complex applications.

Keep in mind a number of basic principles:

- Try to clearly define those tasks which are best done in the context of the server and in the context of the client,
- Use the class mechanism of NeWS to your advantage: inherit behavior, don't reinvent it,
- Design your CPS application to minimize round-trips between server and client.

Where do you go from here? We suggest that you try a number of programs written solely for use in the NeWS server. Learn how NeWS, and its mechanisms, work before you attempt to create a new application. Many of the programs used in the demo menu are good examples of programs written solely for use in NeWS. They represent a wide variety of interface design and many use the class mechanism extensively.

As you cut and pasted the examples in this manual, we urge you to cut examples out of these files for use in your own applications.

This manual has presented only the fundamentals of the mechanisms needed in developing a NeWS application split over the client-server bridge. There is much more to NeWS than we have been able to present here.

In the spirit of this approach, we urge you to continue to experiment with NeWS.

6.10. NeWS Operators, Methods and Keys

These are the NeWS operators and CPS directives that we have introduced in this chapter.

C: #define

C: #define constant value
Allows definition of a C *constant* with *value* from the `cps` file.

cidinterest1only

id **cidinterest1only** interest
A special form of **cidinterest** which processes only one code fragment. It automatically **exits** by itself, rather than requiring the client to send the exit.

eventmgrinterest

eventname eventproc action canvas **eventmgrinterest** interest
Makes an interest. Suitable for use by **forkeventmgr** or **expressinterest**.

expressinterest

event **expressinterest** –
Input events matching *event* will be queued for reception by **awaitevent**.

See Chapter 3, Input *of the* NeWS Manual *for more information on interest matching.*

killprocess

process **killprocess** –
Kills the specified NeWS *process*.

max

a b **max** c
Compares *a* and *b* and leaves the maximum of the two on the stack. Works on any data type for which **gt** is defined.

min

a b **min** c
Compares *a* and *b* and leaves the minimum of the two on the stack. Works on any data type for which **gt** is defined.

pause

– **pause** –
Suspends the current process until all other eligible processes have had a chance to execute.

setcanvas

canvas **setcanvas** –
Sets the current canvas to be *canvas*. Implicitly executes **newpath initmatrix**.

super

– **super** instance
Used as the target object with **send**, **super** refers to the method being overriden by the current method. Unlike **self**, **super** cannot be used outside the context of **send**.

XLocation

number **XLocation** number
System events are labeled with the cursor location at the time they are generated; this value is used to determine which canvas(es) the event can be distributed to. It is available to recipients and is transformed to the current canvas' coordinate system. This key accesses the X-coordinate of the location. It is ignored in interests.

YLocation

number **YLocation** number
This key accesses the Y-coordinate of the event location; see the explanation under **XLocation** above. It is ignored in interests.

A

Conventions

A

Conventions

This appendix lists the NeWS coding conventions which have come into common use. We encourage you to come back to this appendix as your understanding progresses[38].

A.1. Naming Conventions

When you write a new function, provide the reader with a list of what that function expects to find on the stack, what it leaves on the stack after completion, and what its purpose is. A typical function header would be of the form:

Figure A-1 *vanilla function header*

```
/vanilla { % var1 var2 array1 => value (returns a value from an array)
    ....
} def
```

A.2. Indentation

As with the C language, the POSTSCRIPT language can be made nearly unintelligible by neglecting to separate functions into logical subdivisions by indentation. Indentation has no functional effect on the interpretation of POSTSCRIPT but it is almost a requirement to understanding.

Indentation should follow the flow of control and execution. Consider this small function:

Figure A-2 *no indentation*

```
/uniquecid {currentcid 1 add /currentcid 1 index store} def
```

Unless you are quite experienced with POSTSCRIPT, this function is quite cryptic. Now look at the same code with indentation. Note how the form separates different steps in the execution. This form also makes the inclusion of comments much easier.

