HyperLook

Copyright notice

©1992 The Turing Institute Limited ALL RIGHTS RESERVED

Copyright in the whole and every part of this manual belongs to The Turing Institute Limited ("The Institute"). No part of this manual may be copied transmitted or reproduced in any material form (including photocopying or storing it in any medium or by electronic means and whether or not transiently or incidentally to some other use of this manual) without the written permission of The Institute.

This manual was written using the NeWS version of FrameMaker 2.1.1.

The following are trademark or registered trademarks of their respective companies or organizations:

HyperLook, HyperLook logo / The Turing Institute Limited
Turing Institute Logo / The Turing Institute Limited
Sun Microsystems, NeWS / Sun Mircosytems Incorporated
OpenWindows, SunOS, Solaris / Sun Microsystems Incorporated
Adobe, PostScript / Adobe Systems Incorporated.
OPENLOOK, UNIX / UNIX system Laboratories Incorporated
FrameMaker / Frame Technology Corporation
SmallTalk / Xerox Corporation
Helvetica, Times / Linotype Corporation
ITC Zapf Dingbats / International Typeface Corporation
Lucida / Bigelow and Holmes
Puma / Staubli Unimation A.G.

The Turing Institute Limited

George House

North Hanover Street

Glasgow G1 2AD, Scotland

Tel: +44 41 552 6400

Fax: +44 41 552 2985





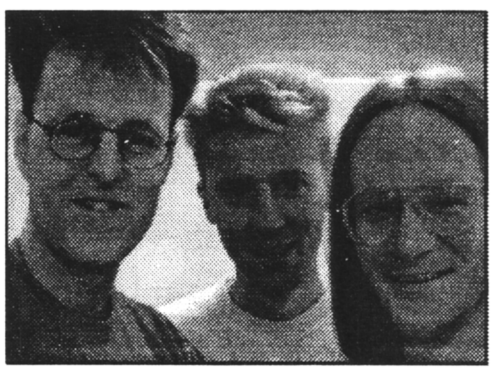
Acknowledgements

I would like to thank Don Hopkins for his hard work and late hours, Dug Scoular for his support, Tim Niblett for his expert advice, Rafael Bracho for his extremely useful feedback and Marleen for being so patient. I would also like to thank Jim Rudolf, Suhayya Abu-Hakima, and Danny Pearce for their work on earlier systems.

Many thanks to the hundreds of users who have provided the feedback which has been essential to the success of HyperLook.

Have fun,

Arthur van Hoff



Arthur, Dug and Don.

Table of contents

Chapter 1 Introduction

Open windows with HyperLook 1-2

Introducing HyperLook 1-3
Stacks and Cards 1-3
Desktop Management 1-3
Graphics Editing 1-4
Rapid Application Development 1-5
Interactive Design 1-5
Design by Example 1-6
Providing Help 1-7
Extensibility 1-7
Customizing Components 1-8
Object Oriented 1-9
Interfacing to Other Languages 1-9
Platforms 1-10

Background 1-11
HyperLook 1-11
The Turing Institute 1-11

Reading this Manual 1-13
The Mouse 1-13

Chapter 2 Getting Started

Starting HyperLook 2-3

HyperLook Basics 2-5
The SystemStatus Stack 2-6
Exiting HyperLook 2-6

Trouble Shooting 2-8
Starting OpenWindows 2-8
Running HyperLook 2-8

Using HyperLook Every Day 2-9
Using the File Manager 2-9
Setting Up Your Environment 2-10
HyperLook on Color Screens 2-11
HyperLook and Multiple Screens 2-12



Chapter 3 Using HyperLook

Using Help 3-3

Use of the Mouse 3-4

Using Menus 3-5
- Pop Up Menus 3-5
- Pull Right Menus 3-5
- Menu Key Accelerators 3-6

Using Stacks 3-7
- The Home Card 3-7
- Moving Between Cards of a Stack 3-8
- Every Window is a Stack 3-8
- Opening Stacks 3-8
- The Stack Menu 3-9
- Moving Stacks 3-9
- Bringing Stacks to the Front 3-10
- Iconifying Stacks 3-11
- Resizing Stacks 3-11
- Redrawing Stacks 3-12
- Zapping Stacks 3-12

Using HyperLook Components 3-13
- Using Buttons 3-13
- Using Checkboxes 3-13
- Using Scrollbars 3-14
- Editing Text 3-14
- Scrolling Lists 3-16
- Pulldown Buttons 3-17
- Color Selectors 3-17

The Clipboard 3-18

The Color Pallet 3-20

The Stack Manager 3-22

The Resource Manager 3-23
- User Resources 3-23
- Adding Resource Directories 3-24
- Removing Resource Directories 3-24
- Application and System Resources 3-24
- Locating a Resource 3-25

Handling Files 3-27
Changing Directories 3-27
Opening Files 3-28
Saving Files 3-29
Selecting Directories. 3-30
Short Cuts 3-30

The System Properties. 3-31
Changing the System Fonts 3-31
Specifying a Printer 3-32
Scrolling Parameters 3-32
Logging Messages 3-33
Report Errors 3-33

The OpenWindows Function Keys. 3-35

The Stack Menu 3-36

Chapter 4 Graphics Editing

Getting Started 4-4
The Tool Pallet 4-5
Loading Drawings 4-6
Saving Drawings 4-6
Printing Drawings 4-7
Zooming. 4-8
Moving Around 4-8
Selecting Objects 4-9

Editing Objects 4-11
Creating Objects 4-11
Creating Multiple Objects 4-12
Moving Objects 4-13
Changing the Size of Objects 4-13
Rotating Objects. 4-14
Constrained Editing 4-15
Using the Grid 4-15
Aligning Objects 4-16
Selecting the Fill and Line Color 4-17
Selecting the Line Width. 4-18
Selecting Arrowheads 4-19
Front to Back Order 4-20
Grouping 4-20

Using the Clipboard 4-22
Copying and Pasting Objects 4-22
Duplicating Objects 4-22
Deleting Objects 4-23
Undo 4-23

Special Objects 4-24
Text Objects 4-24
Pie Objects 4-25
Rounded Corner Rectangles 4-26
Polygon and Spline Objects 4-26
Image Objects 4-29
Encapsulated PostScript Objects 4-30
Solid Objects 4-31
Clipped Objects 4-32

The Graphics Editor Menus 4-34

Chapter 5 Editing Stacks

Designing Your Own Stack 5-2
Edit Mode 5-3
The Stack Menu in Edit Mode 5-4

Editing and Saving Stacks 5-5
Creating a New Stack 5-5
Saving Stacks 5-6
Reverting Stacks 5-6
Throw Away Changes? 5-7
Printing Cards 5-7
Stack Properties 5-8
Stack Properties Explained 5-9
Designing the Shape of a Stack 5-10

Creating New Objects 5-12
Using the Object Warehouse 5-13

Editing Objects 5-15
Selecting Objects 5-15
Moving Objects 5-16
Sizing Objects 5-17
Constrained Editing 5-17
Aligning Objects 5-18
Front to Back Order 5-18



Using the Clipboard 5-20
Copying and Pasting Objects 5-20
Duplicating Objects 5-21
Deleting Objects 5-21
Copying and Pasting Object Properties 5-22
Pasting Text and Graphics 5-22
Copy as Drawing 5-23

Cards and BackGrounds 5-24
Adding and Deleting Cards 5-25
Card Properties 5-25
Card Properties Explained 5-26
Copying Cards to the Clipboard 5-26
Using BackGrounds 5-26
BackGround Properties 5-27

Defining Help 5-29
Editing Help Text 5-30
Looking for Help 5-30

Black and White Stacks 5-32

Customizing Your Editing Environment 5-33
Stack Editor Properties 5-33
Stack Editor Properties Explained 5-33
Creating Your Own Untitled Stack 5-34
Creating Your Own Object Warehouse 5-34

Edit Mode Menus 5-36

Chapter 6 Object Properties

Generic Properties 6-3
Editing Properties 6-4
Object Name Property 6-4
Layer Property 6-5
Glue Property 6-5
Position and Size Properties 6-6
Visible Property 6-6
Font Property 6-7
Color Properties 6-7



Button Properties 6-8
Push Buttons 6-9
Transparent Buttons 6-9
Checkboxes 6-10
Drawing Buttons 6-10
Alternating Drawing Buttons 6-10

Field Properties 6-12
Field Properties Explained 6-12
Numeric Fields 6-13
Controlling the Input Focus 6-13

Text Properties 6-15
Text Properties Explained 6-15

List Properties 6-18
List Properties Explained 6-19

PullDown Properties 6-20
PullDown Properties Explained 6-21
Pull Right Menus in PullDown Objects 6-21

Slider Properties 6-23
Slider Properties Explained 6-23

ColorSelect Properties 6-25
ColorSelect Properties Explained 6-25

DrawTool Properties 6-26
DrawTool Properties Explained 6-27

Chapter 7 Scripting

Editing Scripts 7-3
An Example of a Script 7-4

The PostScript Language 7-5
Postfix Notation 7-5
Stack Operations 7-5
Notation 7-6
More Information on PostScript 7-7

Classes 7-8
Super Classes and Sub Classes 7-9

The Documentation Browser 7-10
Using the Documentation Browser 7-10
Printing Class Documentation 7-11



Messages 7-12
Messages and Methods 7-12
The Message Hierarchy 7-13
Finding Out About Messages 7-15

Writing Scripts 7-16
The Action Method 7-16
Manipulating Stacks 7-18
Sending Messages 7-19
Addressing Objects 7-20
Linking Buttons to Cards 7-21
The SetValue Method 7-22
Getting the Value of an Object 7-24
The OnHelp Method 7-24
The Show and Hide Methods 7-25
The OnOpen and OnClose Methods 7-25
Defining Your Own Methods 7-26
Using IncludeScript 7-26

Scripting Examples 7-28
Fahrenheit and Celsius 7-28
Listing Mail Messages 7-30
Animated Slider 7-31

Dealing with Errors 7-33
Syntax Errors 7-34
Invalid Names 7-34
Execution Errors 7-34

Unix Scripting Utilities 7-35
The HyperLook PostScript Shell 7-35
Sending Messages from the Shell 7-35

Chapter 8 Client Programming

The HyperLook Client Interface 8-2
High Level Message Passing 8-2

Client Interface Overview 8-4
Initializing the Client Interface 8-4
Connecting to Stacks 8-5
Receiving Messages from HyperLook 8-5
Sending Messages to HyperLook 8-6
PostScript Data Structures in C 8-7

The Hamburger Client 8-8
Creating the Hamburger Stack 8-8
The Hamburger Code 8-9
The Hamburger Code Explained 8-10
HyperLook Include File 8-10
Debugging 8-11
Initialization 8-11
Registration 8-12
Notification 8-12
Message Handlers 8-13
Cleanup 8-14
Compiling the Hamburger 8-14

The UserName Client 8-15
The UserName Code 8-16
The UserName Code Explained 8-17
Message Handler Arguments 8-18
Compiling the UserName client 8-18

Appendix A Glossary

Appendix B Methods

HyperLook Methods and Variables B-2

Stack Methods B-3

Card Object Methods and Variables B-4
Button Methods B-4
Field Methods B-5
Text Methods B-5
List Methods B-5
PullDown Methods B-5
Slider Methods B-5
DrawTool Methods B-5
ColorSelect Methods B-6

Appendix C Unix commands

drawps(1) C-2
drawps

exithyperlook(1) C-4
exithyperlook
hlpath(1) C-5
hlpath
hlpsh(1) C-6
hlpsh
hlsend(1) C-7
hlsend
hyperdraw(1) C-8
hyperdraw
hyperlook(1) C-9
hyperlook
install_hyperlook
showstack(1) C-12
showstack
hidestack

Appendix D Library functions

hl_alloc(3) D-2
hl_alloc
hl_destroy
hl_use
hl_free
hl_constant
hl_any(3) D-3
hl_any
hl_new_null
hl_new_boolean
hl_new_number
hl_new_string
hl_new_string_body
hl_new_name
hl_new_array
hl_new_array_body
hl_exists(3) D-6
hl_exists

hl_flush(3) D-7
hl_flush
hl_flush_input
hl_get(3) D-8
hl_get
hl_put
hl_listen(3) D-10
hl_listen
hl_register
hl_register_any
hl_register_quit
hl_register_timeout
hl_register_ioerror
hl_path(3) D-14
hl_path
hl_print(3) D-15
hl_print
hl_ps(3) D-16
hl_ps
hl_send(3) D-17
hl_send
hl_send0
hl_show(3) D-19
hl_show
hl_hide
hl_connect
hl_start(3) D-21
hl_start
hl_stop
hl_verbose
libhl.a(3) D-24
libhl.a

Is Index



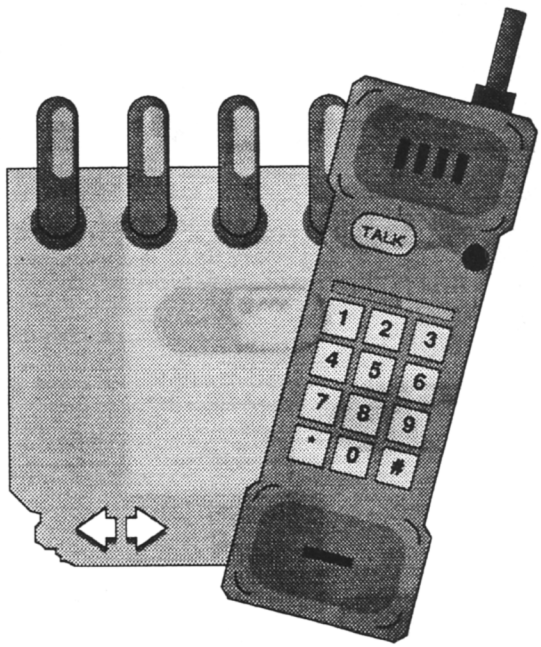
1 Introduction

Open windows with HyperLook

Welcome to the exciting world of HyperLook™ user interface design[1]. HyperLook is a powerful, easy to use development tool, and it's fun, too!

Figure 1

HyperLook is fun to use!



1. HyperLook is a trademark of The Turing Institute Limited.

Introducing HyperLook

This chapter gives you a whistle-stop tour of HyperLook. From it you'll get an idea of the things you can do with HyperLook and you'll be introduced to some of the terminology of the HyperLook world.

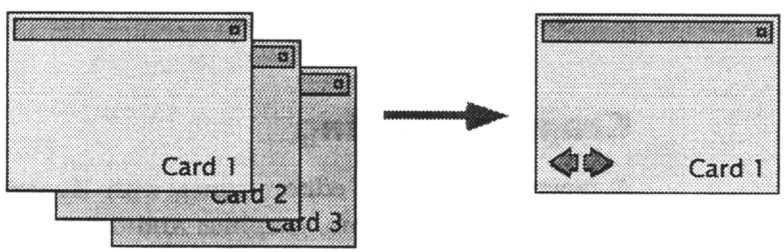
Don't worry if you don't understand every new term, they're all explained later. Now read on and find out what HyperLook can do...

Stacks and Cards

HyperLook uses the names **stacks** and **cards** instead of windows and window layouts. This widely used terminology was chosen to emphasize the fact that HyperLook stacks (windows) can have more than one card (layout).

Figure 2

Stacks can have multiple cards. You can get to them using the arrow buttons.



One card per window? Or all cards in one stack!

Each window layout is called a card, and may contain graphics and user interface components such as buttons, text and sliders. You can flip between cards like turning pages in a book.

Desktop Management

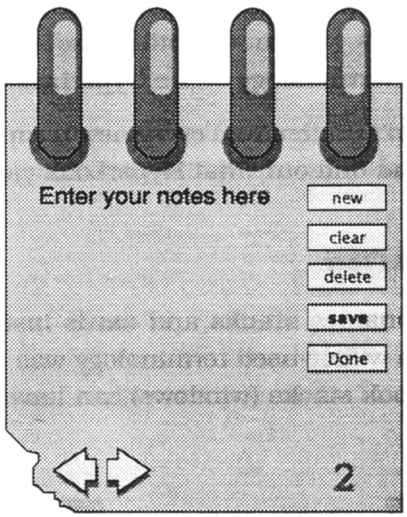
HyperLook is an easy to use environment. Any user of OpenWindows will find that it is easy to learn how to use HyperLook.

You are free to configure your working environment, and you can modify and extend the user interface.

HyperLook provides a comprehensive set of desktop tools including a clipboard, color pallet, notepad and many others.

Figure 3

The HyperLook Notepad stack.

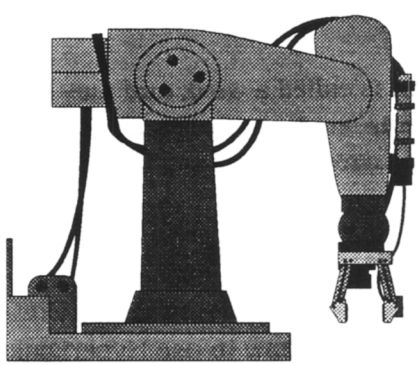


Graphics Editing

A powerful graphics editor comes with HyperLook. It lets you create sophisticated diagrams, graphics and other artwork, and it's simple to use.

Figure 4

An example of a drawing created with the HyperLook graphics editor.



Graphics created with HyperLook can be incorporated into other environments and used with other tools. For example, most of the artwork and diagrams in this manual were created using the PostScript® graphics editor and were then imported into FrameMaker™[1]. PostScript graphics can be imported into many desktop publishing tools.

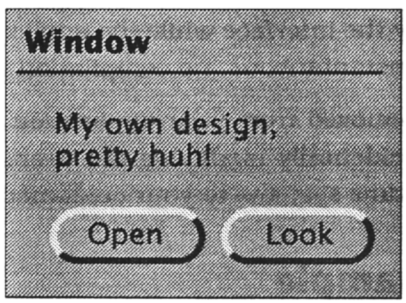
Rapid Application Development

HyperLook simplifies the task of building user interfaces. You can prototype user interfaces quickly by direct manipulation, without restriction of the creative process.

HyperLook is not restricted to a single look and feel. You can create a new look and feel for your application, or adapt an existing look and feel.

Figure 5

HyperLook can have any look and feel.



Interactive Design

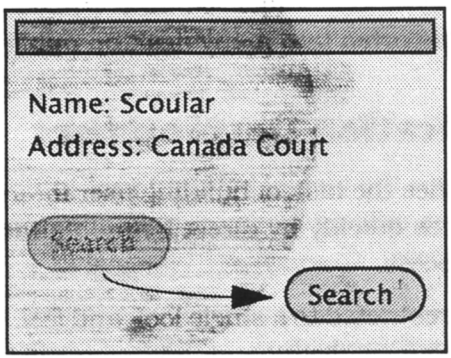
The interface designer can do much of the interface development by direct manipulation: by drawing, dragging and sizing objects with the mouse, without writing a line of code.

1. PostScript is a registered trademark of Adobe Systems Incorporated, FrameMaker is a trademark of Frame Technology Corporation.

One of the most time consuming aspects of creating interactive applications is the design of the interface. Even non-programmers can design applications using direct manipulation in HyperLook.

Figure 6

Design your interface by direct manipulation. Pick up a button, move it, and go!



You can even modify the interface while the application is alive and running! Changes are instantaneous, no compilation is needed!

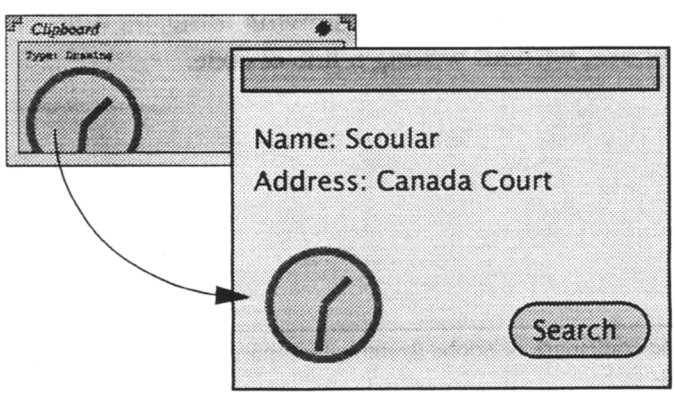
The editor can be removed from your application to stop users of your application from accidentally modifying the user interface. In this way you can deliver runtime systems to your customers.

Design by Example

Not only can you design your own user interface, but you can also use components designed by others. By copying and pasting components, you can insert them into your application, complete with their functionality.

Figure 7

Pasting a clock into your application.



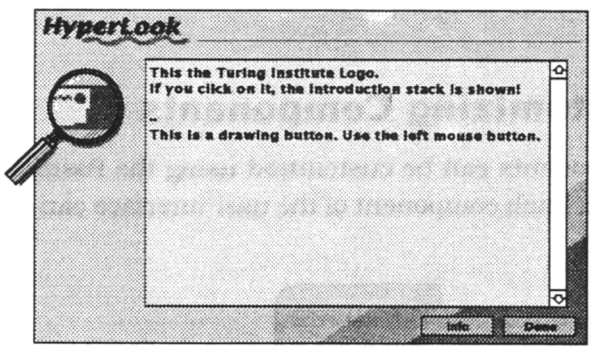
HyperLook provides example interfaces and components which you can plug right into your application.

Providing Help

Not only does HyperLook provide an on-line help system which helps you to navigate through the system, it also lets you define your own help for the users of your application. No programming is required!.

Figure 8

The HyperLook Help stack.



Help can be provided as text but you can also design your own help stack. It can use graphics and all the other functionality provided by the HyperLook system.

Extensibility

A comprehensive set of user interface components like buttons, scroll-bars and pull-down menus is provided. This set can be extended and tailored to the needs of any application.

HyperLook is not limited to the components it provides. You can modify existing components or add completely new ones. You can draw new components, customize existing ones, or create entirely new objects with their own interactive behaviour.

It is possible to design and implement a corporate style, or to design a new set of components for your environment. For example, you could make a set of components for use in a process control environment. Once tested they can be used to build industrial process control applications.

Figure 9

Designing a knife switch by drawing the on and off positions.

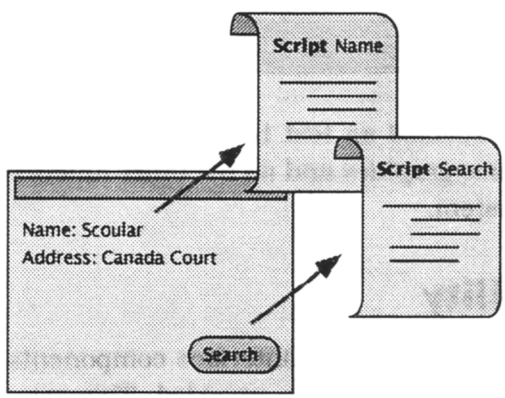


Customizing Components

Components can be customized using the PostScript programming language. Each component of the user interface can have a *Script* associated with it.

Figure 10

Every component can have a Script.



Scripting (programming user interface components) lets you to change the component's look and feel.