[38] Because of the flexibility of NeWS and the packages that we provide these conventions cannot be considered to be cast in stone. However, we do our best to adhere to them wherever possible and recommend that you do the same.

Figure A-3 *indentation*

```
/uniquecid {                    % create a unique id number
    currentcid 1 add            % increment /currentcid
    /currentcid 1 index         % /currentcid currentcid
    store                       % store value in /currentcid
} def
```

A.3. Stack Manipulation

Functions that depend on manipulation of the stack are efficient. However they can be very difficult to understand unless you comment them thoroughly. In-line comments about the contents of the stack are often useful. Here's an example:

Figure A-4 *stack-based manipulation*

```
/drift { % x y w h event => - (scale, translate, launch an event)
     5 1 roll                             % event x y w h
     4 2 roll                             % event w h x y
     translate                            % event w h
     scale                                % event
     dup begin                            % event event
        /Name   /Repaired def             % set /Name field
        /Canvas currentcanvas def         % set /Canvas field
     end                                  % event
     ...
} def
```

A.4. Cliches

Example programs in this document use of some of the mechanisms developed in the *Lite* packages provided with your NeWS release. The most common of these mechanisms we call *clichés*. They are very useful, performing commonly-needed operations such as creating a window, or scaling and translating it. These *clichés* are annotated both in the program comments and in the margins.

Understanding Cliches

These *clichés* are often representative of the sophistication of NeWS. You don't need to understand *clichés* before you use them, but it pays to have some grasp of the techniques they use. When you decide that you want a deeper understanding of NeWS you'll find numerous examples of the *clichés* in the *Lite* window packages. Classes and the inheritance of properties (key features in most uses of *clichés*) are discussed in some detail in Chapter 6 of the *NeWS Manual, Classes.*

A.5. A Cliche That Creates a Window

The following *cliché* serves to illustrate that the power of NeWS lies in its implementation of classes:

Figure A-5 *creating a window*

```
/win framebuffer /new DefaultWindow send def        % cliche
```

This *cliché* is used in Chapter 2 of this manual to create a sample working window. The order of interpretation is not immediately apparent. This *cliche* defines an instance of a window **/win** with the pair:

```
/win ... def
```

The **send** primitive (defined in Chapter 6 of the *NeWS Manual*, *Classes*) puts the class hierarchy of **DefaultWindow** on the dictionary stack, executes the **/new** method in that context (defined in Chapter 7 of the *NeWS Manual*, *Window and Menu Packages*):

```
framebuffer /new DefaultWindow send
```

/new takes a single argument, framebuffer: a canvas within which to create a new window. Further messages may be sent to this window with:

```
win send
```

Thus,

```
{/PaintClient {.5 setgray fill} def} win send
```

will alter the **/PaintClient** instance variable of the window (**/win**). It will now fill the window with a shade of gray when damage next occurs.

B

Tailoring An Application

B

Tailoring An Application

In this appendix you will learn how to add a menu of your own design to the go program. You will also learn how to adapt the icon to display a scaled-down image of the go board.

B.1. Server Side Changes

The changes required on the POSTSCRIPT language side (server) are few in number. You are already familiar with the use of tags and function definitions and so this will not be presented in great detail.

adding the menu

Adding a menu to the client canvas is a relatively simple task. The window package looks to see if the instance variable **/ClientMenu** is defined with a menu. If so, the window package will create an event manager for the menu button for the client canvas. The following code allows you to explicitly create a menu for the client canvas:

Figure B-1 *a new client menu*

```
/ClientMenu [              % alter class menu
   (Erase Board) {MENU_TAG tagprint ERASE_CMD typedprint}
   (Fill Board)     {MENU_TAG tagprint FILL_CMD typedprint}
] /new DefaultMenu send def
```

The **/new** method passed to **DefaultMenu** with the `send` expects an array of menu choices and associated actions[39]. This sequence:

```
[ (Erase Board)  {MENU_TAG tagprint ERASE_CMD typedprint}
  (Fill Board)       {MENU_TAG tagprint FILL_CMD typedprint}
] /new DefaultMenu send
```

puts the menu class object **DefaultMenu** on the stack and executes the method **/new** in the **DefaultMenu**'s class context. This creates an instance of the class **DefaultMenu** with the specified items.

[39] This is explained in detail in Chapter 7 of the *NeWS Manual, Window and Menu Packages.*

The pair:

```
/ClientMenu ... def
```

simply associates this menu with the instance variable **/ClientMenu**. When the **/ForkFrameEventMgr** method is called it looks to see whether the **/ClientMenu** variable is other than null-defined[2]. If a **/ClientMenu** exists the **/ForkFrameEventMgr** method creates an interest as specified by the menu. So, the frame event manager process will now have an interest in **/DownTransitions** of the right mouse button and will pop-up the client menu.

As you can see, we also need to add a tag value and a function that can be called from the client side. The tag value is:

```
#define MENU_TAG      5
```

The function call simply returns the command (cmd) that was invoked. It is as follows:

```
cdef get_menu(int cmd) => MENU_TAG(cmd)
```

The commands (erase and fill) are represented on both the C side and the POSTSCRIPT side:

```
% Constants needed in both C & PostScript:
C: #define ERASE_CMD    0
C: #define FILL_CMD     1
#define ERASE_CMD       0
#define FILL_CMD        1
```

adapting the icon

In **/flipiconic** we want the icon painted if it is going to iconic form and the client canvas is retained. This is because the window package doesn't call the **/PaintIcon** method to repaint the icon if it is already retained.

The only additions required to have the icon display the current condition of the go board are minor. However, they do require some explanation:

[2] **/ClientMenu** is one of a number of instance variables null-defined in class **LiteWindow**.

Figure B-2 *adapting the icon*

```
/PaintIcon { repair } def

/flipiconic { % - => - (Redraw current state for icon)
    /flipiconic super send
    Iconic? IconCanvas /Retained get and {/paint self send} if
} def
```

The **/PaintIcon** instance variable now calls the **/repair** procedure (the same as the **/PaintClient** procedure). Previously, it called the **/draw_board** routine, just drawing the empty go board. Now, it will draw the board with the stones placed just as they are in the full-sized window.

The sequence:

```
/flipiconic super send
```

executes the **/flipiconic** method from the superclass. The line:

```
Iconic? IconCanvas /Retained get and {/paint self send} if
```

is then added to the definition of the method in the current class.

Iconic? is an instance variable of class **LiteWindow**. It is a boolean which reflects the current condition of the window. It is true when the window is displayed in icon form (rather than the window itself).

```
IconCanvas /Retained get
```

The POSTSCRIPT language operator get returns a boolean value of the dictionary key **/Retained** for the canvas **IconCanvas**. If the canvas is retained, then the value returned is true[41]. The POSTSCRIPT operator and does a logical "and" between the two booleans. Thus, if either of them are false (the window is iconic or the icon canvas is not retained) a boolean with a value of false is put on the stack.

Finally, the if operator executes the procedure body

```
{/paint self send}
```

if the boolean left on the stack is true. The **/paint** method is sent to the current window, causing the icon to repaint with the current condition of the go board[42].

[41] See Chapter 11 of the *NeWS Manual, NeWS Type Extensions* for an explanation of canvas dictionary fields.

[42] The **/paint** method is described in Chapter 7 of the *NeWS Manual, Window and Menu Packages.*

In this fashion, the effect of damage can be simulated and the canvas redrawn with the current go board pattern. If the icon is unretained it will be redrawn every time another canvas uncovers it. This way it functions efficiently as a retained canvas but it can still be redrawn as the need arises.