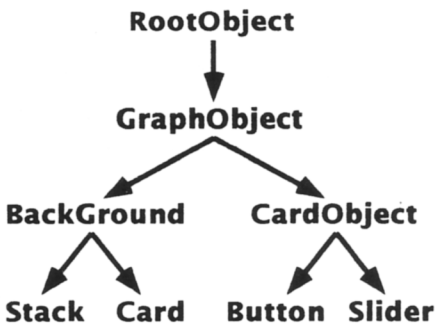
There is a C-like scripting language available for HyperLook, called PdB. PdB is the ideal scripting language if you are already familiar with C. You can write new component classes in PdB too!

Object Oriented

HyperLook is completely object oriented. All components are implemented as classes which are defined using a classing mechanism similar to that of SmallTalk™[1].

Figure 11

Functionality is Inherited using classes.



An easy to understand message passing mechanism is used for communicating between objects and communicating with client programs.

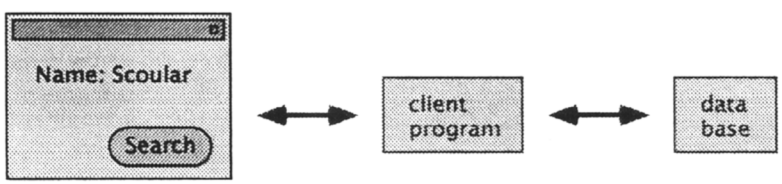
It is possible to add your own classes to the system. The classing mechanism provides multiple inheritance, persistent variables, dynamic loading, and other features useful for the creation of dynamic object oriented user interfaces.

Interfacing to Other Languages

HyperLook allows any UNIX® process to access it through an easy to use C programming interface[2]. Interfaces to other languages such as C++, Prolog, Lisp and Perl are available.

Figure 12

Linking a database client to HyperLook.



1. SmallTalk is a trademark of Xerox Corporation.
2. UNIX is a registered trademark of Unix system Laboratories Incorporated.

Any program that can link in a small C library is able to use the HyperLook interface. The C programming interface lets you create interfaces to complex applications such as database managers, expert systems, games, etc.

Platforms

HyperLook uses the OpenWindows™ system and runs on SunOS™ and Solaris® platforms[1]. It integrates into the existing Sun desktop. It makes use of the Sun desktop tools such as the file manager and the mail tool.

HyperLook provides complete access to the powerful PostScript capabilities of the OpenWindows environment.

1. OpenWindows, SunOS and Solaris are trademarks or registered trademarks of Sun Mircosystems Incorporated.

Background

HyperLook

The development of HyperLook began in October 1987 at The Turing Institute, and the initial system was completed in April 1988.

The programming and detailed design of HyperLook was done by Arthur van Hoff of The Turing Institute.

After 5 years of development and testing, worldwide distribution and 300 sites using the system, HyperLook was released as a commercial product in June 1992.

The Turing Institute

The Turing Institute is an independent company specializing in the research, development and commercial exploitation of the latest advances in computer software. The company was formed in 1983 in honor of Alan Turing, whose pioneering work helped provide a formal basis for present day computing and artificial intelligence.

Figure 13

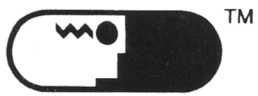
Alan M. Turing, 1912-1954



The Institute's activities are sharply focused on solutions which allow computers to interact with both humans and their environments. Such a strategy has led to a number of important developments and products in areas as diverse as user-interfaces, document image processing, decision support, fault diagnosis, machine learning, advanced robotics, computer vision and intelligent documentation.

Should you require any further information about the Turing Institute or about any of its technologies or products, please contact:

The Turing Institute Limited[1]



George House

North Hanover Street

Glasgow G1 2AD, Scotland

Tel: +44 41 552 6400

Fax: +44 41 552 2985

1. The Turing Institute logo is a trademark of The Turing Institute Limited.

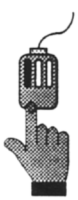


Reading this Manual

This manual provides a complete overview of the HyperLook system.

- Using HyperLook
  - **Getting Started** explains how to start HyperLook and how to set up your environment.
  - **Using HyperLook** explains how to use HyperLook stacks and components. It explains how to use the tools which are provided by HyperLook.
  - **Graphics Editing** explains how to use the HyperLook graphics editor.
- Editing HyperLook stacks
  - **Editing Stacks** explains how you create your own stacks and cards and how you edit components.
  - **Object Properties** explains the properties of the HyperLook objects.
- Programming HyperLook
  - **Scripting** explains how to write scripts for objects.
  - **Client Programming** explains how to communicate with HyperLook from UNIX programs.

The Mouse



Where this manual says *click the mouse* or *press the mouse* you should use the **left** mouse button. HyperLook uses the left mouse button most of the time, that is why the mouse button specification is usually omitted.

If you are meant to use the **middle** or **right** mouse button, it will be specified explicitly.

2 Getting Started

This chapter explains how to start HyperLook and how to exit it. Once you know how to start HyperLook, you can move on to the next chapter, which explains how to use the HyperLook user interface.

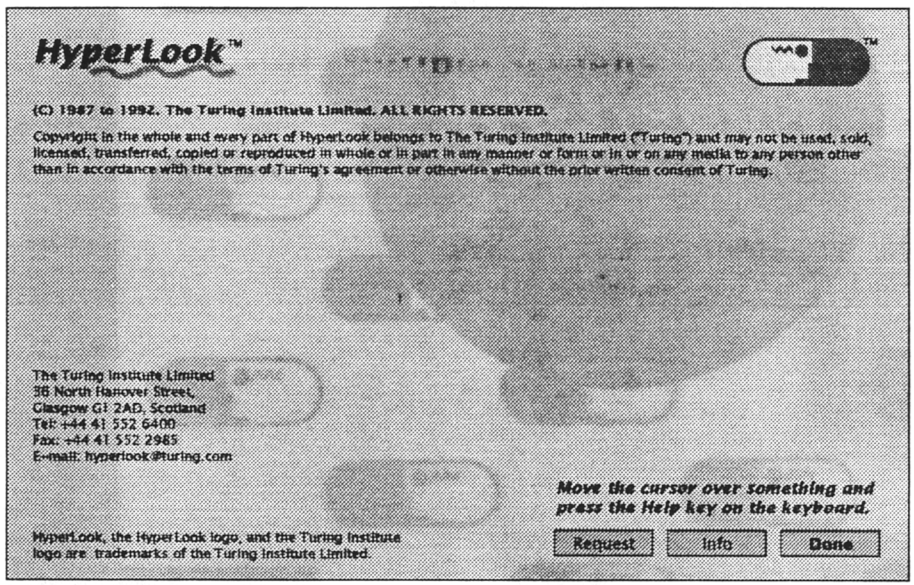


Figure 14

The introduction stack. It is shown when you start HyperLook.

Starting HyperLook

Before using HyperLook, it must be installed on your computer. Refer to the *Installation Guide* which you received with HyperLook for installation instructions[1].

Once HyperLook is installed, you must make sure that you have access to the directory where it lives. This directory is called the HyperLook home directory.

HyperLook is usually installed in the directory **/usr/src/hyperlook**. If this directory does not exist on your system, ask your system administrator if who will tell you where HyperLook is installed.

You start HyperLook with the **hyperlook** command. Assuming that HyperLook is installed in **/usr/src/hyperlook**, you can start HyperLook by typing the following command in the UNIX shell:

```
/usr/src/hyperlook/bin/hyperlook
```

Below is an example of the HyperLook startup procedure. The things that you type are in bold, what the computer types is in italics.

```
Where is OpenWindows installed?

Enter OPENWINHOME: /usr/src/openwin3

Where is HyperLook installed?

Enter HLHOME [/usr/src/hyperlook]:<Return>

Starting OpenWindows version 3.0
Using OPENWINOPTIONS = -cubesize large
Using OPENWINHOME = /usr/src/openwin3
Using HLHOME = /usr/src/hyperlook
```

If you have not started OpenWindows, HyperLook will ask you where OpenWindows is installed, so it can start the window system for you. If you don't know where it's installed, refer to your system administrator.

1. You can find more information on how to install HyperLook in the INSTALL file in the HyperLook home directory.



If all goes well, the Introduction Stack will appear in the center of your screen after 30 seconds or so. Figure 14 on page 2-2 shows a picture of this stack.

If you don't succeed in starting HyperLook you should read "Trouble Shooting" on page 2-8.

Read "Using HyperLook Every Day" on page 2-9 if you want to install HyperLook more permanently.

Now that you have loaded the Introduction stack, click the mouse button (the left one) on the **Done** button, this will remove the Introduction stack, and show the **system** stack and the **SystemStatus** stack.

Press the **Request** button if you want to request more information by electronic mail. Press **Info** for more information on HyperLook.

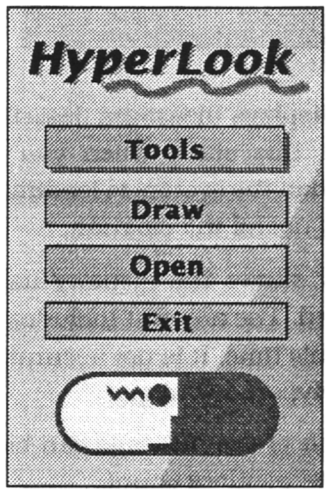


HyperLook Basics

The system stack contains buttons which give you access to functions that you will use frequently.

Figure 15

The HyperLook system stack.



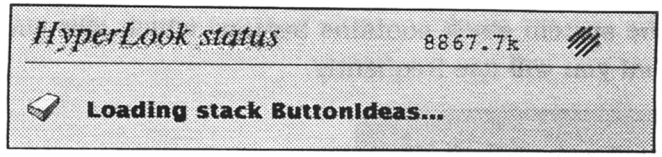
The buttons on the system stack provide the following functions:

- **Tools** Access frequently used tools (see Chapter 3, "Using HyperLook").
- **Draw** Open a HyperLook graphics editor (see Chapter 4, "Graphics Editing").
- **Open** Open or create a stack (see "Opening Stacks" on page 3-8).
- **Exit** Exit the HyperLook system (see "Exiting HyperLook" on page 2-6).

The SystemStatus Stack

Figure 16

The SystemStatus stack.



The **SystemStatus** stack displays messages describing what HyperLook is doing. Keep an eye on this stack when you are learning to use HyperLook. It shows you when the system is loading stacks, when a loading error occurs, and other useful information.

The top right hand corner shows the memory used by OpenWindows since HyperLook was started. The amount includes the memory used by other applications during this time. It is not accurate in determining how much memory is used by HyperLook alone!



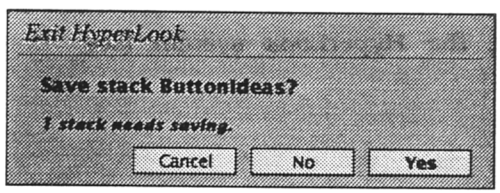
The eraser button to the left of the message can be clicked to erase the message. It also updates the memory count.

Exiting HyperLook

After using HyperLook you can exit by pressing the **Exit** button in the system stack. This will show the **ExitHyperLook** stack. If you have been editing stacks, you will be asked whether you want to save the stacks that you have changed before exiting.

Figure 17

Do you want to save changes before exiting?



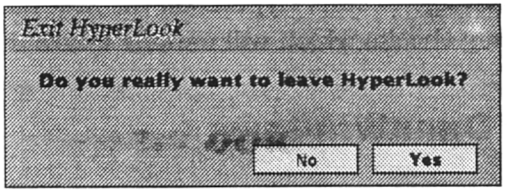
You can save a stack by pressing the **Yes** button, discard changes to the stack by pressing the **No** button, or decide not exit at all by pressing the **Cancel** button.



Once you are sure you want to exit HyperLook, press the **Yes** button. You can still decide not to exit HyperLook by pressing the **No** button.

Figure 18

Really exit HyperLook?



Trouble Shooting

If you don't manage to start HyperLook the first time: *don't panic*. There are several easy checks which will usually locate the problem.

Starting OpenWindows

1. Make sure you are using the correct version of OpenWindows. Refer to the installation instructions to find out which versions of OpenWindows are supported.
2. Conflicting initialization files in your home directory (some programs create these files without asking you), in particular files called **`.user.ps`**, **`.startup.ps`** and **`.init.ps`**[1], may set your environment up incorrectly. Try removing these files and then start HyperLook as described in "Running HyperLook" on page 2-8.

If you are using a version of OpenWindows that is supported by HyperLook and the above does not work then you should try to start OpenWindows from scratch. Refer to the OpenWindows installation guide if you have any problems doing so.

You may want to remove **`.xinitrc`** from your home directory if you are running a new version of OpenWindows for the first time. OpenWindows requires certain initializations which are not standard for X window systems.

Running HyperLook

Once OpenWindows has started, open a console window and try running HyperLook from that window as described in "Starting HyperLook" on page 2-3. Pay attention to error messages appearing in the console window.

If the HyperLook introduction stack does not appear, ask your system administrator to make sure that HyperLook is installed correctly.

1. To list files all files in your home directory use: **ls -a ~**

Using HyperLook Every Day

If you are using HyperLook a lot (and why shouldn't you?) you may want to take a few more steps so that it's really easy to start and stop HyperLook at any time.

Using the File Manager

The Sun desktop environment provides a file manager tool which provides an iconic representation of your file system. The Sun documentation tells you how to use the file manager. You can use the file manager to start HyperLook and to open stacks and drawings.

Before you can use the file manager with HyperLook you need to install the HyperLook icons. To install them execute the following command:

```
/usr/src/hyperlook/bin/install_hyperlook
```

Once you have done this you must exit the file manager and restart it. It will now recognize HyperLook files.

Figure 19

The HyperLook icon and a stack icon.

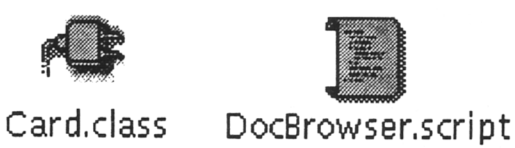


You can start HyperLook by double clicking the HyperLook icon, or you can open a stack by double clicking a stack icon.

The class icon represents a HyperLook user interface object, it contains code for the object. It cannot be opened by itself. A script contains code for a particular user interface object. It can be opened for editing.

Figure 20

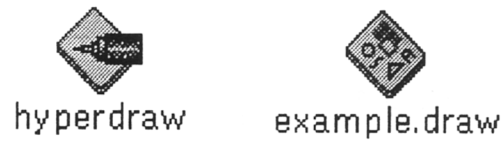
The HyperLook object class icon and the script icon.



Double clicking the hyperdraw icon starts the graphics editor. A drawing can be opened by double clicking a drawing icon (see Chapter 4, "Graphics Editing").

Figure 21.

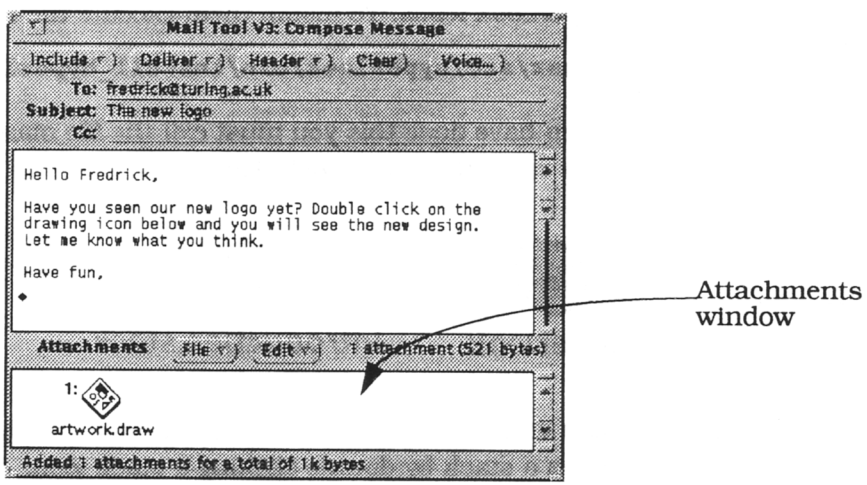
The HyperDraw icon and a drawing icon.



If you use the Sun desktop mail tool, you can drag stacks and drawings into the mail tools attachments window when composing a message to another user. This lets you to easily mail stacks and drawings to other users.

Figure 22

Using the mail tool to mail a drawing to another user.



Setting Up Your Environment

You may want to install HyperLook into your environment permanently. This involves changing your **`.login`** file and setting some environment variables.

You don't need to set your environment up to run HyperLook. HyperLook can set your environment up for you.

If you want to set up your environment, then you must edit your **.login** file to include the following lines:

```
# This is where OpenWindows lives
setenv OPENWINHOME /usr/openwin

# This is where HyperLook lives
setenv HLHOME /usr/src/hyperlook

# Make HyperLook executables accessable
set path=($HLHOME/bin $path)
```

Note that the above is only an example and that the directories may not be the same on your system.

Once you have changed your **.login** file, log out and log in again. You can then start HyperLook either from your login shell or from a window by typing **hyperlook**.

When starting HyperLook from your login shell it will start OpenWindows for you. You can specify parameters to be passed to OpenWindows. This is necessary if you want to run OpenWindows on multiple screens, or if you want to run OpenWindows with the **-nosunview** or **-includedemos** option.

You can specify options in the **OPENWINOPTIONS** variable. Add the following to your **.login** file:

```
setenv OPENWINOPTIONS "-nosunview -includedemos"
```

Refer to the OpenWindows documentation for other options that you can specify when starting OpenWindows.

HyperLook on Color Screens

To display colors correctly, HyperLook needs to run in an environment with a large color cube size. The color cube size is selected by the OpenWindows **-cubesize** large option. It is set as follows:

```
setenv OPENWINOPTIONS "-cubesize large"
```



HyperLook works fine if this option is not set but you may find that some of the colors are not correct. HyperLook supports both dynamic and static visual modes.

HyperLook and Multiple Screens

HyperLook uses the **DISPLAY** environment variable to connect to the OpenWindows server. This variable is automatically set when you startup OpenWindows.

After connecting to the server HyperLook determines whether the server has a color screen. If it has, then this screen will be used.

See the OpenWindows documentation how the **DISPLAY** environment variable can be used to run HyperLook remotely.

3

Using HyperLook

This chapter describes how to use the HyperLook user interface. HyperLook has its own user interface style, but it is just like any other application running on the OpenWindows desktop. That means that you can cut and paste from HyperLook to other desktop applications. The mouse is used is the same way, too!

This chapter explains how HyperLook is used in *browse* mode. This is the mode in which you normally use the system. The other mode is called *edit* mode. It is explained in Chapter 5, "Editing Stacks".

The best way to read this chapter is to get HyperLook started (see Chapter 2, "Getting Started") and try things out as you go through.

Using Help

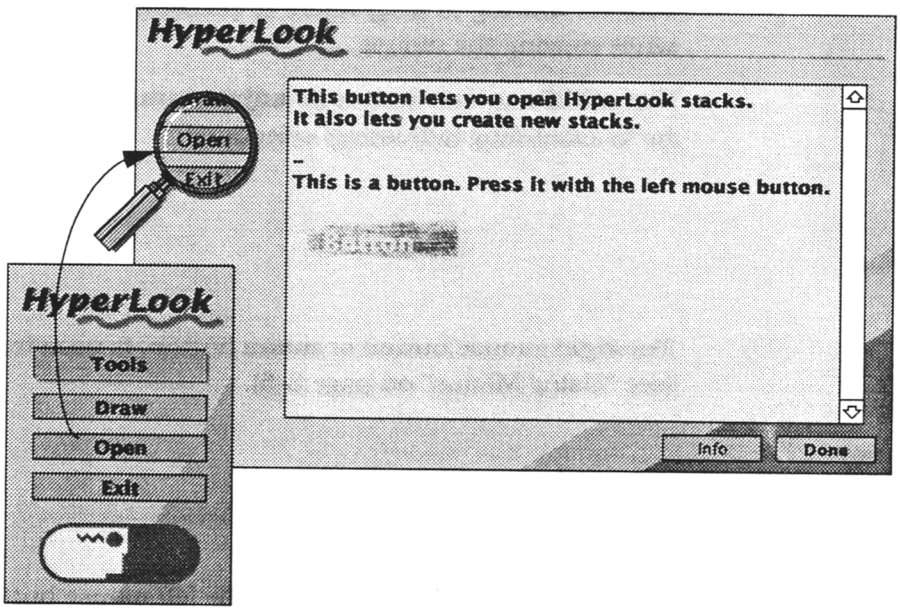


HyperLook has a built in help system which lets you get help about the user interface. To get help on something, move the mouse over the object and press the **Help** key on the keyboard[1].

Most components of the HyperLook user interface provide a short explanation of their functionality this way.

Figure 23

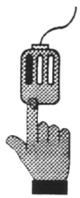
Getting help on the **Open** button of the system stack.



Remember to use Help if you get stuck. It provides useful information on most of the HyperLook components.

1. On most keyboards the **Help** key is on the bottom left.

Use of the Mouse

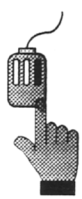


The **left** mouse button or **select** button is used in most cases. It is used to activate buttons, sliders, to select from pull down menus, to make selections etc.

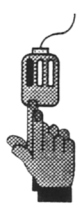
Unless otherwise specified, a *"mouse click"* or *"press the mouse"* refers to the left mouse button. A *click* refers to pressing and releasing the button without moving. A *drag* refers to pressing and holding down the button while moving the mouse.



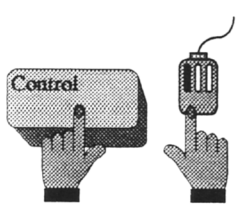
The **middle** mouse button or **adjust** button, is not used very often. It is for to extending (adjusting) selections.



The **right** mouse button or **menu** button, is used to show pop-up menus (see "Using Menus" on page 3-5).



To **double click** the mouse click the left mouse button twice in the same position.



You sometimes need to use the mouse while holding down a key on the keyboard. First hold down the key and then, while holding the key down, use the mouse.

Using Menus

Pop Up Menus

Holding down the right mouse button in any stack will pop up a menu. To select a menu item move the mouse over the desired item, while holding down the right mouse button.

Figure 24

The stack menu.



As you move over items, they highlight to indicate which one will be executed when the mouse button is released. Releasing the mouse over a selected item executes the item and the menu disappears.

If you don't want to select *any* item, move the mouse off the left edge of the menu until no item is selected and release the mouse button.

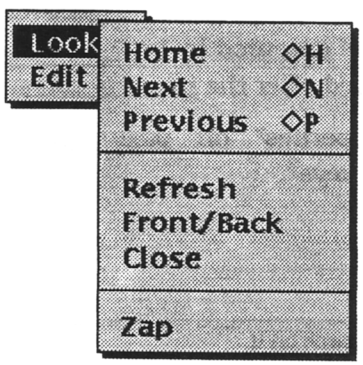
The HyperLook stack menu is explained in "The Stack Menu" on page 3-36.

Pull Right Menus

Some menu items have a ">>" sign on the right as in Figure 24. This means that they have a pull right sub menu. To show it, move the mouse over the ">>" sign.

Figure 25

Selecting a sub menu.



You can hide a pull right menu by moving the mouse over its left edge until it disappears.

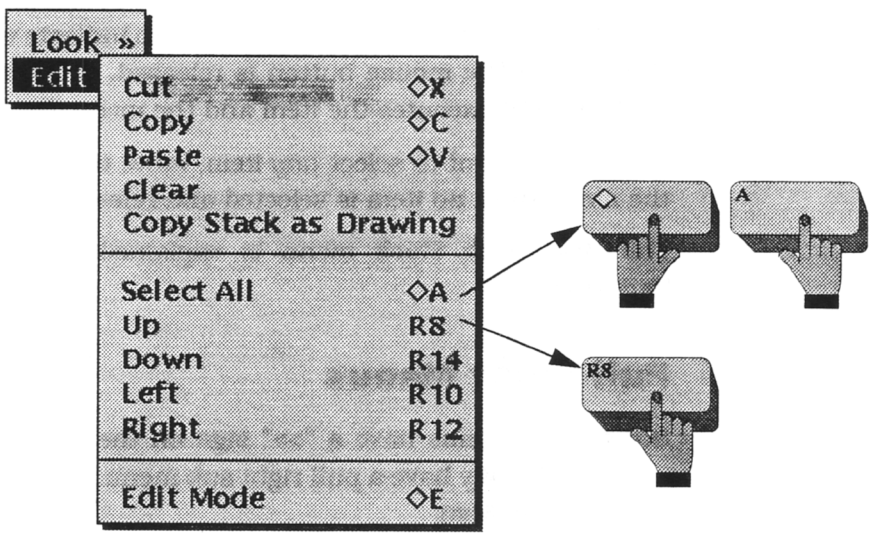
Menu Key Accelerators

Frequently used menu items have menu key accelerators, sometimes called keyboard accelerators. This lets you execute a menu item from the keyboard without showing the menu.

These menu items are displayed with a diamond and a letter. They are executed by holding down the meta key[1] and, while it is held down, typing the appropriate letter.

Figure 26

Menu key accelerators.



Some menu items can be executed by typing a function key[2]. The function key name is displayed after the menu item.

See "The System Properties" on page 3-31 on the use of the OpenWindows function keys[3].

1. The meta key on the Sun keyboard has a diamond shape on it.
2. Function keys are labelled F1 through F12 and R1 through R15.
3. The OpenWindows function keys are labelled Stop, Again, Props, etc.

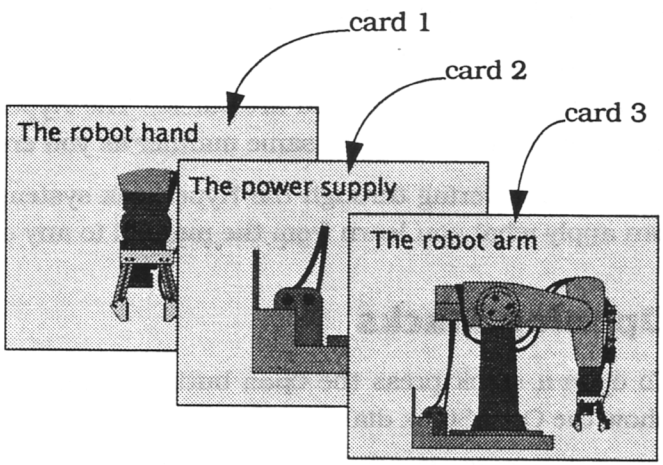
Using Stacks

A HyperLook window is called a *stack* (see also "Stacks and Cards" on page 1-3). The reason for this is that each stack may contain several cards just like a stack of cards. A card represents a layout of the stack.

Let's look at an example. A help system may provide help on several topics.This could be solved by designing a stack which contains a card for each topic. When help on a topic is needed the appropriate card of the help stack is shown.

Figure 27

The 3 cards of a Robot information system.



Each card can hold objects such as buttons but it can also hold graphics. You can change cards like turning pages of a book.

The Home Card

The first card of a stack is called the *home* card. When a stack is displayed for the first time, the home card is usually visible. The home card is often used as an index to the other cards in the stack.

Pressing the **Home** key on the keyboard takes you to the home card. The **End** key takes you to the last card of the stack.

Moving Between Cards of a Stack

Each stack can have more than one card. You can move to the next card of a stack by pressing the **PgDn** key on the keyboard. Pressing the **PgUp** key takes you to the previous card.

Going to the previous card from the home card takes you to the last card of the stack. The same is true in reverse, the next card after the last card in the stack is the home card.

Every Window is a Stack

While you learn to use HyperLook you will notice that all the HyperLook windows which are provided are stacks! The HyperLook system stacks were created in exactly the same manner as you create stacks.

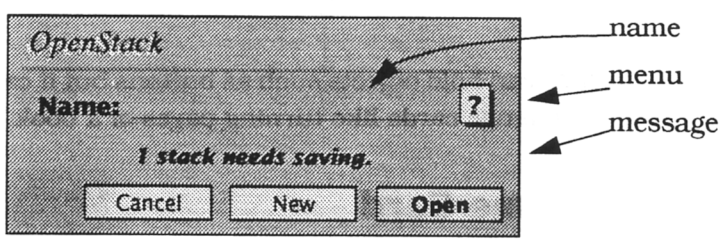
While maneuvering through the HyperLook system remember that you can apply what you learn from the manual to any of the system stacks.

Opening Stacks

To open a stack press the **Open** button in the system stack. This will show the OpenStack dialog.

Figure 28

The OpenStack dialog.



To open a stack, move the mouse over the OpenStack dialog and type the name of the stack and hit return. When the stack is loaded the dialog will disappear and the stack is shown. Try opening a stack called **Notepad**.

If the stack is not found, the message in the OpenStack dialog will say so. Refer to "The Resource Manager" on page 3-23 on how to open stacks from other Unix directories.



The "?" pulldown menu in the OpenStack dialog lists recently opened stacks. If you select a stack from this menu it will be opened (use the left mouse button). Stacks that have been changed are marked with a "*" character.

While typing the stack name you can type **Tab** to complete the name of the stack if it has been recently loaded.

Press **Cancel** to cancel if you don't want to open a stack after all. Press **New** if you want to create a completely new stack (see "Creating a New Stack" on page 5-5).

So now you can see how easy it is to explore HyperLook. Once you know the name of a system stack you can open it, exactly like a stack of your own.

The Stack Menu

Every stack has a pop up stack menu. This menu provides some basic functions to manipulate the stack. Figure 24 on page 3-5 shows the stack menu.

One word of warning, some objects have their own menu which is shown when holding down the right mouse button inside them. To show the stack menu make sure that you hold down the mouse over an empty region of the stack.

Moving Stacks

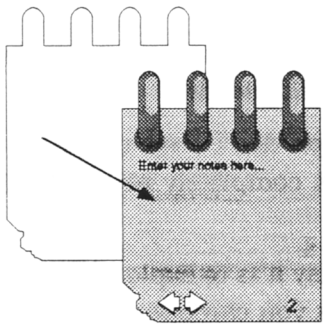
Most stacks can be moved to a new position on the screen by holding down the mouse button and dragging the stack to a new position. Go back and reread "Use of the Mouse" on page 3-4 if you are having trouble with the mouse buttons.



To move a stack, hold down the mouse in an unoccupied area of the stack or in a non-active object like the title of the stack. Clicking in an active object usually activates the object, instead of moving the stack.

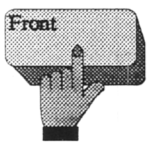
Figure 29

Moving a stack.



Some stacks can't be moved, so attempts to move them are ignored. Most of HyperLook's dialogs and property stacks can't be moved.

Bringing Stacks to the Front

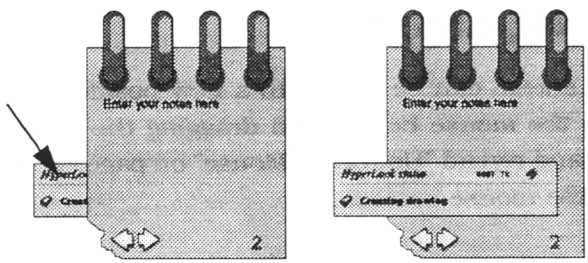


In a multi-window environment, stacks and window can cover each other, so sometimes you need to change their front to back ordering.

To bring a stack to the front, simply click in an unoccupied area of the stack. You can also bring it to the front, or send it to the back by pressing the **Front** key on the keyboard, while the mouse is anywhere over the stack.

Figure 30

Bringing a stack to the front.



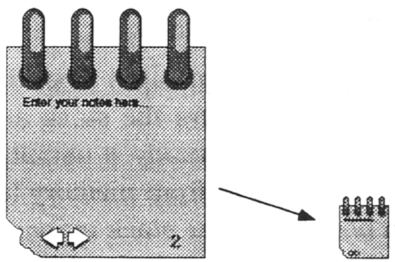
When you press the **Front** key and, the stack is already at the front, it will be sent to the back instead!

Iconifying Stacks

Stacks can be iconified. A stack icon is a miniature version of the stack, displayed at the bottom of the screen. An iconified stack does not accept any user input until it is de-iconified (opened).

Figure 31

Iconifying a stack.

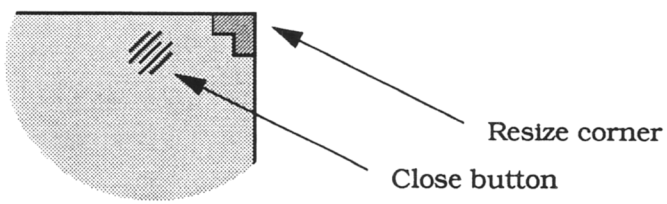




To iconify a stack, press the **Open** button on the keyboard. This might seem odd to you at first, but you'll soon get used to it. To open an icon you also use the **Open** button or you can double click on the icon.

Figure 32

A close button and resize corner.



Some stacks also have a close button. When you click on it, it iconifies the stack.

Resizing Stacks

Some stacks, such as the graphics editor, have resize corners. The resize corners lets you change the size of the stack.

Hold down the mouse in one of the resize corners, and drag it until the stack has the desired size.



Redrawing Stacks

Sometimes a stack needs redrawing. The stack can be redrawn by pressing the **Again** key on the keyboard while the mouse is over the stack.

Zapping Stacks

After finishing with a stack, you can zap it from the screen. Select **Zap** from the stack menu. Zapping clears the stack and the icon from the screen and from memory as well. Actually, it usually takes a few minutes before the stack is really removed from memory (see "The Stack Manager" on page 3-22 on how to use the stack manager).

Using HyperLook Components

This section will briefly discuss how to use the most common HyperLook components. Some components such as text and text fields have keyboard accelerators which you will find useful.

Using Buttons

Buttons are very easy to use. Hold the mouse down inside the button and the button will highlight. Releasing the mouse while the button is highlighted executes its action.

Figure 33

Pressing a button.



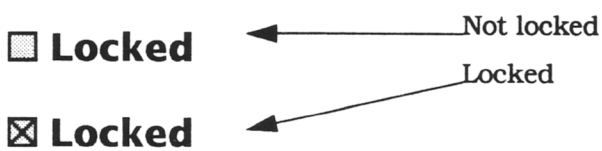
If you change your mind while holding down the mouse, move the mouse out of the button's area until it is no longer highlighted and release. Then the action is not executed.

Using Checkboxes

A checkbox is another type of button. It can be on or off. If the checkbox is crossed it means that it is on.

Figure 34

Examples of checkbox.



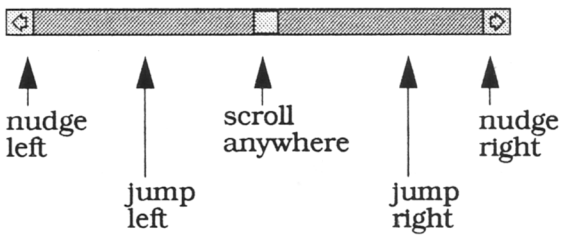
To change the state of the checkbox, click the mouse inside it.

Using Scrollbars

Scrollbars are used in text objects, graphics editors, lists etc. A scroll bar is very simple, it lets you scroll text or graphics in a horizontal or vertical direction.

Figure 35

An active horizontal scrollbar.



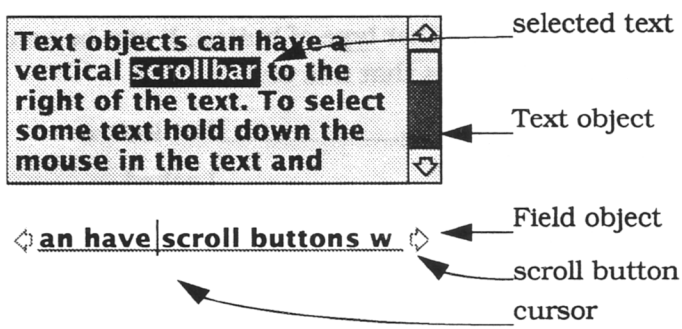
Horizontal and vertical scrollbars behave in the same way. The drag box of a scrollbar (the little box in the middle) lets you scroll to any position. Hold the mouse down in the drag box, drag it to the desired location and release.

Editing Text

Text objects are divided into single line text objects called *Fields* and multi line text objects simply called *Text*.

Figure 36

A text object and a field object.



Text objects can have a vertical scrollbar to the right of the text. Fields can have scroll buttons when the string which you are editing is longer than the field itself.

To select some text hold the mouse down at the beginning and, while holding the mouse down, drag the mouse to the end of the selection and release. The selected text will be highlighted.

See section "The System Properties" on page 3-31 on how to use the clipboard when editing text.

Text objects accept keyboard commands. These commands are listed below. Commands that assume multiple lines of text do not work for Field objects.

- **Control-A** -- go to the beginning of the line
- **Control-B** -- go to the previous character
- **Control-E** -- go to the end of the line
- **Control-F** -- go to the next character
- **Control-G** -- go to the end of the next word
- **Control-N** -- go to the next line
- **Control-P** -- go to the previous line
- **Control-Q** -- go one page up
- **Control-R** -- go to the beginning of the previous word
- **Control-S** -- go to the beginning of the text
- **Control-V** -- go down a page
- **Control-Z** -- go to the end of the text

You can also use the arrow keys (**R8**, **R10**, **R12** and **R14**) to move around in the text.

- **Control-C** -- delete the current line
- **Control-D** -- delete the next character
- **Control-K** -- delete until the end of the line
- **Control-O** -- split the current line
- **Control-U** -- delete until the beginning of the line
- **Control-W** -- delete the previous word



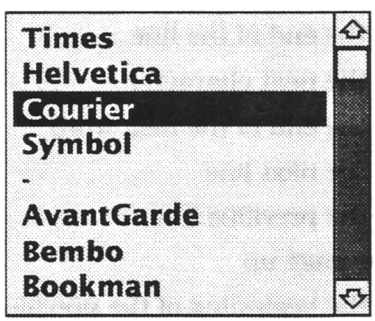
- **Control-L** -- redraw the text
- **Control-T** -- go to the next text field (see also "Controlling the Input Focus" on page 6-13)
- **Esc** -- execute the action of the text

Scrolling Lists

Scrolling lists let you select an item from a list. On the right hand side they have a scrollbar which lets you scroll through the list.

Figure 37

A scrolling list font selector.



You can use the arrow keys on the keyboard (**R8** and **R14**) to move the selection up or down. Another useful way of selecting an item in the list is by typing the first letter of the item. If there are more items which start with the same letter, then type the letter again to select the next one.

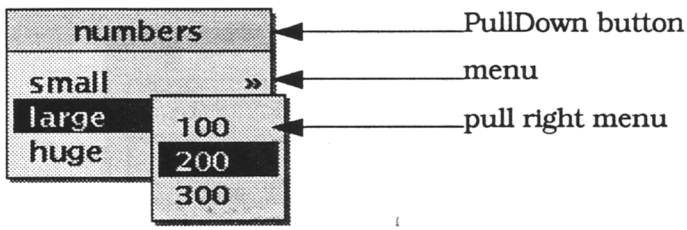
Some scrolling lists can have multiple items selected (nonexclusive). Others can have only one item selected at a time (exclusive). Double clicking on a selected item usually executes the action of the list.

Pulldown Buttons

Pulldown buttons are buttons which have their own pulldown menu. Holding down the mouse in the button shows the menu. You can then drag the mouse over the desired item and release it.

Figure 38

Selecting an item from a PullDown button.



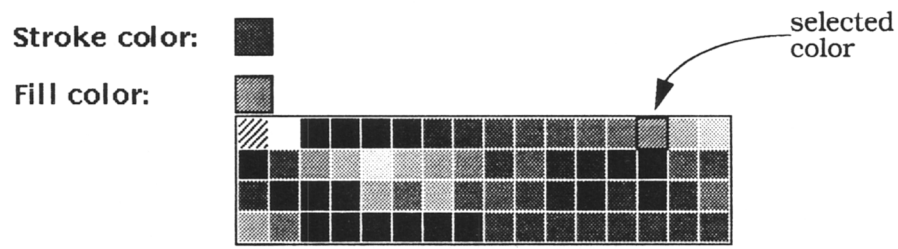
The menu of a pulldown button can also have pull right sub-menus. See "Using Menus" on page 3-5.

Color Selectors

A ColorSelect item lets you select a color from the color pallet. Most property stacks contain ColorSelect items (see Chapter 6, "Object Properties").

Figure 39

Selecting a color using a ColorSelect object.



Holding down the mouse in the ColorSelect button shows pops up a menu containing the same set of colors as the ColorPallet. Drag the mouse to the desired color and release.

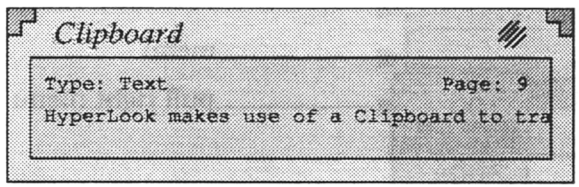
See "The Color Pallet" on page 3-20 to find out how to change the colors on color pallet.

The Clipboard

HyperLook uses a Clipboard to move data between windows or applications. You can show the Clipboard by selecting **Clipboard** from the **Tools** pulldown in the system stack.

Figure 40

The Clipboard stack.



It's possible to *cut* or *copy* some text onto the clipboard, and then *paste* it from the clipboard into another application.

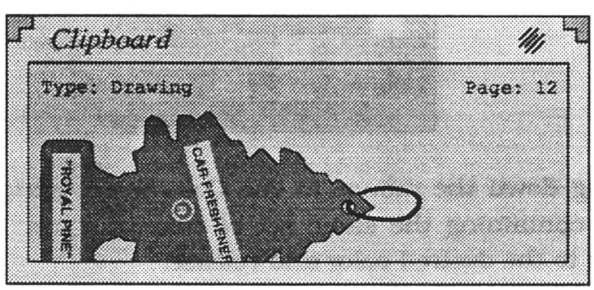
Use the OpenWindows function keys labelled **Cut**, **Copy**, **Paste** for clipboard operations (see "The OpenWindows Function Keys" on page 3-35).

The HyperLook clipboard can hold several types of data. Data types supported by the HyperLook system include text, drawings, HyperLook components, etc.

Artwork, in the form of drawings, can be copied from one graphics editor to another (see Chapter 4, "Graphics Editing"). While the drawing is on the clipboard it will be displayed in the window.

Figure 41

The Clipboard stack containing a drawing.



The Clipboard stack has 16 pages. Each time you copy some data onto the Clipboard it is put onto the next page. When all pages are full, the first page will be used again and any data it previously contained before is overwritten.

You can use the **PgDn** and **PgUp** keys on the keyboard to change pages in the Clipboard. Select the page containing the data you want, the paste it into your application.

This makes it easy to copy and paste multiple objects from one application into another. First copy all the objects from the original application, then paste them from the clipboard into the destination. This avoids unnecessary switching between applications or stacks.

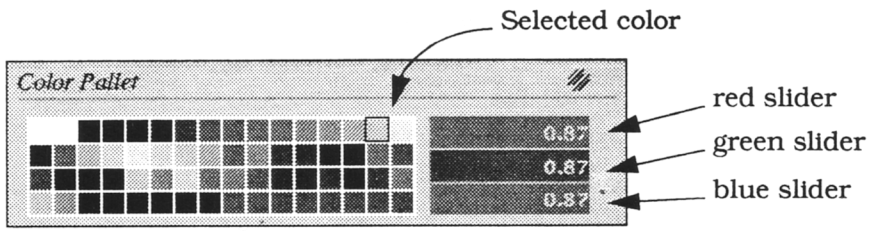


The Color Pallet

HyperLook graphics get their colors from a color pallet which you can change using the ColorPallet stack. You can show the ColorPallet stack by selecting **ColorPallet** from the **Tools** pulldown in the system stack.

Figure 42

The ColorPallet stack.



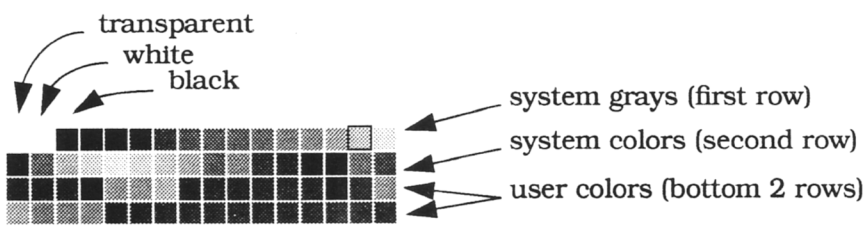
The set of colors that you choose the ColorPallet is used in all the ColorSelect buttons.

On the right the red, green and blue sliders indicate the proportions of red, green and blue in the selected color. The selected color has a little rectangle around it.

The values of the red, green and blue sliders range from 0 to 1 where 0 means the minimum intensity and 1 means maximum intensity.

Figure 43

The pallet of colors.



A bright red color is 1.0 for red and 0.0 for green and blue. A pink color is 1.0 red, 0.7 green and 0.7 blue.

The top two rows of the pallet are system colors. You can modify only the user colors (the bottom two rows). Select a user color and change the color sliders to the desired value.