B.2. Client Side Changes

The changes to the client side are somewhat more extensive than the changes to the server side (though not very complicated). We need to add a function to the client side to randomly generate a stone using the `rand()` library function. The function is as follows:

Figure B-3 *pick a random stone*

```
enum stone pickastone ()
{
    extern int rand();

    int i = rand() % 7;
    return (i>2 ? CROSS : (enum stone) i);
}
```

adding the menu

In addition, the following code fragment is added to support the menu calls. We add it to the already lengthy `if` statement:

Figure B-4 *responding to menu requests*

```
else if (get_menu(&cmd)) {
  for (x = 0; x < BOARD_SIZE; x++)
    for (y = 0; y < BOARD_SIZE; y++)
       board[x][y] = (cmd == FILL_CMD ? pickastone() : CROSS);
  repaint(id);
} ...
```

With the addition of the variable definition (as an integer) for `cmd` our client-side code is now complete.

B.3. A Complete Model

Following is a complete listing of both server and client sides of the go program:

Figure B-5 *go/go6.cps*

```
% Constants needed in both C & PostScript:
C: #define BOARD_SIZE      19
C: #define ERASE_CMD       0
C: #define FILL_CMD        1

% tag values (see function call defs...)
#define DONE_TAG          1
#define DAMAGE_TAG        2
#define BLACK_TAG         3
```

```
#define WHITE_TAG              4
#define MENU_TAG               5

cdef initialize()
/BOARD_SIZE  19 def                                   % number of lines drawn
/BOARD_MAX   18 def                                   % BOARD_SIZE - 1
/STONE_SIZE  .80 def                                  % stone diameter
/BLACK_EVENT /LeftMouseButton  def                    % place black stone
/WHITE_EVENT /MiddleMouseButton  def                  % place white stone
/ERASE_CMD   0 def                                 % erase board
/FILL_CMD    1 def                                 % fill with stones

% define colors
/black_color    0 0 0 rgbcolor def                 % black
/white_color    1 1 1 rgbcolor def                 % white
/board_color    .9 .69 .28 rgbcolor def            % Wood color
/line_color     black_color def                    % line color
/outline_color  black_color def

/repair { % - => - (repair the board)
   DAMAGE_TAG tagprint                             % send tag to client
   uniquecid dup typedprint                        % send id to client
   [exch cidinterest] forkeventmgr                 % launch waiting process
   waitprocess pop                                 % clear stack
} def

/draw_board { % - => - (draw the playing surface)
   board_color setcolor clippath fill
   line_color setcolor
   0 1 BOARD_MAX {                                 % draw the lines
      dup 0 moveto 0 BOARD_MAX rlineto
      0 exch moveto BOARD_MAX 0 rlineto
   } for
   stroke                                          % stroke board path
   pause
} def

/stone { % outline_color stone_color x y => - (draw stone)
   STONE_SIZE 2 div 0 360 arc                      % set stones path
   gsave                                           % save context
     setcolor fill                                 % fill with stone_color
   grestore                                        % restore context
   setcolor stroke                                 % stroke restored stone path
   pause                                           % allow other processes
                                                   %  to execute
} def

/cross { % x y => - (draw cross)
10 dict begin                                      % begin local dictionary
   /y exch def                                     % save as local var
   /x exch def                                     % save as local var

   % clear the stone
```

```
  x .5 sub y .5 sub 1 1 rectpath
  board_color setcolor fill

  % draw the two cross strokes, carfully adjusting for edge locations:
  x .5 sub 0 max y moveto x .5 add BOARD_MAX min y lineto    % horiz stroke
  x y .5 sub 0 max moveto x y .5 add BOARD_MAX min lineto    % vert stroke
  line_color setcolor
  stroke                                        % stroke cross
  pause                                         % allow other processes
                                                %  to execute
end                                             % end local dictionary
} def