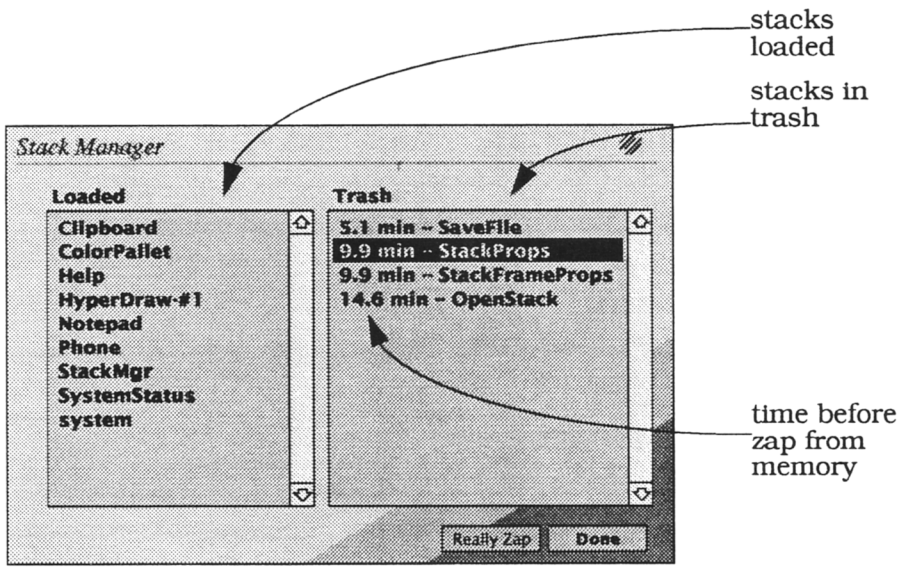
The quality of the colors in the color pallet depends on how you start OpenWindows. The best results are achieved when OpenWindows is started with the **-cubesize large** option (see "HyperLook on Color Screens" on page 2-11).

The Stack Manager

The StackMgr stack is shown by selecting **StackMgr** from the **Tools** pull-down in the system stack. The stack shows two lists, on the left is a list of all the stacks that are currently loaded into the system, on the right is a list of all the stacks in the trash.

Figure 44

The StackMgr stack.



When you zap a stack from the screen it is not immediately deleted from memory. It is kept in memory for a preset time which can vary per stack. When this time expires the stack is really zapped from memory.

Stacks are kept in memory for a while because reloading is then much faster. Some stacks are used frequently and loading them from disk each time would be slow.

The remaining time that a stack has to live is listed in the list on the left. This list is called the *trash* because stacks can be deleted from it without further notice. You may zap a stack from memory before the official time is expired by selecting it in the list and then pressing the **Really Zap** button. You might want to do this to free up some memory for other applications.

The Resource Manager

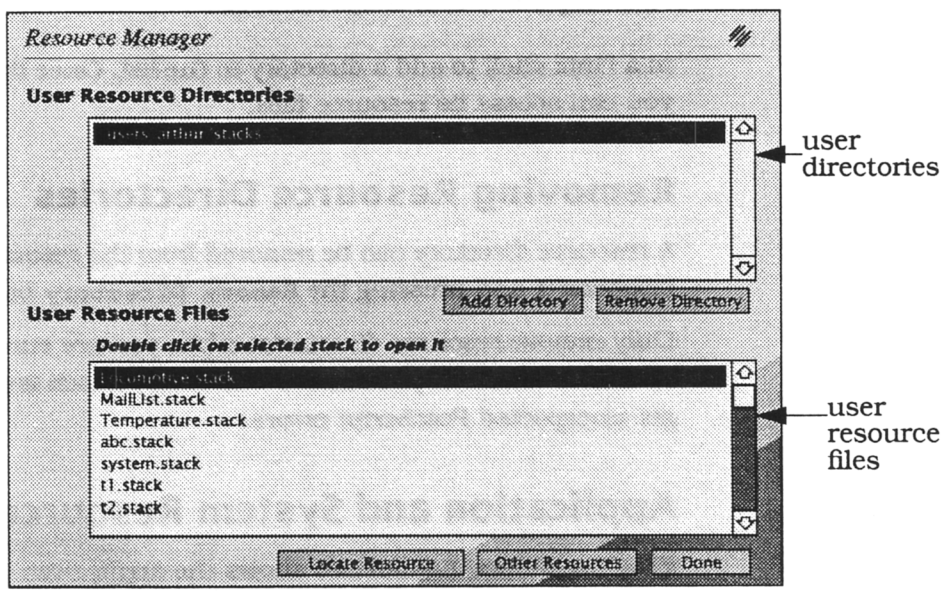
HyperLook keeps a lot of your data in files on disk. The resource manager's job is to locate these files. Files containing data such as stacks, drawings, images, sounds etc. are called *resources*. The resource manager's task is to locate these resource files.

To show the ResourceMgr stack select **ResourceMgr** from the **Tools** menu in the system stack.

The resource manager maintains a list of directories which are searched when a resource is needed. For example the **stacks** directory in your home directory is part of this list. It is used to store your own stacks.

Figure 45

The ResourceMgr stack.



User Resources

The ResourceMgr stack contains two lists. The top one is a list of the user resource directories. The bottom list is a list of resource files in the selected resource directory.

Select the appropriate directory in the top list to examine the files in a resource directory. The resource files are then displayed in the bottom list. You can double click on a resource file to open it.

Adding Resource Directories

To access resources from other directories you must add the directory to the resource directory list. Press the **Add Directory** button and select the appropriate directory using the dialog (see "Selecting Directories" on page 3-30).

You can also use the **hlpath** program to add to the resource list. Type:

```
hlpath /usr/src/video/rsrc
```

in a Unix shell to add a directory to the list. Once the directory is added, you can access its resource files.

Removing Resource Directories

A resource directory can be removed from the resource directories list by selecting it and pressing the **Remove Directory** button.

Only remove resource directories when you are sure that no application or stack needs it. If you remove a directory which is still needed, you may get unexpected PostScript errors.

Application and System Resources

Pressing **Other Resources** shows the application and system resource directories. You cannot change this list, only applications can.

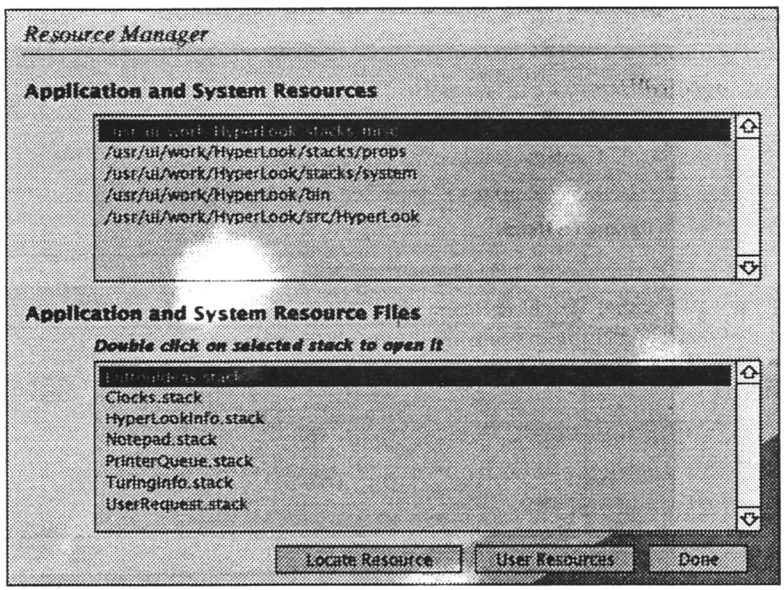
When an application is started, it usually adds its own resource directory to this list.



You can view application resource files or system resource files by selecting the resource directory.

Figure 46

Application and system resources.



To get back to the list of user resources press **`User Resources`**.

Locating a Resource

The HyperLook system locates resources by searching the resource directories in the following order (each list by itself is searched from top to bottom).

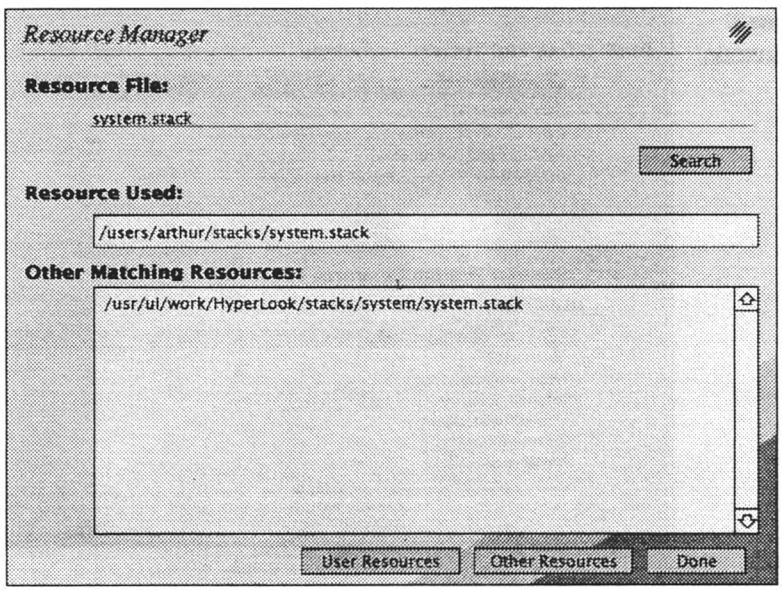
1. user resource directories
2. application resource directories
3. system resource directories

If a resource file occurs in more than one resource directory, the first occurrence is used. This means that user resources can override resources in application or system directories.

Press **Locate Resource** to find out exactly where a resource is located. After pressing the button, enter the name of the resource file and press **Search**. This searches for the resource.

Figure 47

Locating a resource file.



If the resource is found, its file name is shown as well as a list of any other file names matching that resource.

Press **User Resources** to get back to the user resources list.

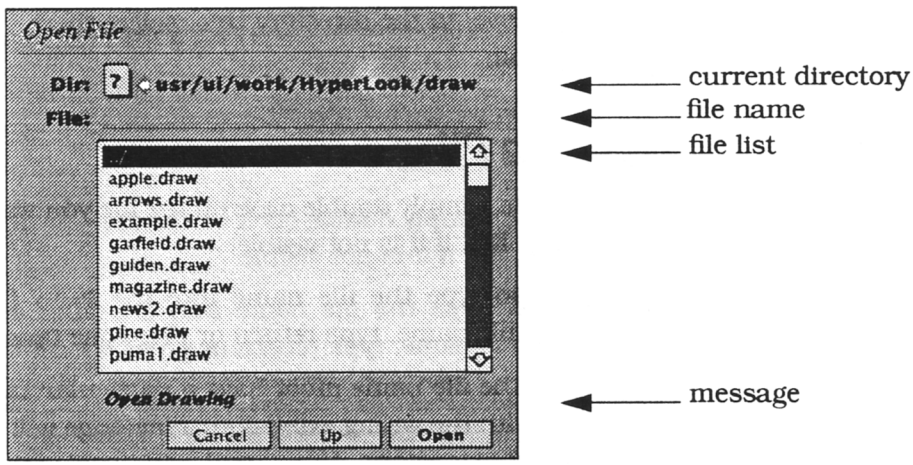
Handling Files

HyperLook provides file dialogs which are used by applications such as the graphics editor to open and save files.

Each dialog is a simple directory browser, which lets you select the appropriate directory, to open or save your file.

Figure 48

The OpenFile dialog.



The HyperDraw graphics editor uses the OpenFile dialog, to open drawing files (see "Loading Drawings" on page 4-6).

When the dialog appears, it displays the message "*reading directory*" while it reads the directory. When it's done reading, the files are displayed alphabetically.

If you want to cancel the whole operation, press the **`Cancel`** button.

Changing Directories

In the file list, directories are displayed ending in "**`/`**".

The easiest way to change to a new directory is to type its name into the **`File`** field, and type return. Either enter a full path name, or a path name relative to the current directory.

Another way to change to a new directory is to double click on the name of the directory in the file name list.The new directory will be opened and the new list of files displayed.



You can also select a recently used directory from a pull down. Just press the left mouse button down over the "**?**" button, and select a directory name from the list.

To go up a level in the directory tree, select the "**../**" directory or press the **Up** button.

Opening Files

To open a file, simply double click on the file you want. Use the scrollbar to scroll the list, if it is not visible.

You can also type the file name into the **File** field. Once you have entered the file name, type return or press the **Open** button.

Sometimes the file name must have a particular extension. If you try to open a file with the wrong extension, a message will be displayed to show you the legal extensions, for example: "*Extension must be .draw*".

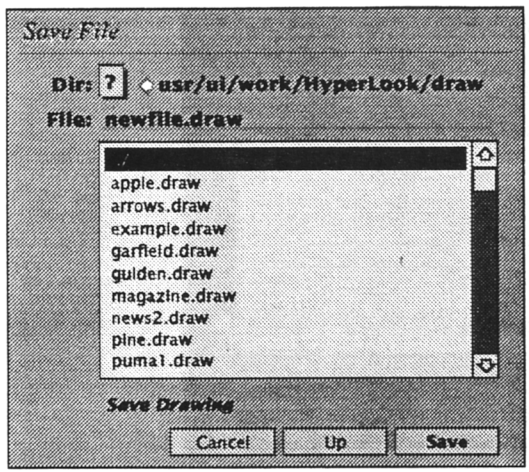
Other error messages are displayed in the message field. For example, a "*cannot read file*" message will be displayed when you don't have read permission for the file.

Saving Files

To save a file the SaveFile dialog is used. This dialog is similar to the OpenFile dialog.

Figure 49

The SaveFile dialog.



The graphics editor uses the SaveFile dialog, to save drawings (see "Saving Drawings" on page 4-6).

Select a directory and enter a file name in the same way as the OpenFile dialog. Once you have entered the correct file name in the file field, type return or press **Save** to save the file.

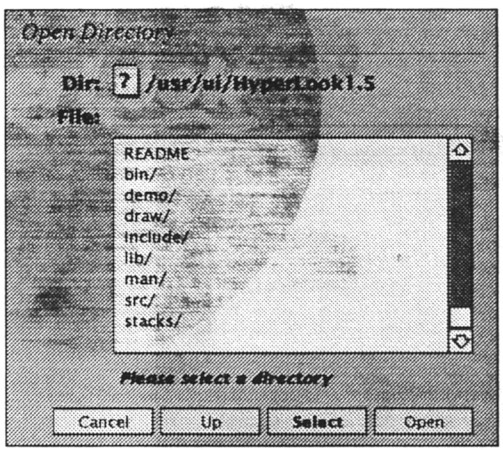
A file can also be selected by double clicking on the file name in the file list.

Selecting Directories

The OpenDir dialog lets you select a directory. It is similar to the OpenFile dialog.

Figure 50

The OpenDir dialog.



You must press the **Select** button to select a directory.

The Resource Manager uses the OpenDir dialog to select resource directories (see "The Resource Manager" on page 3-23).

Short Cuts

The OpenFile, SaveFile and OpenDir dialogs provide some useful short cuts, which can make life a lot easier.

The most useful short cut is file name completion. You enter only the first few character of a file name, then type **Tab**, and the file name will be completed as much as possible.

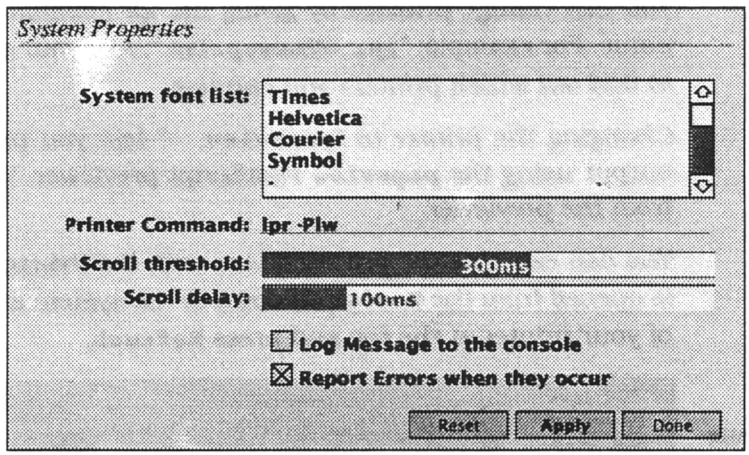
Another useful feature is using "~" in path names. Typing "~" followed by return takes you to your home directory. You can also type **~/subdir**, to refer to a subdirectory of your home directory. You can easily access directories of other users, by typing **~username**.

The System Properties

Selecting **SystemProps** from the **Tools** pulldown in the system stack shows the system properties stack. This stack lets you change system properties.

Figure 51

The System Properties stack.



Once you have changed the properties press **Apply**. To hide the window press **Done**. Changes can be undone by pressing **Reset**.

When you press **Apply**, the values which you have entered are saved to a file called **.hyperlook** in your stacks directory. This is a text file which contains some frequently used variables which need saving between sessions.

Changing the System Fonts

The system font list lets you specify which fonts you want to use in menus, pulldowns and selectors. You can change the default list and add fonts that you use most frequently.

When you press **Apply**, undefined fonts are removed from the list. A single "–" character separates groups of fonts.

Specifying a Printer

The default printer command used by tools such as the graphics editor is "**lpr -Plw**". The lpr program prints a file to the printer named by the **-P** option (see the UNIX documentation for the lpr command)[1].

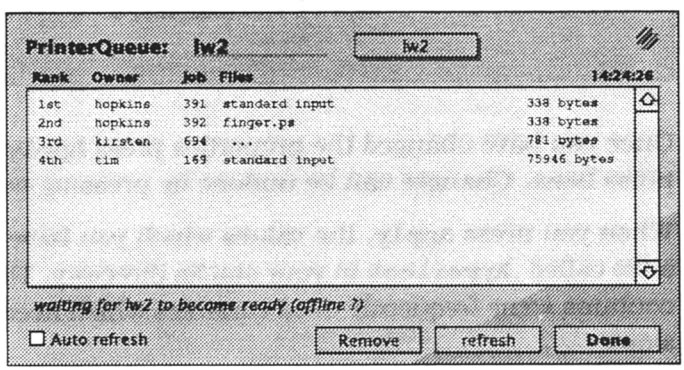
You can change printers by giving the lpr command a different printer name. For example: "**lpr -Pnewsprint**". Ask your system administrator to find out which printers you can use.

Changing the printer to "**pageview -**" lets you preview the generated output using the **pageview** PostScript previewer. You can print the file from the previewer.

You can examine the printer queue with the **PrinterQueue** stack which is opened from the **Tools** pulldown in the system stack. Enter the name of your printer at the top and press **Refresh**.

Figure 52

The PrinterQueue stack.



Scrolling Parameters

The System properties stack lets you select the scrolling threshold and scrolling delay times. These affect the speed at which text, scrolling lists and fields scroll.

The **Scroll threshold** is the initial delay in milliseconds between pressing the mouse down and the moment scrolling starts.

1. It must be a PostScript printer.

The **Scroll delay** is the time between the scrolling of each line of text. This affects the scrolling speed.

Logging Messages

HyperLook uses messages to pass information around in the system. You can switch logging on to monitor these messages. See Chapter 7, "Scripting" for more information on message passing. If you don't intend to do any scripting, you don't need to worry about logging messages.

Message logging means that messages are printed to the console as they occur. This lets you analyze the message traffic of your application. Here are examples of some messages:

```
from [?] to Stack(system) --> StackOnLeft()
from Stack(system) to Button(#2) --> OnMouse()
from Button(#2) to Button(#2) --> Action(0)
from Button(#2) to Stack(OpenStack) --> InitStack()
```

Before switching logging on, make sure that you have opened a console window. If you haven't, messages will be printed all over the screen.

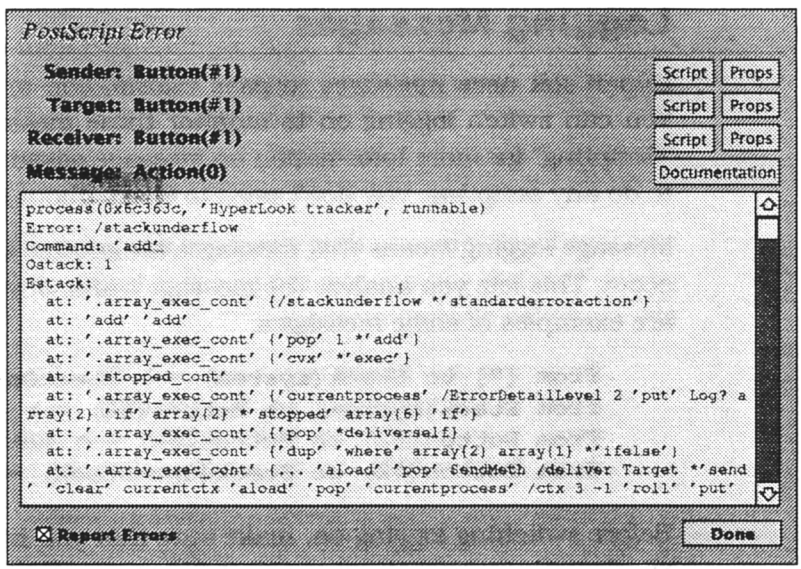
Report Errors

If you are writing your own scripts, you may find that errors are displayed when they occur. When an error occurs the error message is displayed in the **PostScriptError** stack.

It is sometimes useful to disable this feature, especially when an error occurs frequently, or when your error causes more errors. Click on the check box in the system properties stack to turn error reporting off.

Figure 53

The PostScriptError stack.



You can also switch error reporting off from the PostScriptError stack. Use the checkbox in the lower left corner. See Chapter 7, "Scripting" for more information on the PostScriptError stack.

The OpenWindows Function Keys

Sun keyboards have some labelled function keys which have a function defined by OpenWindows. The HyperLook functionality for these function keys is the same as, or similar to the OpenWindows functionality:

Again
Redraw the stack (see "Redrawing Stacks" on page 3-12).

Front
Bring a stack to the front (see "Bringing Stacks to the Front" on page 3-10), or send it to the back if it's already in front.

Open
Iconify a stack (see "Iconifying Stacks" on page 3-11), or open it if it is already an icon.

Undo
Undo the last operation in the graphics editor (see "Using the Clipboard" on page 4-22).

Copy, Paste, Cut
Clipboard editing operations (see "The Clipboard" on page 3-18).

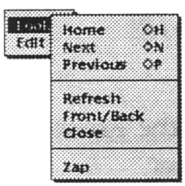
Help
Show help (see "Using Help" on page 3-3).

Home, End, PgUp, PgDn
Move to the first, last, previous or next card of the stack (see "Moving Between Cards of a Stack" on page 3-8).



The Stack Menu

All of the operations that are available on the stack menu are described in other parts of the manual. Here is a complete list of references to the relevant sections.



Look

Home

Go to the home card. See "The Home Card" on page 3-7.

Next
Previous

Go to the next or previous card. See "Moving Between Cards of a Stack" on page 3-8.

Refresh

Redraw the stack. See "Redrawing Stacks" on page 3-12.

Front/Back

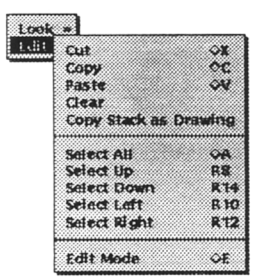
Send the stack to the front or to the back. See "Bringing Stacks to the Front" on page 3-10.

Close

Iconify the stack. See "Iconifying Stacks" on page 3-11

Zap

Zap the stack from the screen. See "Zapping Stacks" on page 3-12.



Edit

Cut
Copy
Paste
Clear

Clipboard editing operations. See "The System Properties" on page 3-31.

Copy Stack as Drawing

Copy a stack to the clipboard as a drawing. See "Copy as Drawing" on page 5-23.

Select All
Select Up
Select Down
Select Left
Select Right

Move the cursor. See "Editing Text" on page 3-14.