/checkloc { % float => int (convert location to legal board location)
  0 max BOARD_MAX min round
} def

/placestone { % event tag => - (place stone at event's x,y)
  ClientCanvas setcanvas                        % set current canvas
  tagprint uniquecid dup typedprint             % send tag & id to client
  exch                                          % uniquecid event
  begin                                         % begin local dictionary
    XLocation checkloc YLocation checkloc       % round location
  end                                           % end local dictionary
  typedprint typedprint                         % send x,y to client
  [exch cidinterest1only] forkeventmgr          % create eventmgr
  waitprocess pop                               % wait for events
} def

/downeventinterest {/DownTransition ClientCanvas eventmgrinterest} def

/startinput { % - => - (Wait for input)
  /ButtonMgr [
    BLACK_EVENT {BLACK_TAG placestone} downeventinterest
    WHITE_EVENT {WHITE_TAG placestone} downeventinterest
  ] forkeventmgr store
} def

/makewin { % - => - (builds a go window)
    /GoWindow DefaultWindow                     % create a subclass
    dictbegin                                   % begin instance variables
      /FrameLabel ( 3rd Go board ) def          % label window
      /ButtonMgr null def                       % create instance
    dictend                                     % end instance variables
    classbegin                                  % begin class definitions
      /PaintClient { repair } def               % define client canvas image
      /PaintIcon { repair } def                 % define icon image
      /DestroyClient {                          % override method
          ButtonMgr killprocess                 % kill ButtonMgr
          DONE_TAG tagprint                     % inform client
      } def
      /flipiconic { % - => - (Redraw current state for icon)
          /flipiconic super send                % alter superclass
```

```
            Iconic? IconCanvas /Retained get and {/paint self send} if
        } def
        /ForkFrameEventMgr {                          % alter class method
            /ForkFrameEventMgr super send
            startinput                                % start ButtonMgr
        } def
        /ClientPath { % x y w h => - (define client path)
            4 2 roll translate
            BOARD_SIZE div exch BOARD_SIZE div exch scale
            .5 .5 translate
            -.5 -.5 BOARD_SIZE BOARD_SIZE rectpath
        } def
        /IconPath {ClientPath} def
        /ClientMenu  [                                % define dedicated menu
            (Erase Board)   {MENU_TAG tagprint ERASE_CMD typedprint}
            (Fill Board)    {MENU_TAG tagprint FILL_CMD typedprint}
        ] /new DefaultMenu send def                   % install menu
    classend def                                      % end class definitions

    /win framebuffer /new GoWindow send def           % cliche
    /reshapefromuser win send                         % resize window
    /map win send                                     % draw it
} def
% end initialize()

%  function call definitions
cdef done() => DONE_TAG()                                     % inform client
cdef get_damage(int id) => DAMAGE_TAG(id)                     % inform client
cdef get_black(int id, int x, int y) => BLACK_TAG(id, y, x)   % inform client
cdef get_white(int id, int x, int y) => WHITE_TAG(id, y, x)   % inform client
cdef get_menu(int cmd) => MENU_TAG(cmd)                       % inform client
cdef draw_board(int id)                                       % draw the go board
    id {draw_board} sendcidevent
cdef black_stone(int id, int x, int y)                        % draw  black stone
    id {outline_color black_color x y stone} sendcidevent
cdef white_stone(int id, int x, int y)                        % draw  white stone
    id {outline_color white_color x y stone} sendcidevent
cdef cross(int id, int x, int y)                              % draw  cross
    id {x y cross} sendcidevent
cdef repaired(int id)                                         % close channel
    id {exit} sendcidevent
cdef repaint()                                                % repaint go board
    /paintclient win send
cdef execute () makewin                                       % execute program
```