Edit Mode

Switch to edit mode. See Chapter 5, "Editing Stacks". This changes the stack menus to include editing features (see "Edit Mode Menus" on page 5-36). To get back to browse mode, select **Browse Mode** from the **Edit** menu,

4

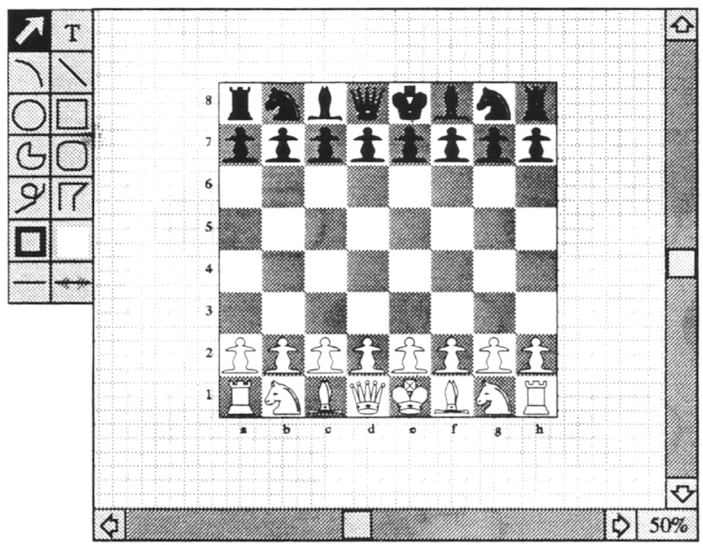
Graphics Editing

HyperLook provides a powerful graphics editor which is fully integrated into the HyperLook system. The HyperLook graphics editor is useful because it's just an object like a button or a slider. This makes it very easy to add graphics editing capabilities to your own applications.

The editor is implemented using a graphics editor object called a **`DrawTool`**. The functions described in this section therefore apply to all the **`DrawTool`** objects in the system, even those in your own applications.

Figure 54

A PostScript Graphics editor object.



The editor is an *Object Oriented PostScript Graphics Editor*. This means that it edits objects such as rectangles, ovals, lines, and images. This type of editor is particularly useful for creating diagrams and technical drawings.

The fact that the editor uses PostScript graphics means that drawings produced with this tool can be scaled and rotated, and produce very high quality output when printed on a PostScript device.

PostScript is a high level graphical programming language. It is also a standardized page description language. It can produce high quality output on a large range of devices.

Figure 55

A drawing created using the graphics editor.



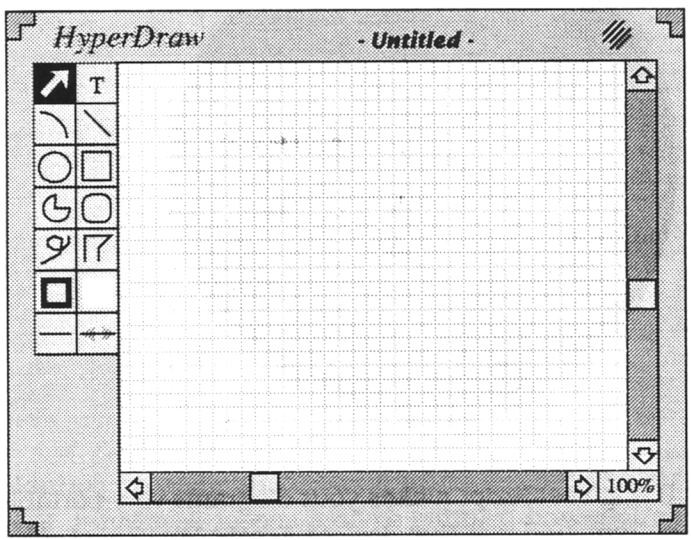
Using PostScript makes your diagrams very portable. Lots of other applications and printing devices accept PostScript. Most of the diagrams in this manual were produced with the graphics editor and then imported into FrameMaker.

Getting Started

The editor is started by pressing the **Draw** button on the system stack. This will show a HyperDraw stack. Pressing the button again creates another instance of the stack.

Figure 56

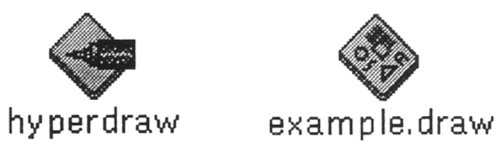
The graphics editor stack.



When working with more than one drawing, it is sometimes useful to have several editors active. Drawings or parts of drawings can be copied and pasted between editors.

Figure 57

The **hyperdraw** program icon and the drawing file icon.



Another way to start the editor is double clicking the **hyperdraw** program icon in the Sun desktop file manager (see "Using the File Manager" on page 2-9 to see how to install the icons in the file manager). If you want to edit a drawing file, double click the drawing file icon.

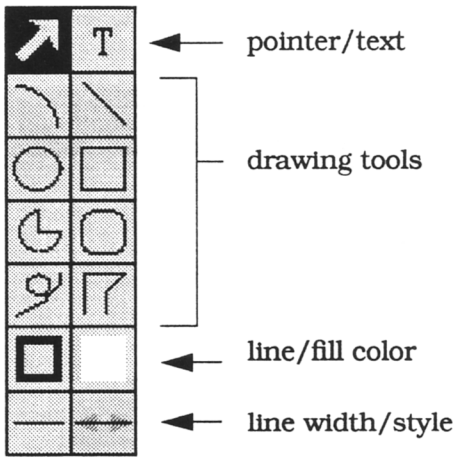
You can get rid of the editor when you are finished by selecting **Zap** from the stack menu (see "Zapping Stacks" on page 3-12).

The Tool Pallet

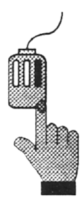
Figure 56 on page 4-4 shows the graphics editor stack. On the left hand side it has a tool pallet which lets you choose drawing tool, colors, fill and line styles. The bottom right corner displays the current magnification.

Figure 58

The tool pallet.



The scrollbars let you move your drawing around in the central portion of the stack, which is called the paper. The stack can be resized using the resize corners ("Resizing Stacks" on page 3-11).



Holding down the right mouse button on the paper shows a popup menu. It has a number of submenus which give you access to the features of the tool. In the following sections you will read instructions like "select **Open** from the **File** menu". This means "select the **Open** option from the **File** sub menu of the popup menu".

If you want an at-a-glance guide to the graphics editor menu functions you should read "The Graphics Editor Menus" on page 4-34. Otherwise read on and you will be taken on a tour of the graphics editor.

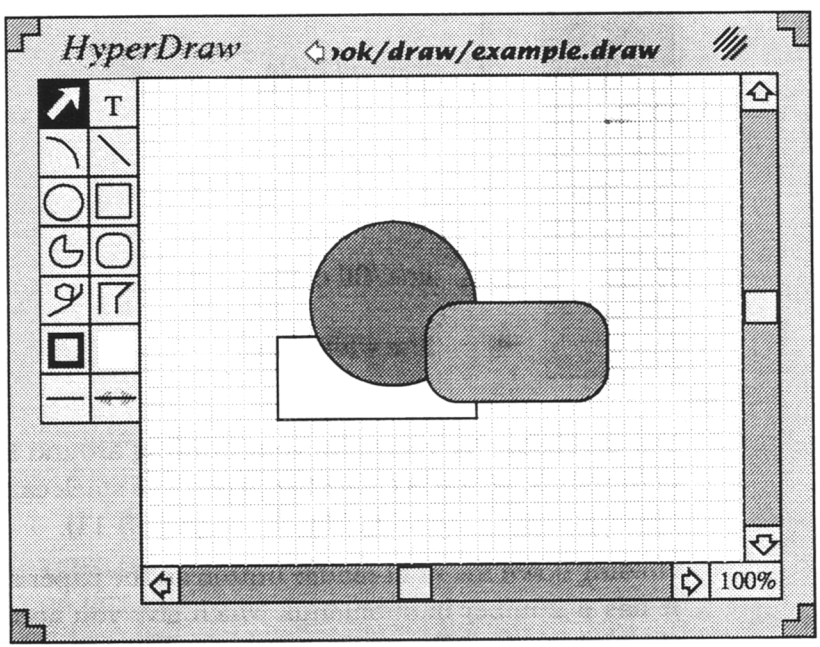


Loading Drawings

Select **Open** from the **File** menu to open a drawing. In the example in Figure 59, the drawing **draw/example.draw** was loaded from the HyperLook directory. See "Opening Files" on page 3-28 on how to use the **OpenFile** dialog. Note that drawing files always have a "**.draw**" file name extension.

Figure 59

After loading **example.draw**.



Saving Drawings

To save a drawing select **Save As...** from the **File** menu. You can enter a file name using the SaveFile dialog (See "Saving Files" on page 3-29).

Drawings can also be saved in the file from which they were originally loaded. Use **Save** from the **File** menu to save the drawing in the file from which it was loaded.

You can also save the drawing as *Encapsulated PostScript*. Select **Save EPS..** from the **File** menu to save the drawing in this format. See also "Encapsulated PostScript Objects" on page 4-30.

Saving a drawing as EPS is useful if you want to import the drawing into another application. Many word processing applications can import *Encapsulated PostScript graphics into their documents.*

Printing Drawings

A drawing is printed by selecting **Print** from the **File** menu. A popup message will appear to show that the drawing is queued for printing. See "Specifying a Printer" on page 3-32 for how to specify the printer on which the drawing is printed.

You can also use the **drawps** program to prepare a drawing for printing on a PostScript printer. It converts a drawing into PostScript format. The program can also be used to generated Encapsulated PostScript.

To print a drawing from a shell use[1]:

```
drawps < example.draw | lpr
```

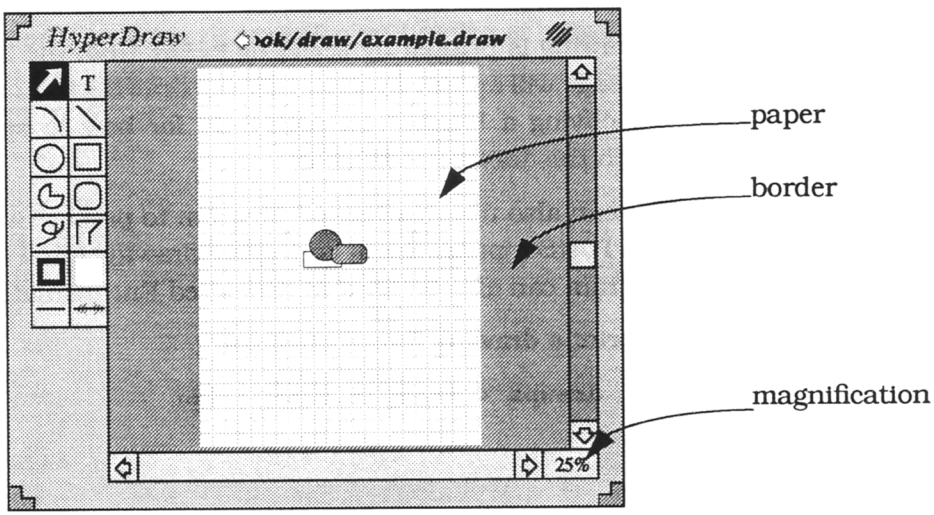
1. Use the **-P** option of the **lpr** command to specify a printer if necessary.

Zooming

After loading a drawing you can view it at different magnifications. To zoom out (make the drawing appear smaller) select **Zoom Out** from the **View** menu.

Figure 60

Zooming out will show the edges of the paper.



The magnification is displayed in the bottom right hand corner.

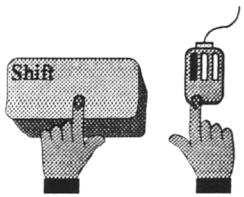
As you zoom out a few times, you will start to see the edges of the paper. The paper edges indicate how the drawing fits on the paper when it is printed.

You can zoom back in using the **Zoom In** option of the **View** menu. The magnification can be varied from 12.5% to 800%.

At any time it is possible to zoom back to the default (100%) magnification by selecting **Zoom Normal** from the **View** menu.

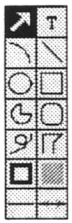
Moving Around

The position of the view can be changed using the scrollbars to the right and below the paper. See "Using Scrollbars" on page 3-14 if you're not sure how to use the scrollbars.



A very useful way to change the position of the view is by holding the **Shift** key down while dragging the left mouse button. This lets you drag the view around.

Selecting Objects



Most drawing operations apply to the currently selected objects. Objects can be selected by moving the mouse over them and clicking the left mouse button. The fact that an object is selected is shown by its marker points, usually at the four corners.

When selecting objects, make sure that the pointer is selected in the tool pallet.

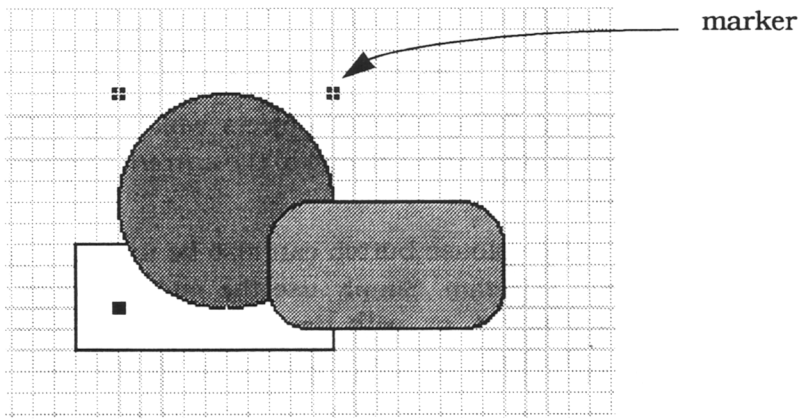


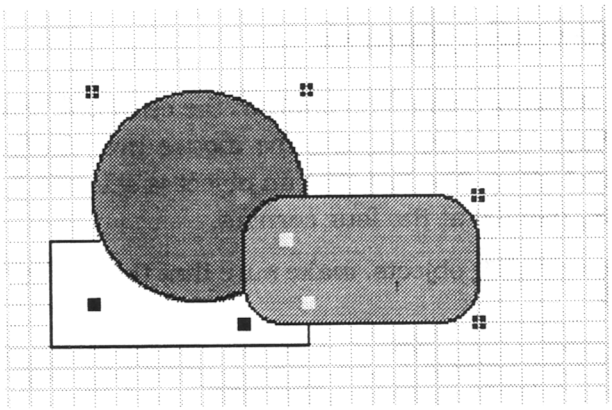
Figure 61

The circle is selected.

You can also select more than one object at the same time. Hold the mouse down somewhere outside the desired objects and drag a rectangle around them. The objects inside the rectangle are selected

Figure 62

The circle and the rounded corner rectangle are selected.



To extend the number of objects which are selected, use the middle mouse button. This will add to the current selection, without deselecting objects.

The middle mouse button can also be used to remove objects from the current selection. Simply use the middle mouse button to click on a selected object, or, with the middle button, drag a box around some selected objects to remove them from the selection.

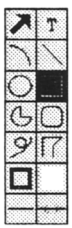
You have two options when you want to select an object which is behind another object. Either enclose the obscured object by dragging a box around it, or click more than once on the object. Each click will select the object behind the currently selected one.

To select all objects in the drawing choose **Select All** from the **Edit** menu.

Editing Objects

Now you know how to view objects and how to select objects, we will move on to editing objects. Most objects are created and edited in the same way. In order to illustrate the creation and editing of objects we will consider the editing of a rectangle in detail. The methods illustrated can be applied to other object types in the same way.

Creating Objects



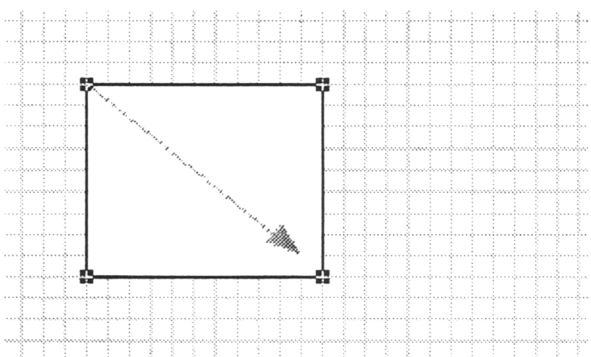
Let's start by drawing a new rectangle. First clear the view by selecting all the objects and either pressing the **Delete** key or selecting **Clear** from the **Edit** menu.

The rectangle tool is displayed in the panel on the left. When you click on it will be highlighted, just like in the picture, to indicate that the next thing that you draw will be a rectangle.

To create a rectangle, simply hold down the mouse at the desired location on the paper, and drag until the rectangle is the correct size. When you release the mouse button, the rectangle will be added.

Figure 63

Adding a rectangle.



After the rectangle is added the tool is switched back to the pointer automatically.

Creating Multiple Objects



If you double click in the tool pallet, the selected tool will be displayed in gray instead of black. This means that the tool will not switch back to the pointer when the object is created. Instead it will stay active until you explicitly click on the pointer. This is useful when you want to add several objects in a sequence.

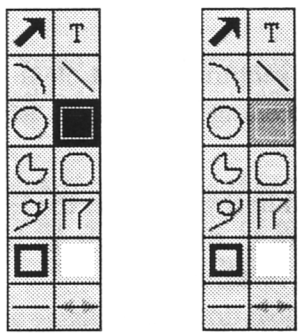


Figure 64

Creating a single rectangle (left) and creating multiple rectangles (right).

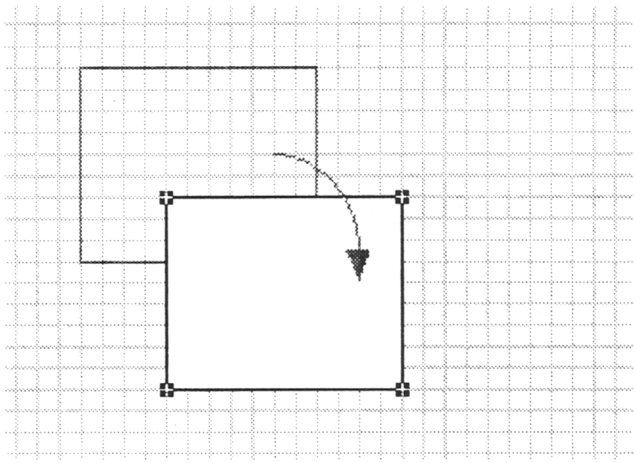
Polygons have a special behavior when the multiple object feature is enabled. See "Polygon and Spline Objects" on page 4-26 for more information.

Moving Objects

The rectangle is moved by holding down the mouse in the area of the object when it is selected, and dragging the object to the desired location.

Figure 65

Moving an object to a new location.



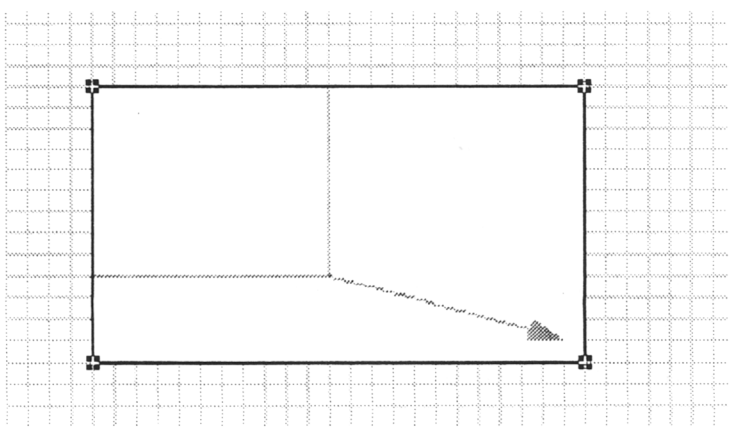
When more than one object is selected, all are moved accordingly.

Changing the Size of Objects

Objects can be edited using their markers. You can change the size of an object by holding the mouse down in a marker, and dragging it until the object has the required size.

Figure 66

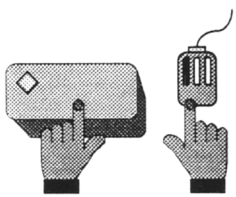
Changing the size of an object.



The width and height of the object (in pixels) is displayed near the cursor when you are resizing it.

Another way to scale objects is to select **`Scale 50%`** or **`Scale 200%`** from the **`Special`** menu. This is an easy way to make the selected objects twice or half their size. See also "Constrained Editing" on page 4-15 on how to change the size of an object without changing its width/height ratio.

Rotating Objects

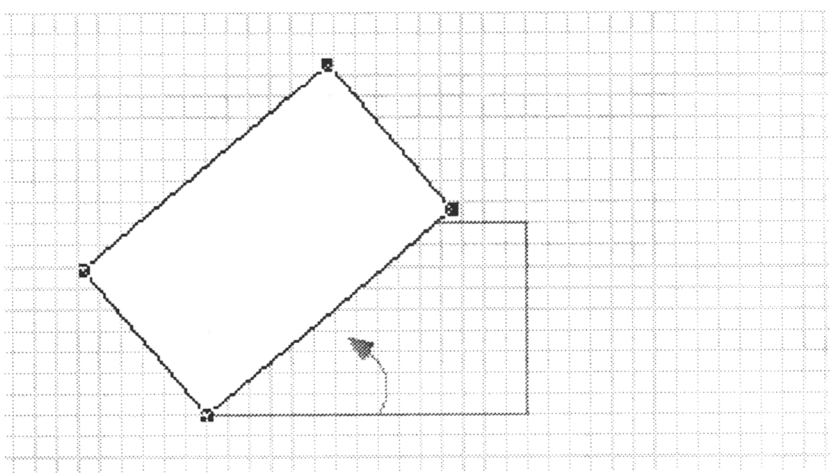


To rotate an object (or a group of objects), hold the mouse button down in a marker point while holding down the **Meta** key.

While the mouse is down you are able to rotate the object around the selected point. When the button is released, the object is displayed in its new orientation.

Figure 67

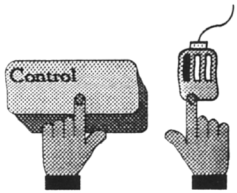
Changing the size of an object.



The degrees of rotation are displayed near the cursor while you are rotation the object.

To undo the rotation, select **`Flip to normal`** from the **`Special`** menu.

Constrained Editing



While editing objects, it is possible to apply a constraint. Hold down the **Control** key while dragging an object. It will only move vertically or horizontally.

Similarly, holding down the **Control** key while stretching the corner of an object will restrict its shape to a square or a circle, or it will maintain the image's original width/height ratio.

Rotating an object while holding down the **Control** key will restrict the angle of rotation to multiples of 45 degrees.

Using the Grid

You will notice that all editing operations are constrained to a grid. The size of the grid can be changed from the **Grid** menu.

The default grid size is 8, but smaller and larger sizes can be selected. Grid alignment can be switched off by selecting **None** from the **Grid** menu.

The grid lines are usually displayed in the background of a drawing. They are spaced according to the grid size and the current magnification. If the grid is small, not all grid lines are displayed.