Figure B-6 *go/go6.c*

```
#include "go6.h"

enum    stone {CROSS=0,BLACK=1,WHITE=2};
static  enum stone board[BOARD_SIZE][BOARD_SIZE];

enum stone pickastone ()
{
    extern int rand();
    int i = rand() % 7;
    return (i>2 ? CROSS : (enum stone) i);
}

main()
{
  int x, y, id, cmd;

  if (ps_open_PostScript() == 0 ) {
    fprintf(stderr,"Cannot connect to NeWS server\n");
    exit(1);
  }

  initialize();

  execute();

  while (!psio_error(PostScriptInput)) {
    if (get_black(&id, &x, &y)) {
       if (board[x][y] == BLACK) {
          board[x][y] = CROSS;
          cross(id,x,y);
       } else {
          board[x][y] = BLACK;
          black_stone(id,x,y);
       };
    } else if (get_white(&id, &x, &y)) {
       if (board[x][y] == WHITE) {
          board[x][y] = CROSS;
          cross(id,x,y);
       } else {
          board[x][y] = WHITE;
          white_stone(id,x,y);
       };
    } else if (get_menu(&cmd)) {
       for (x = 0; x < BOARD_SIZE; x++)
          for (y = 0; y < BOARD_SIZE; y++)
             board[x][y] = (cmd==FILL_CMD ? pickastone():CROSS);
       repaint();
    } else if (get_damage(&id)) {
       draw_board(id);
       for (x = 0; x < BOARD_SIZE; x++)
          for (y = 0; y < BOARD_SIZE; y++)
             switch(board[x][y]) {
               case BLACK:  black_stone(id,x,y);
                            break;
               case WHITE:  white_stone(id,x,y);
```

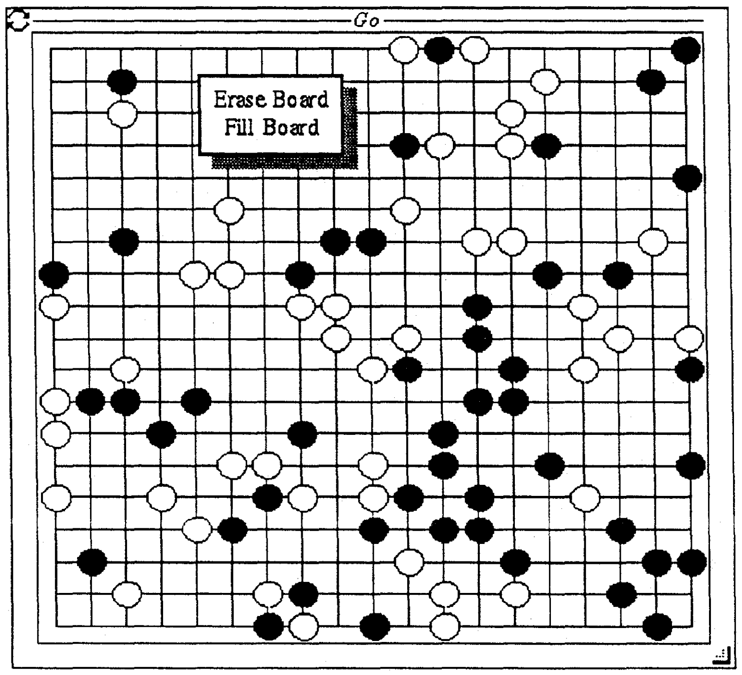
```
                                        break;
                    }
               repaired(id);
          } else if (done() || psio_eof(PostScriptInput))
               break;
     }
     ps_close_PostScript();
     exit(0);
}
```

B.4. The Board Plus Menu

The go board now has a menu which will allow you to either erase all the stones or to fill it with a pseudo-random pattern of black and white stones.

When the window is represented by an icon the icon will display the same pattern as the board does. Here is the board with the menu:

Figure B-7 *the go board plus menu*



B.5. NeWS Operators, Methods and Keys

The (UI) label means that these window methods are generally not used by the client but are accessed primarily through the user interface of the window.

/flipiconic

– **/flipiconic** –
Alternate between opened (window) and closed (iconic) state. (UI)

/paint

– **/paint** –
Repaint the window or icon. If the window is open, **paint** calls both **/paintframe** and **/paintclient**. The default **/Damaged** handler sets the canvas clip to the damage region and calls **/paint** automatically. (UI)

self

– **self** instance
Used as the target object with **send**, **self** refers to the instance that caused the current method to be invoked. It does *not* refer to the class the method is defined in. The **self** primitive can also be used anywhere to refer to the currently active instance.

Index