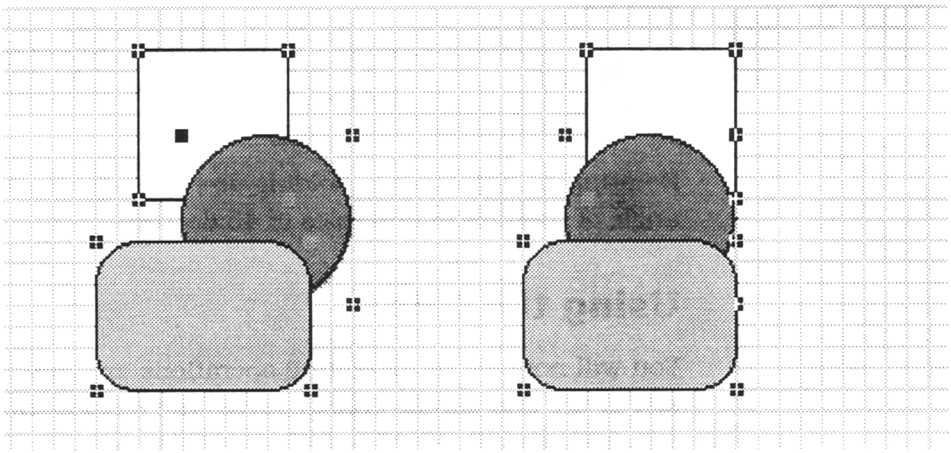
Drawings created using a particular grid size should be edited using that same size or a smaller one. Otherwise, if the grid size is too large, it will be hard to edit the drawing without realigning objects to the new grid.

Aligning Objects

It is sometimes useful to align objects so that certain edges are on the same line.

Figure 68

Aligning some objects to their *right* edge.



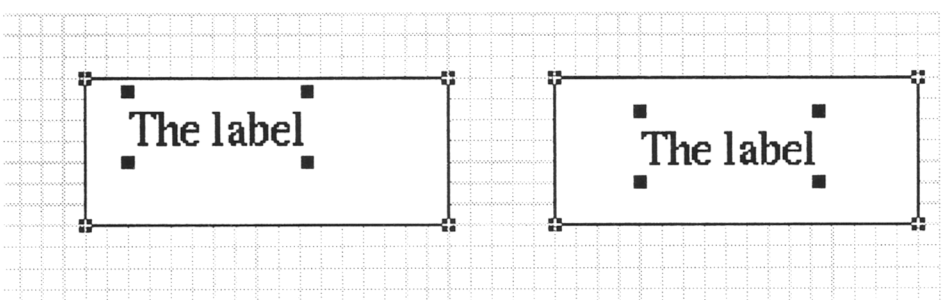
The **Align** menu provides several options. The *top*, *left*, *right*, or *bottom* edge can be aligned. Also, objects can be aligned by their center point, either at one point, along a vertical or horizontal line, or to the grid.

Alignment is always relative to the back most object. This object is not moved, the other objects are aligned to it.

Aligning the center of objects is useful when you want to put a text string in the center of a bounding box.

Figure 69

Centering a text string in a box.

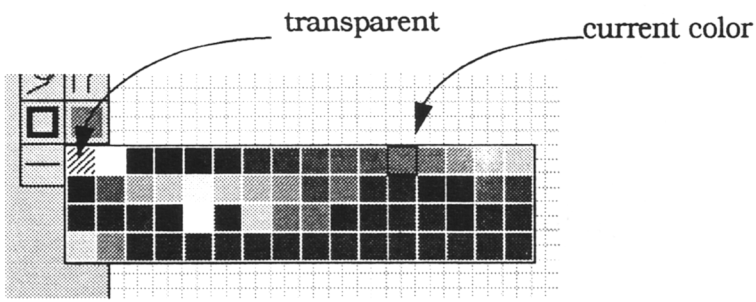


Selecting the Fill and Line Color

Objects are created using the current colors specified in the tool pallet. You can change the fill color of the selected objects by selecting a new color.

Figure 70

Selecting a fill color.



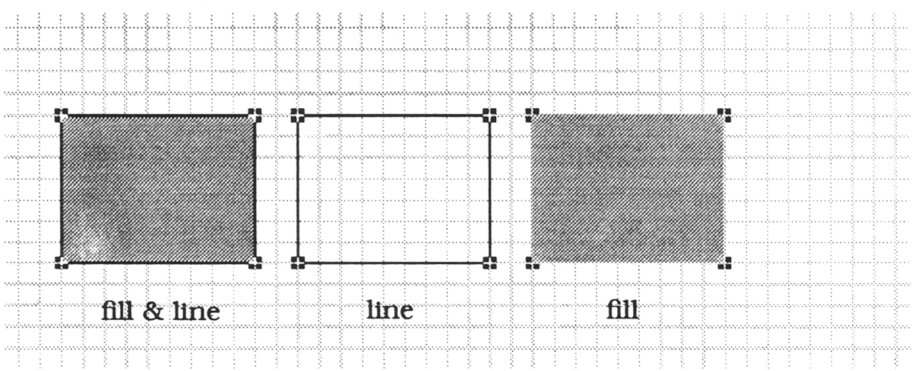
When you hold down the mouse on the fill color tool (on the right), a pallet of colors is shown. You can point the mouse at the appropriate color. When you release the mouse over a color, it will be applied to the selected objects. See also "Color Selectors" on page 3-17.

The first color in the pallet (in the top left corner) is transparent. If this color is chosen, the object is not filled (objects behind it will be visible).

The line color is chosen in the same way, using the line color tool (on the left). A transparent line color means no line at all[1].

Figure 71

Changing the color of a rectangle.



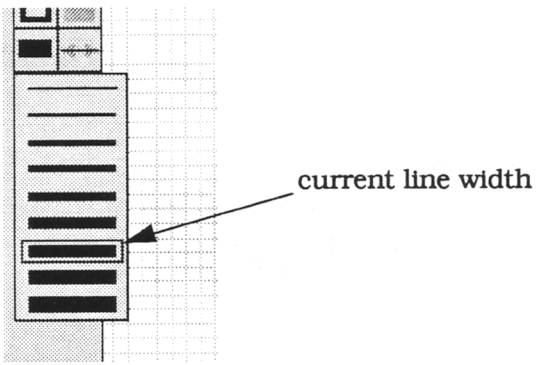
1. It is possible to give an object a transparent fill color and a transparent line color. It will be invisible!

Selecting the Line Width

When you hold down the mouse in the line width tool, a panel pops up, from which you can select a new line width.

Figure 72

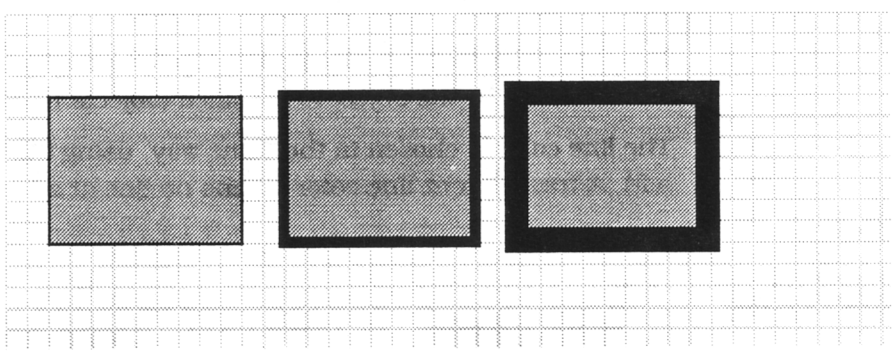
Selecting the line width.



This applies to the selected objects and all new objects that you create. The figure below shows rectangles with different line widths.

Figure 73

Changing the line width of the rectangle.

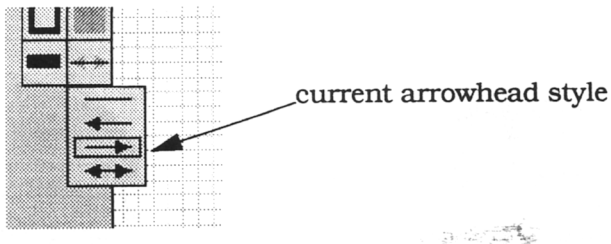


Selecting Arrowheads

Lines and arcs can have arrowheads. To try this out, you must first create a line. Then select an arrowhead style with the arrowhead tool.

Figure 74

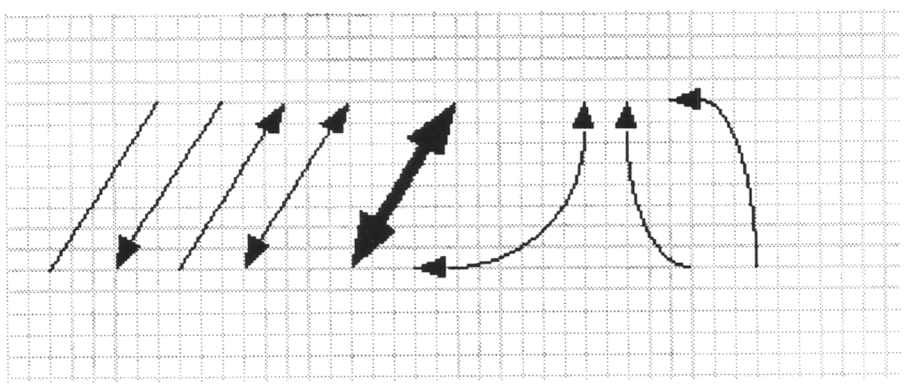
Selecting arrow heads.



The arrowhead style applies to the selected lines and curves. Only arcs and lines can have arrowheads. The size of the arrowhead depends on the line width of the line or arc.

Figure 75

Some lines and arcs with arrow heads.

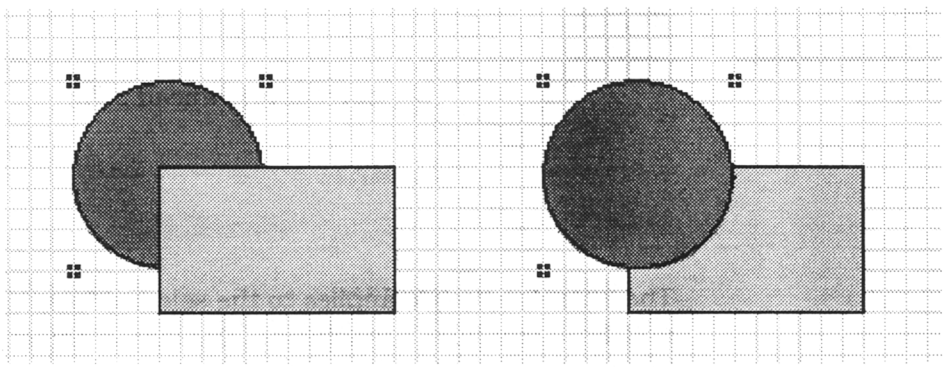


Front to Back Order

As you create objects, you will notice that they can overlap. If you want to bring an object to the front, select **Front** from the **Arrange** menu.

Figure 76

Bringing an object to the front.



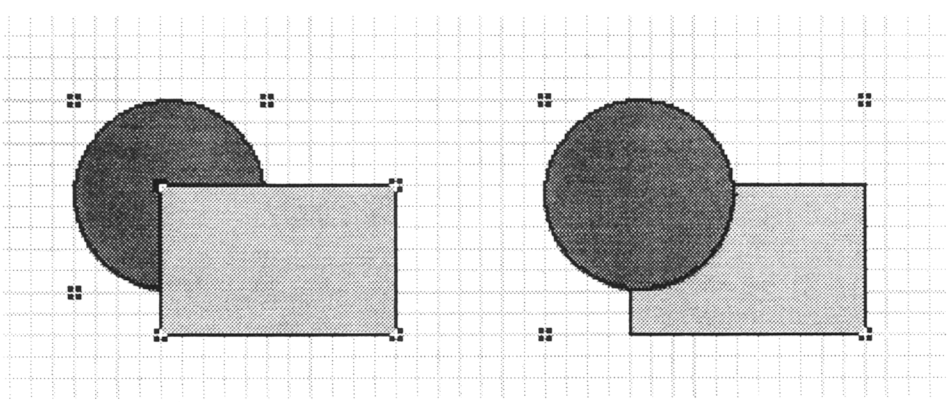
Alternatively you can select **Back** from the **Arrange** menu to send the selected object to the back.

Grouping

It is sometimes useful to treat a set of objects as a single object. This is achieved by grouping. Select several objects, and choose **Group** from the **Arrange** menu. You will see that the markers for the individual objects are replaced by markers for the entire group. The bounding box of the group is just large enough to enclose all the grouped objects.

Figure 77

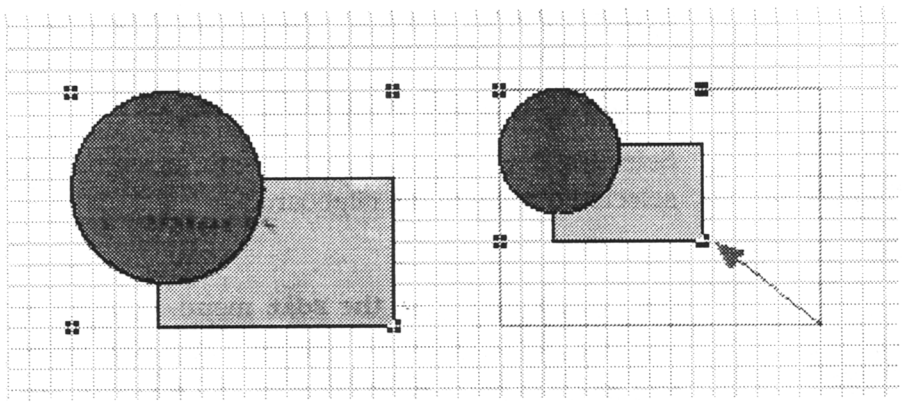
Grouping two objects together.



Once objects are grouped together, they are treated as if they were a single object. You can move, resize and scale a group just as you do with an individual object.

Figure 78

Changing the size of a group.

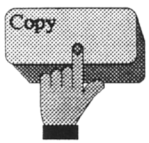


The grouping of objects can be undone by selecting **Ungroup** from the **Arrange** menu.

Using the Clipboard

Before reading this section make sure you are familiar with the use of the HyperLook clipboard (see "The System Properties" on page 3-31).

Copying and Pasting Objects



Selecting **Copy** from the **Edit** menu, or using the **Copy** key, copies the selected objects to the clipboard.



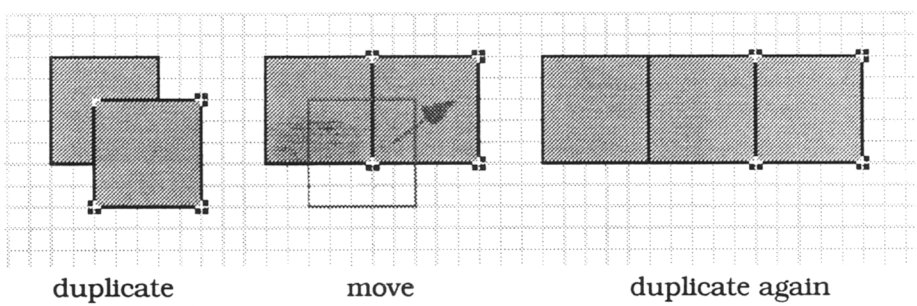
Selecting **Paste** from the **Edit** menu, or using the **Paste** key, pastes drawing objects from the clipboard into your drawing. You can use copy and paste to move graphics from one graphics editor to another.

Duplicating Objects

To duplicate the selected objects use **Duplicate** from the **Edit** menu. An exact copy of the selected object is added to the drawing a little below and to the right of the original.

Figure 79

Duplicating an object.

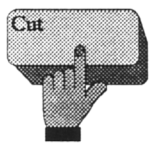


If you duplicate an object and then move the new object, HyperLook remembers the offset. If you duplicate again the new (third) object has an offset equal to that between the original and the first duplicate.



Deleting Objects

To delete objects from a drawing, select them and hit the **Delete** key. You can also select **Clear** from the **Edit** menu.



Select **Cut** from the **Edit** menu, or use the **Cut** key, to cut the selection from the drawing to the clipboard.

Undo



Most editing operation can be undone. If you have made a mistake while editing a drawing, simply select **Undo** from the **Edit** menu, or use the **Undo** key, to undo the last change.

Special Objects

So far, you have learned how to edit objects such as rectangles and lines. Some of the other objects have special features, which are discussed in this section.



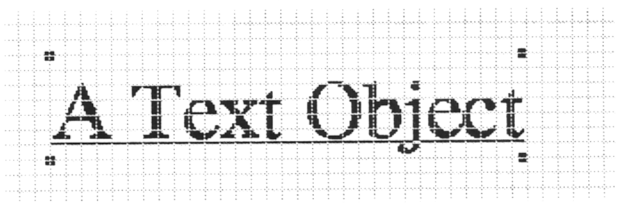
Text Objects

To add a text object, select the text tool from the tool pallet, and click on the location where you want the text. An empty text object will be added. To enter the text, just start typing.

While the object is selected and you're typing text, it will be displayed in *edit* mode.

Figure 80

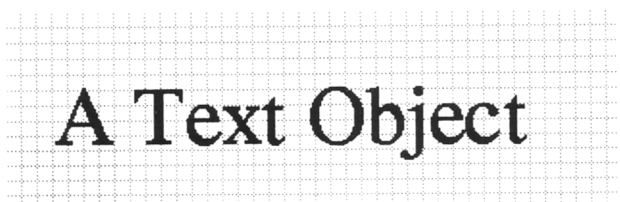
Creating a text object.The object is in *edit* mode.



The text will look rather rough while in *edit* mode. As soon as you deselect it, it will be redisplayed in its normal state.

Figure 81

The text after reselecting it. It is no longer in *edit* mode.



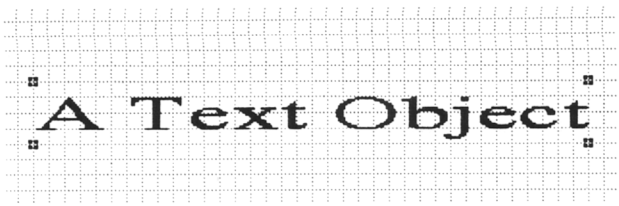
Text objects can have multiple lines of text separated by typing return. If you want to edit an existing text object just select it and start typing.

The **Text** menu has four submenus. The **Font** menu changes the font, the **Style** menu changes the font style, the **Size** menu changes the font size, and the **Align** menu changes the alignment of multi-line text objects.

The size and scaling of the font is adjusted when the object's size is changed. You can resize a text object just like any other object.

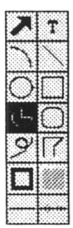
Figure 82

The text after stretching it using the marker points.



You can rotate text objects. All of the operations that you can apply to other objects work with text objects too.

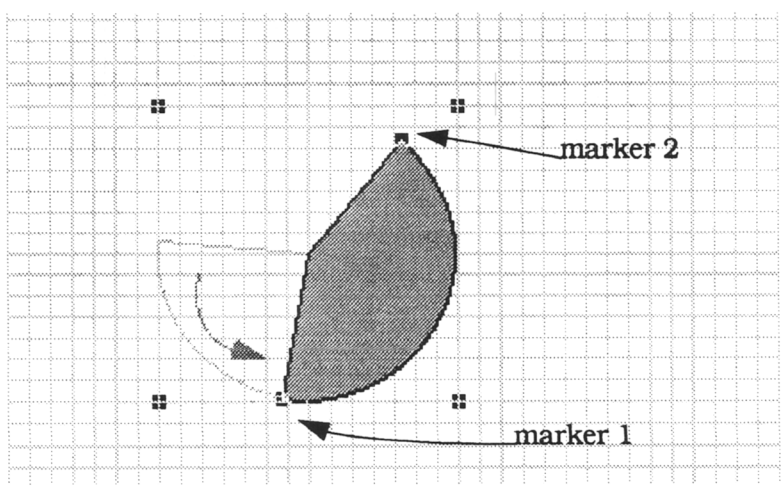
Pie Objects



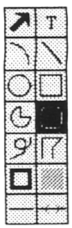
Pie objects let you create sliced pie shaped wedges. A pie object, when selected, shows 2 extra marker points. You can drag them to change the angular size of the pie slice. The exact angle of the slice is displayed near the cursor while you are changing it.

Figure 83

Editing a pie object.



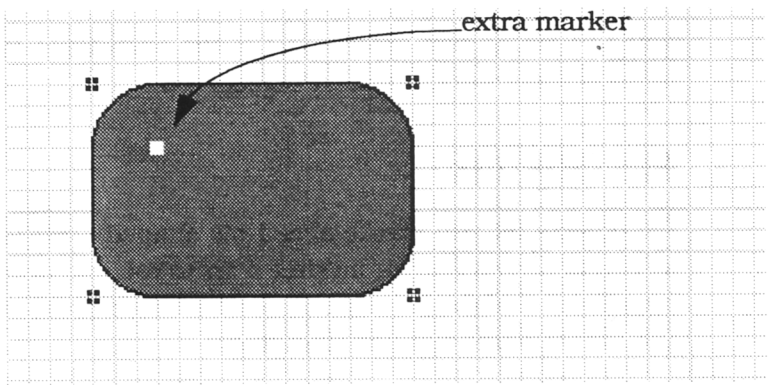
Rounded Corner Rectangles



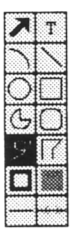
A rounded corner rectangle also has an extra marker. It lets you change the roundness of the corners. The rounding is restricted to the minimum of the width and the height of the rectangle. The size of the rectangle's corners is displayed near the cursor while you are changing it.

Figure 84

A rounded corner rectangle with the extra marker point.



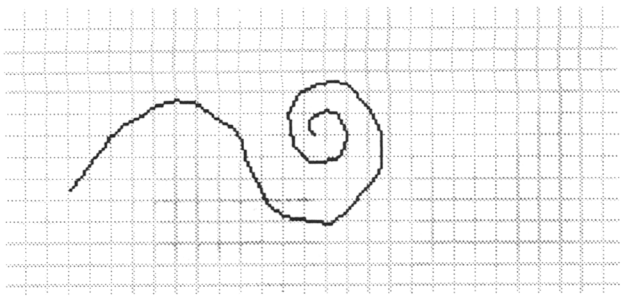
Polygon and Spline Objects



The easiest way to draw a complex curve is with the freehand tool. When it is active, dragging the mouse creates a polygonal path following the mouse. The freehand tool is ideal for entering paths with smooth curves and strange shapes.

Figure 85

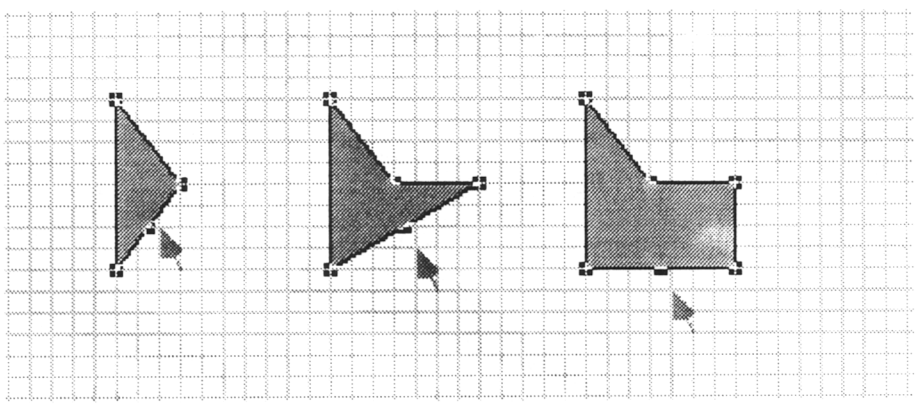
A polygon created with the freehand tool.

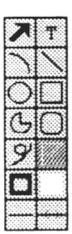


You can also enter a polygon point by point. This is like editing line objects. A polygon has a marker point for every point you enter, plus one extra marker point. This extra point extends the polygon. Dragging the extra marker adds a new point to the polygon.

Figure 86

Editing a polygon using the extra marker point.





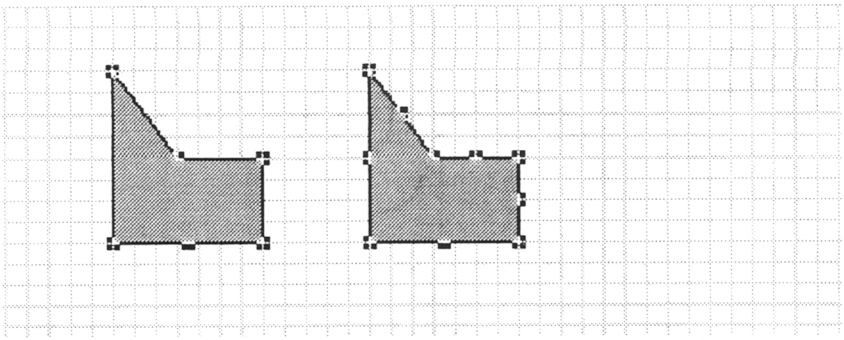
If you double click the polygon tool (see "Creating Multiple Objects" on page 4-12), the next mouse click will add a new point to the polygon, instead of creating a new polygon. This is useful if you want to add a lot of points.

If you want to add an extra point to every edge of the polygon, then use **Double Poly Points** from the **Special** menu. After adding more and more points you can use **Reduce Poly Points** from the **Special** menu to reduce the points to the minimum necessary.



Figure 87

Doubling the number of polygon points.



Once you have created a polygon, you can change it to a spline (a smooth curve). Select **`Flip Poly and Spline`** from the **`Special`** menu. Splines can be turned back into ordinary polygons using the same menu. Splines are edited in the same way as polygons.

Figure 88

A polygon and its spline.

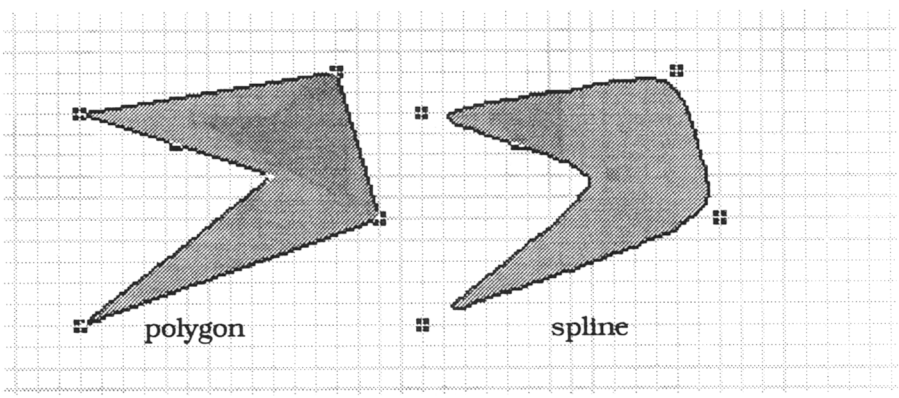
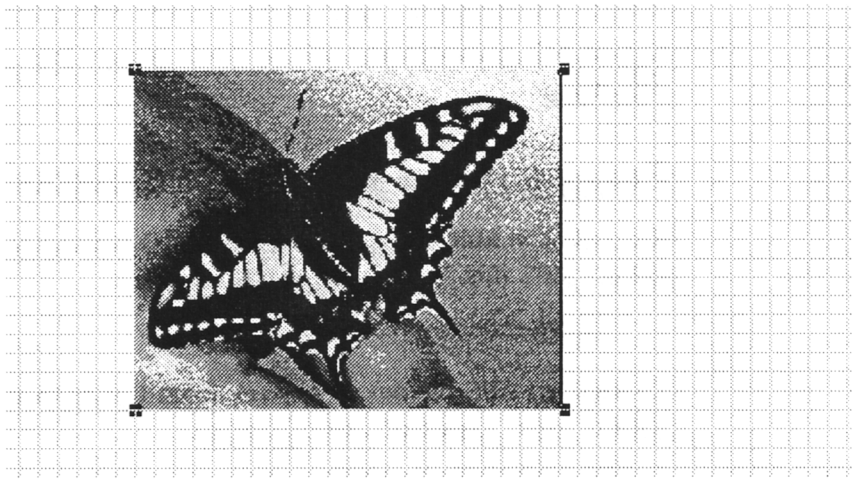


Image Objects

You can add images to a drawing by selecting **Import** from the **File** menu. The editor can import Sun rasterfiles, which often have a **.img** extension (see "The Clipboard" on page 3-18, on how to use the OpenFile dialog to enter an image file name). When the image is loaded, it is added to the drawing in the center of the view.

Figure 89

An image loaded into the editor.



You can also load images with other file formats. To load such an image you must provide a filter to convert the image from its original format into Sun rasterfile format. The name of the filter tells HyperLook what extensions to expect on the image files.

The filter is called with the input file name as the first argument. It should produce the raster file on the standard output.

For example, you might use a filter to convert **tiff** format images into rasterfile format. To do this you create a file called **tiff2img** in a resource directory (for example **~/stacks**). This tells HyperLook that it can convert files with a **.tiff** extension into Sun rasterfiles using the filter. HyperLook will use the filter automatically every time you load a **.tiff** file into the graphics editor.

Suppose you have a filter called **tiffras**, located in the **/usr/local** directory. To install the filter use:

```
ln -s /usr/local/tiffras ~/stacks/tiff2img
```

This will enable you to load images with a **.tiff** extension.

In general, to be able to use other extensions set up a filter in a resource directory **ext2img** if "**.ext**" is the desired extension.

HyperLook provides the **Z2img** filter which lets you load compressed rasterfiles.

Encapsulated PostScript Objects

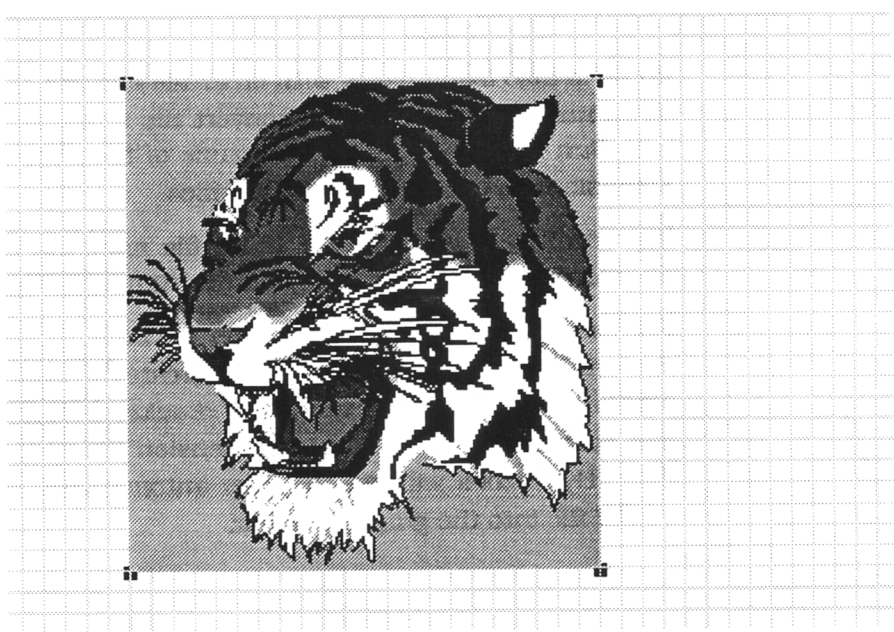
The editor is able to load *Encapsulated PostScript* files (also called EPS or EPSF). EPS files usually start with the line:

```
%!PS-Adobe-2.0 EPSF-1.0
```

To load an EPS file select **Import** from the **File** menu and select a file with a **.eps**, **.epsf** or **.ps** extension.

Figure 90

An Encapsulated PostScript object.



When the file is loaded, it is drawn into an image and the image is displayed. This optimizes the time spent drawing it.

If you prefer the editor to draw the file directly onto the screen, make the object transparent using the fill color tool (see "Selecting the Fill and Line Color" on page 4-17). When the object is transparent, it takes longer to draw onto the screen but it is displayed better at different magnifications.

Solid Objects

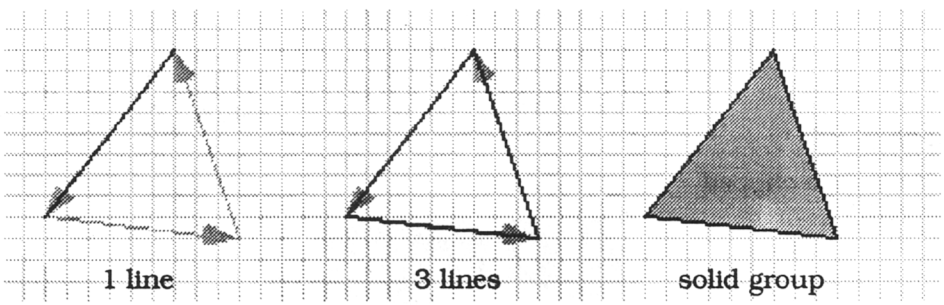
Sometimes you want to draw a shape that is not part of the standard set of graphics objects. An example is a triangle.

The editor lets you to create new shapes by drawing the outline using lines, arcs, rectangles, etc.

To create a triangle, first draw a triangle using three lines, then select all three lines, and select **Solid Group** from the **Special** menu.

Figure 91

Creating a triangle by drawing lines.



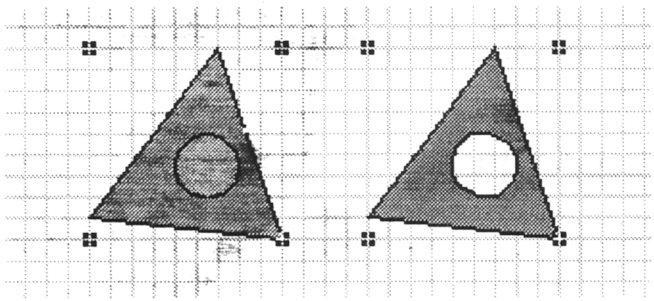
When creating solid objects from lines you must make sure that the lines are properly connected.

The solid group can be ungrouped into its original components with the **Ungroup** item of the **Arrange** menu.

Interesting effects can be created with solid groups. It's possible to create objects that have holes in them. Take a triangle, draw an oval over it, select them both, and make a solid group out of it. The result is a triangle with a hole in it!

Figure 92

Creating a triangle with a hole in it.



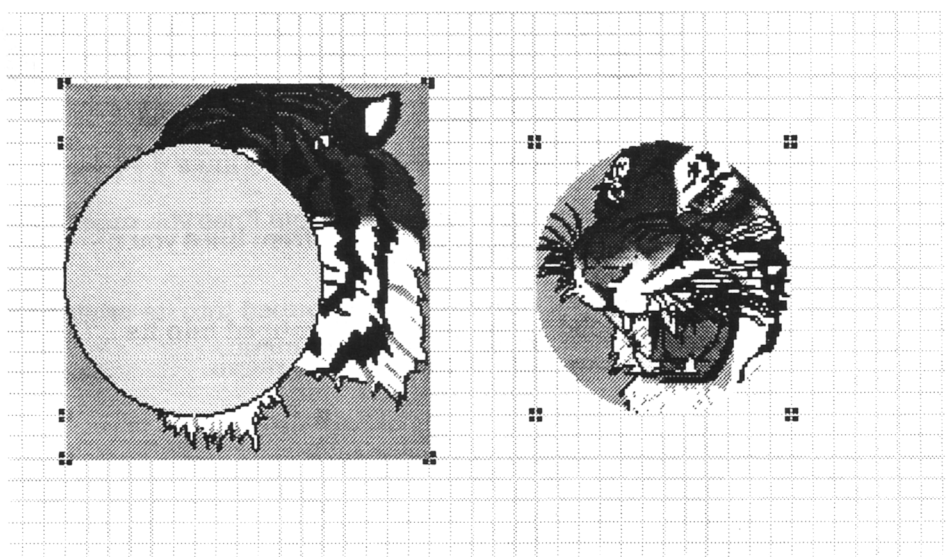
Clipped Objects

The editor lets you draw an object though a stencil of any shape. This technique is called clipping.

To create a clipped object, draw a shape over a drawing, select the shape and the drawing, then choose **`Clipped Group`** from the **`Special`** menu.

Figure 93

Creating a clipped group.

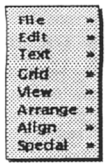


The bounding box of the clipped object is as big as the shape to which the drawing is clipped.

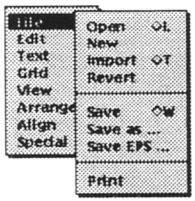
The clipped object can be returned to its original components using **Ungroup** from the **Arrange** menu.



The Graphics Editor Menus



This section explains all the menus and their items. The main menu is shown by holding down the right mouse button anywhere on the paper.



File

Open

Load a drawing (see "Loading Drawings" on page 4-6).

New

Start a new drawing, discarding the current one. You are asked for confirmation.

Import

Import a drawing, Encapsulated PostScript (see "Encapsulated PostScript Objects" on page 4-30) or image (see "Image Objects" on page 4-29).

Revert

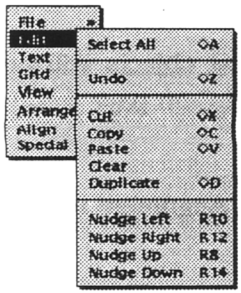
Reload the drawing from its file, discarding any changes made to the current drawing since it was last saved.

Save
Save as...
Save EPS...

Save the drawing (see "Saving Drawings" on page 4-6).

Print

Print the drawing (see "Printing Drawings" on page 4-7).



Edit

Select All

Select all the objects in the drawing (see "Selecting Objects" on page 4-9).

Undo

Undo the previous editing operation (see "Undo" on page 4-23).

Cut, Copy, Paste, Clear

Clipboard editing operations (see "Using the Clipboard" on page 4-22).

Duplicate

Duplicate the currently selected objects. Repeated duplicates will be offset by the same amount (see "Duplicating Objects" on page 4-22).

Nudge Left
Nudge Right
Nudge Up
Nudge Down

Nudge the selected objects in the indicated direction. The distance of the nudge is the grid size. The arrow keys are shortcuts to these menu choices (see "Moving Objects" on page 4-13).



Text

Font

Set the font of the selected text objects (see "Text Objects" on page 4-24 and "Changing the System Fonts" on page 3-31).

Style

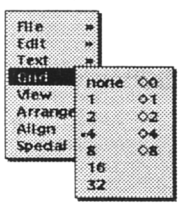
Set the font style of the selected text objects. Options are **Plain**, **Bold**, **Italic** or **BoldItalic**.

Size

Set the size of the selected text objects. The text objects will be resized to fit the new font height (see "Text Objects" on page 4-24).

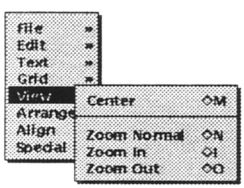
Align

Change the alignment of the selected text objects. Alignment is only significant if the text has multiple lines.



Grid

Change the current grid (see "Using the Grid" on page 4-15).



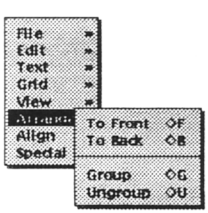
View

Center

Move the selected object into view. If no object is selected, then the whole drawing is moved into view.

Zoom Normal
Zoom In
Zoom Out

Change the magnification (see "Zooming" on page 4-8).



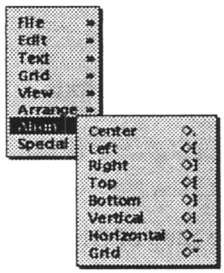
Arrange

To Front
To Back

Change the front to back order of the selected objects (see "Front to Back Order" on page 4-20).

Group
Ungroup

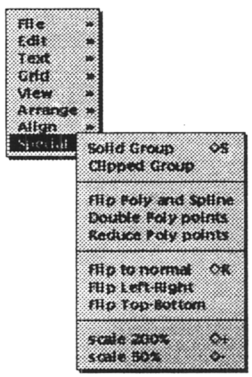
Change the grouping of objects (see "Grouping" on page 4-20).



Align

Align Center
Align Left
Align Right
Align Top
Align Bottom
Align Vertical
Align Horizontal
Align Grid

Align multiple objects (see "Aligning Objects" on page 4-16).



Special

Solid Group

Create a new shape by combining multiple objects (see "Clipped Objects" on page 4-32).

Clipped Group

Clip a part of an object (see "Clipped Objects" on page 4-32).

Flip Poly and Spline

Change a polygon into a spline and vice versa (see "Polygon and Spline Objects" on page 4-26).

Double Poly Points
Reduce Poly Points

Change the number of polygon points (see "Polygon and Spline Objects" on page 4-26).

Flip to Normal

Remove rotation and flipping from an object (see "Rotating Objects" on page 4-14).

Flip Left-Right
Flip Top-Bottom

Flip the selected objects around their vertical or horizontal axis.

Scale 200%
Scale 50%

Make objects larger and smaller (see "Changing the Size of Objects" on page 4-13).

5

Editing Stacks

Every editable HyperLook stack can be in one of two modes[1]. The first is called *browse* mode. In this mode you can use the stack, move from card to card, etc. The other mode is *edit* mode. It lets you change the stack and its objects.

This chapter explains how to create and edit stacks and objects. How to create and use cards and backgrounds is also described in detail (see "Cards and BackGrounds" on page 5-24).

Chapter 6, "Object Properties" explains how to modify HyperLook objects. Chapter 7, "Scripting" explains how to program components and stacks.

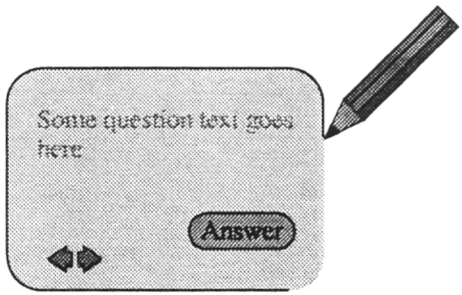
Designing Your Own Stack

Before going into details, it is useful to explain the process by which you create and design stacks.

Normally you will start by creating a new stack from scratch. You then switch the stack into edit mode and add, edit and delete objects, cards and backgrounds.

Figure 94

Designing your own stacks.



When you are happy with your stack, you switch it into browse mode and test it.

When you have finished editing the stack, save it in a file. The stack can be reloaded at any time.

Now you can connect a C program to your stack. See Chapter 8, "Client Programming" on how to communicate with stacks from C programs.

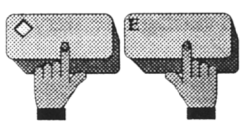
1. Some applications use non-editable stacks.

At any point in the design process, you can test the stack, save it, or revert to a previously saved version. This gives you the confidence to try things out!

Edit Mode

In edit mode you can modify the stack, its cards, backgrounds, and components.

To get to edit mode you can use the stack menu (see "Using Menus" on page 3-5). You will often read instructions like "select **Save** from the **Stack** menu". This really means "select the **Save** option from the **Stack** sub menu of the stack menu".



Select **Edit Mode** from the stack's **Edit** menu to switch to edit mode. To leave edit mode select **Browse Mode** from the **Edit** menu. A quick way to enter and exit edit mode is by typing **Meta-E**.

In edit mode, the objects of a stack do not respond to user input. The stack editor intercepts all user interaction until the stack is in browse mode again.

It is not possible to move a stack when it is in edit mode. To move the stack, first return to browse mode (see "Moving Stacks" on page 3-9).

The fact that a stack is in edit mode is reflected by a change in the cursor. Usually the cursor in browse mode is a little hand, in edit mode the cursor is usually a bull's eye[1].

Figure 95

HyperLook cursors.



browse cursor

edit cursors

1. You will notice that in edit mode the cursor changes depending on its position.

The Stack Menu in Edit Mode

A stack in edit mode has an extended menu. There are 3 different menus which are active depending on the number of selected objects:

- no object selected
- one object selected
- multiple objects selected

Each of the menus is different. The **Look** menu is always available.

Most menu functions apply to either the selected object or objects, the current card, the current background or the stack itself.

If you want to see at a glance what all the menus do, read "Edit Mode Menus" on page 5-36. Otherwise read on and you'll find out about how to edit stacks, encountering options on the stack menu along the way.

Editing and Saving Stacks

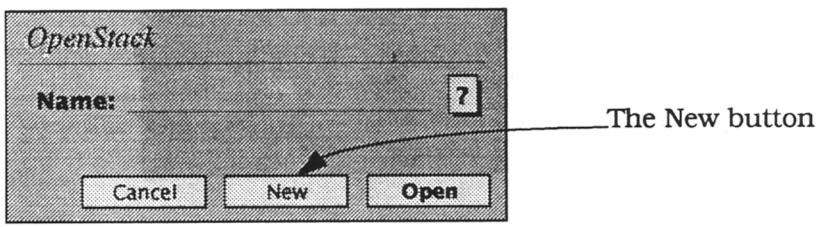
This section explains how to create new stacks and how to save them when you have finished editing them.

Creating a New Stack

To create a new stack, press the **Open** button in the system stack. This will show the OpenStack stack (see "Opening Stacks" on page 3-8).

Figure 96

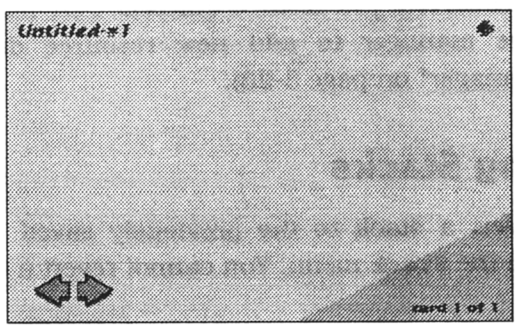
The **new** button in the OpenStack.



Now click on the **New** button and a new untitled stack will appear.

Figure 97

Creating a copy of the Untitled stack.



The standard Untitled stack has some useful components which you can remove if you don't need them.

If you enter the name of an existing stack before hitting the **New** button you will create a copy of the stack. The new stack is given a name that differs from the original.

Another way to create a new stack is to save an existing stack using a new name (see "Saving Stacks" on page 5-6). You can still reload the original stack!

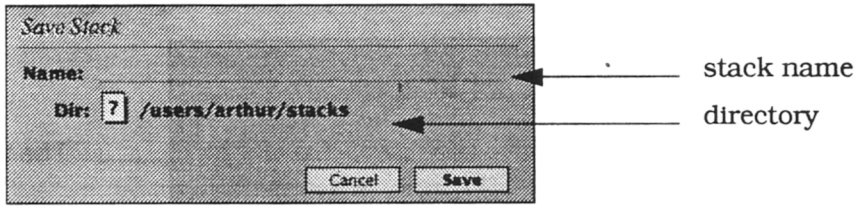
Saving Stacks

After you have made changes to your stack, you can save the stack by selecting **Save** from the **Stack** menu. The stack is saved to the file from which is was loaded.

If the stack is untitled, or if you select **Save As...** from the **Stack** menu, a dialog is shown which lets you choose a new name and a new directory for the stack.

Figure 98

The SaveStack dialog.



Type the name of the stack in the **Name** field and click on the **Save** button. The stack will be saved to the directory which is listed below the name.



The "**?**" pulldown lets you select a different user resource directory. Use the resource manager to add new resource directories (See "The Resource Manager" on page 3-23).

Reverting Stacks

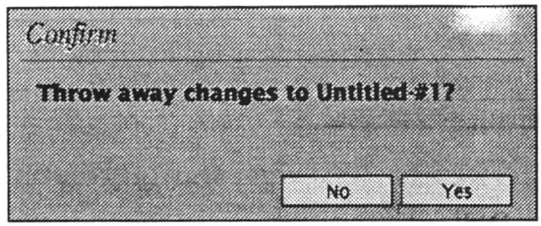
You can revert a stack to the previously saved version by selecting **Revert** from the **Stack** menu. You cannot revert a stack if it has not yet been saved.

Throw Away Changes?

If you try to zap or revert a stack which has been modified, the changes to the stack will be lost. An attempt to do this will show a dialog asking whether you really want to throw away the changes.

Figure 99

Are you sure you want to throw away changes?.



If you really want to throw away the changes, press the **Yes** button. If you have changed your mind and want to save the changes, press **No**.

Printing Cards

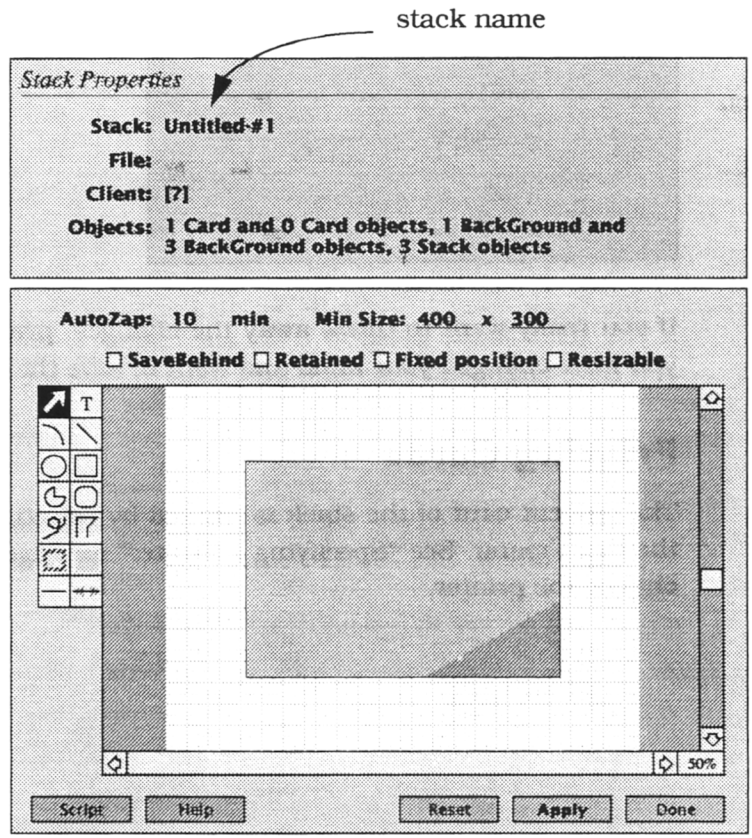
The current card of the stack is printed by selecting **Print Card** from the **Card** menu. See "Specifying a Printer" on page 3-32 if you want to change the printer.

Stack Properties

You can change the properties of a stack by selecting **Stack Info** from the **Stack** menu to show the properties window of a stack.

Figure 100

The properties of a stack.



After changing the properties of the stack press the **Apply** button. You can undo the changes made by pressing **Reset**. When you have finished, press **Done**.

Don't worry about the **Script** and **Help** buttons as they are explained later.

Stack Properties Explained

The stack name field shows the name of the stack. The **File** field shows the file from which the stack was loaded (if any).

The **Client** field shows the name and process id of the client which is connected to the stack. If no client is connected, **[?]** is shown.

The **Objects** field shows some information on the objects, cards and backgrounds of the stack (these are explained later).

The **AutoZap** numeric field lets you specify the time in minutes between the stack being zapped from the screen and really being deleted from memory. See also "The Stack Manager" on page 3-22. If this field is left empty, the stack is not automatically deleted from memory after zapping it (it remains in the trash indefinitely).

The **MinSize** numeric fields let you specify the minimum width and height of the stack. These fields are ignored if the stack is not resizable (see "Making a Stack Resizable" on page 5-11 on how to create resizable stacks).

The **SaveBehind** checkbox determines whether the screen behind the stack is saved to allow for fast drawing when the stack is hidden. This feature is useful for stacks which are visible for a short time only, but you may want to disable this feature if you are short of memory.

The **Retained** checkbox determines whether the stack's content is retained to allow for fast redrawing. This feature is useful when drawing the stack takes a long time. Retained stacks take up a lot of memory!

The **Fixed position** checkbox lets you specify whether the stack can be moved around on the screen or not.

The **Resizable** checkbox indicates whether the stack can be resized or not. See "Making a Stack Resizable" on page 5-11 on how to create resizable stacks.



Designing the Shape of a Stack

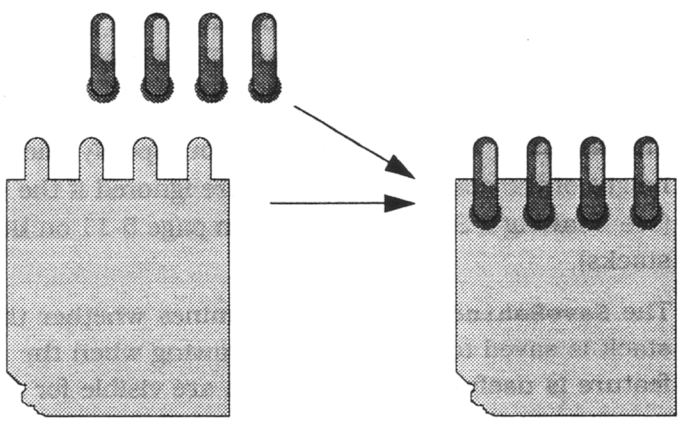
The graphics editor in the stack properties window lets you draw the shape of the stack. The drawing you enter here must consist of at least one graphical object, for example a circle or a rectangle.

The shape of the stack window is defined by the back most graphical object. All the graphics are drawn into that shape.

For example, to create a window with rounded corners you can draw a rounded corner rectangle. You can also create more complex shapes. The Notepad stack is a good example of a stack with a complex shape.

Figure 101

Defining a stack shape for the Notepad stack.



First the ring binder shape was created by linking a set of lines and arcs together into a solid object (see "Solid Objects" on page 4-31). This is then used as the shape of the stack.

Next the rings are drawn on top of the stack shape. The stack shape is now ready. You have created a stack shaped like a ring-binder simply by drawing!

Sometimes the outline of the stack is not completely visible. This is because the outline is clipped by the stack shape. Avoid using complex curves as a stack shape for this reason.

Make sure that the shape of the stack is completely filled with graphics. If parts of the stack's interior are not filled, bits of the things behind the stack may not be painted and show through.

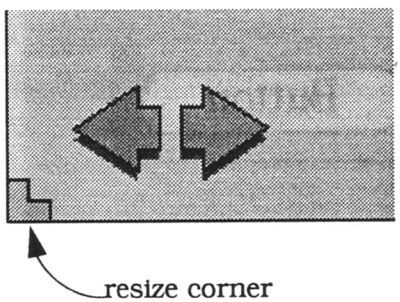
Making a Stack Resizable

By default stacks are not resizable. To make a stack resizable, you must first switch on the **Resizable** property of the stack (see "Stack Properties Explained" on page 5-9).

Now you have to add some resize corners to the stack. The **ButtonIdeas** stack contains a few resize corners you can use. Install all the components of the **ButtonIdeas** stack into the **New** menu by following the directions in "Using the Object Warehouse" on page 5-13.

Figure 102

A resize corner added to the Untitled stack.



When you resize the stack, the stack shape changes. This usually works best for simple stack shapes such as rectangles. Any graphics on a resizable stack should be encapsulated in buttons (see "Drawing Buttons" on page 6-10), otherwise they may be resized incorrectly.

For each object in the stack, you should set its *glue* to make sure that it is moved to its appropriate location after the stack has been reshaped. The glue of an object is set from the object's property stack (see "Glue Property" on page 6-5).

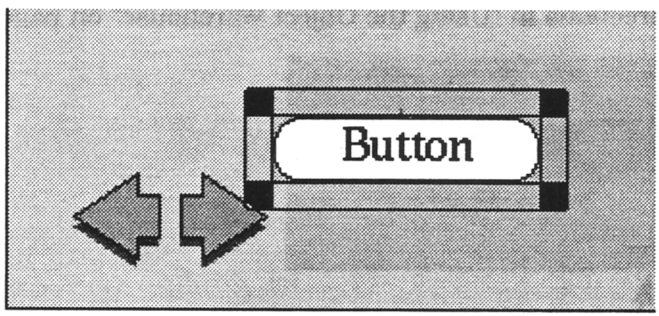
Creating New Objects

Once you have created your own stack, you can create new objects by selecting an object type from the sub menus of the **New** menu.

These menus gives you a choice of HyperLook objects. You can, for example, create a new push button object by selecting **Button** from the **System** sub menu.

Figure 103

Adding a button to the Untitled stack.



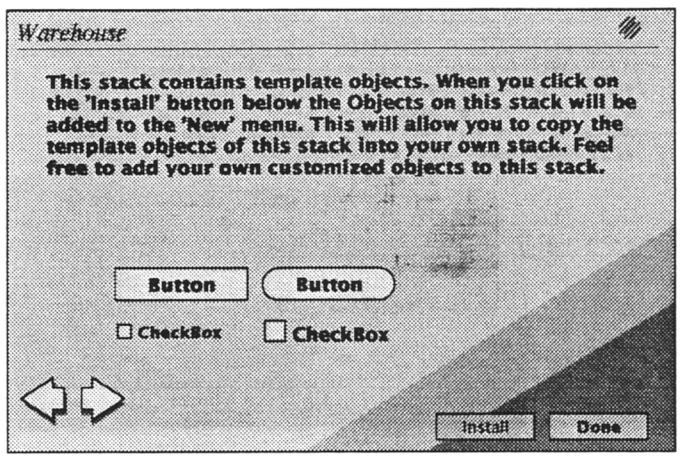
The newly created object is automatically selected. See "Selecting Objects" on page 5-15 for how to select objects.

Using the Object Warehouse

HyperLook provides an object warehouse. To see the warehouse select **Show Warehouse** from the **New** menu.

Figure 104

The object Warehouse.



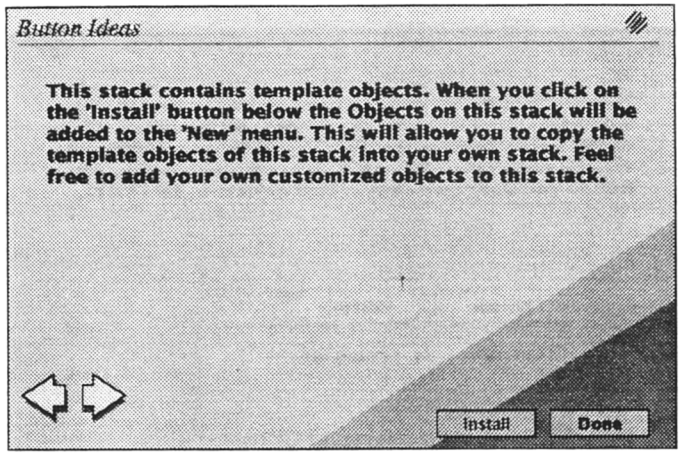
You can copy objects from the Warehouse stack or you can install the **Warehouse** menu by pressing the **Install** button.

After pressing the **Install** button you will find that the **New** menu has a new sub menu called **Warehouse**. This sub menu contains a list of the objects on the Warehouse stack. Selecting an object from this menu creates a new copy of the object from the warehouse.

Another source of objects is the **ButtonIdeas** stack. Use the OpenStack dialog (see "Opening Stacks" on page 3-8) to open the **ButtonIdeas** stack and press **Install**.

Figure 105

The ButtonIdeas stack.



Again you'll see that the **New** menu has a new submenu called **ButtonIdeas**. In particular, the **ButtonIdeas** submenu contains resize corners that may be used when creating a resizable stack.

Editing Objects

Edit cursor 

This section explains how to edit objects on a stack.

A grid is active while editing objects. The grid is not visible but it controls how finely you can adjust the position of objects on the stack. See "Stack Editor Properties" on page 5-33 to see how to configure the stack editor to use a different grid size.

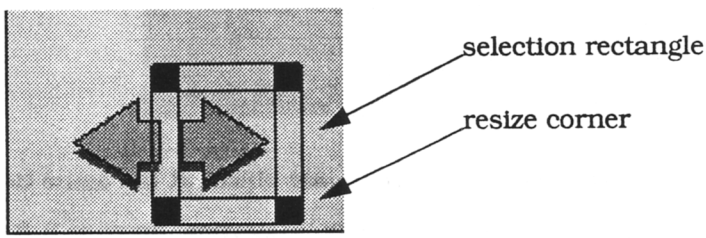
Selecting Objects

Select cursor 

A stack can contain lots of objects. To edit an object you must first select it. A double rectangle with resize corners is drawn around the object to show that it is selected.

Figure 106

A selected object.

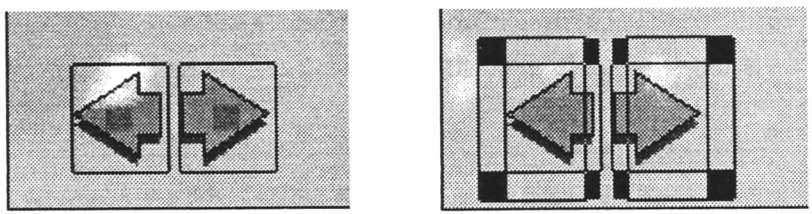


The easiest way to select an object is simply to click on it. If several objects are overlapping, click again to select the object below.

You can select more than one object at a time. Hold down the mouse and drag a rectangle around the center points of the objects that you want to select. While you hold down the mouse button all object rectangles and center points are highlighted.

Figure 107

Selecting multiple objects.



Clicking on an object with the middle mouse button adds or subtracts it from the selection. It is an easy way to select several objects which are far apart.

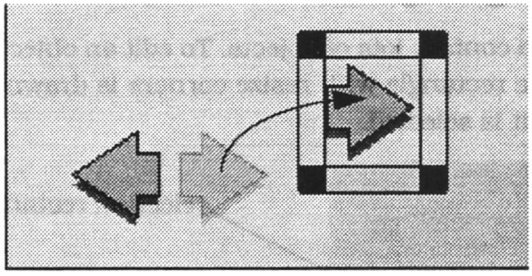
Moving Objects

Move cursor 

An object (or a set of objects) can be moved to a new position by holding down the mouse inside the object's boundary. While holding the mouse down, drag the object to the desired position and release the mouse.

Figure 108

Moving an object.



You can move more than one object at the same time in the same way.

See also "Position and Size Properties" on page 6-6 on how to change the position of an object from the object properties stack.

You can also move objects using the arrow keys on the right hand side of keyboard. The selected objects are nudged in the appropriate direction by one grid measure (see "Stack Editor Properties" on page 5-33 on how to change the grid).

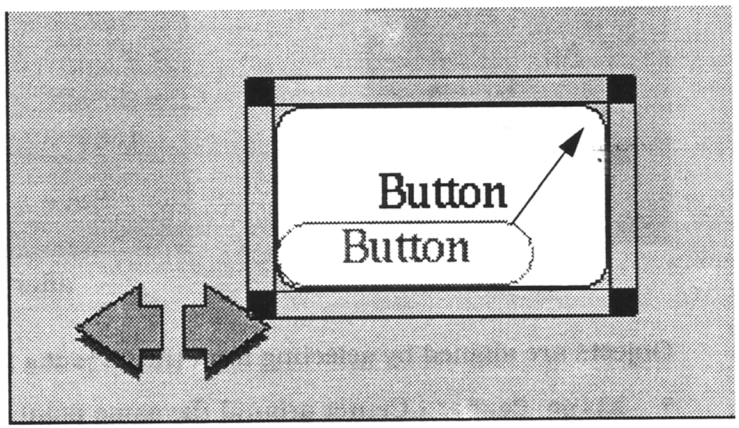
Sizing Objects

Size cursor 

To change the size of an object, hold the mouse down in one of the resize corners of a selected object. While holding down the mouse, change the size by dragging. The object is redrawn at its new size when you release the mouse.

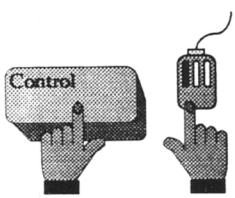
Figure 109

Changing the size of an object.

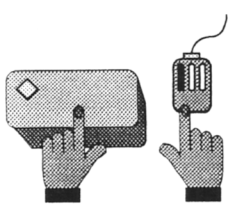


Some objects have a fixed size which does not depend on the object's bounding rectangle. Horizontal scrollbars, for example, have a fixed width.

Constrained Editing



When you resize or move objects it is often useful to constrain the movement horizontally or vertically. Hold down the Control key while sizing or moving objects for constrained editing.



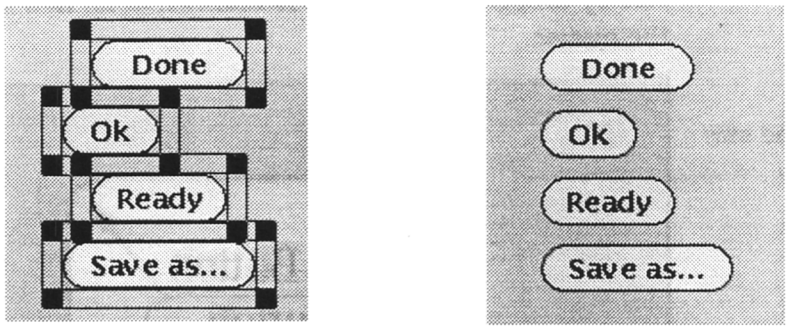
Another constraint is activated when holding down the Meta key while moving or sizing an object. The mouse will move in tiptoe mode. The movements of the mouse are slowed down to give you more control.

Aligning Objects

When more than one object is selected, it is possible to arrange them relative to each other.

Figure 110

Aligning objects using **Align Left**.



before after

Objects are aligned by selecting from the **Objects** menu:

- **Align Center** - Center around the same point.
- **Align Left** - Left side against a vertical line.
- **Align Right** - Right side against a vertical line.
- **Align Top** - Top side against a horizontal line.
- **Align Bottom** - Bottom side against a horizontal line.
- **Align Vertical** - Center along a vertical line.
- **Align Horizontal** - Center along a horizontal line.

The position of the objects is dictated by the first object in the selection. It remains stationary while the other objects are repositioned.

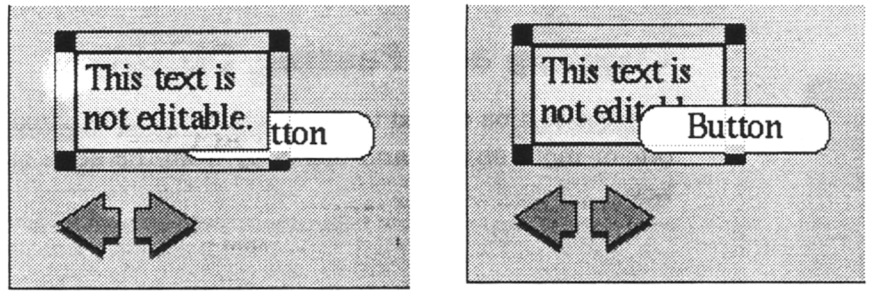
Front to Back Order

The objects of a stack can overlap. Usually this causes no problems. Problems may occur when an object underneath another object redraws itself. Overlapping objects will be affected.

This means that it is only safe to have static objects (such as graphics and static text) overlapped by other objects. It is not safe to have dynamic objects (like editable text) below other objects, because it may cause the stack to be updated incorrectly.

Figure 111

Sending some static text to the back.

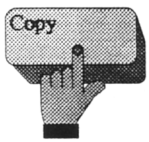


To bring an object to the front, select **To Front** from the **Object** menu. To send an object to the back, select **To Back** from the **Object** menu.

Using the Clipboard

Before reading this section you should be familiar with the use of the clipboard (see "The Clipboard" on page 3-18).

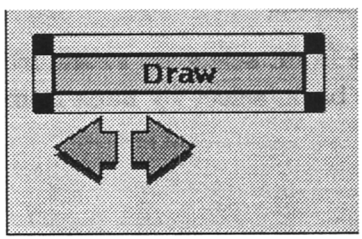
Copying and Pasting Objects



Objects can be copied to the clipboard just like text and graphics. Select one or more objects and use **Copy** from the **Edit** menu, or use the **Copy** key.

Figure 112

A button pasted into the stack.





You can now paste the object into another stack by selecting **Paste** from the **Edit** menu, or use the **Paste** key. Copying and pasting objects is useful if you want to use objects from other stacks.

In the example above the **Draw** button from the system stack was copied from the system stack and pasted into an untitled stack[1].

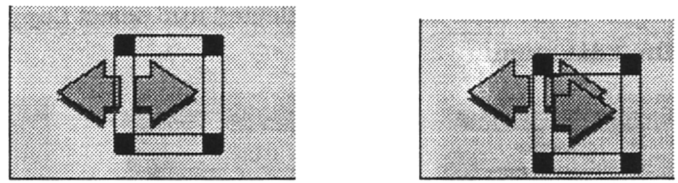
1. It still produces a drawing tool when pressed!

Duplicating Objects

To create a duplicate of an object, select the original and use **Duplicate** from the **Edit** menu.

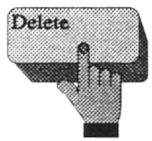
Figure 113

Duplicating an object.



Duplicating an object creates an identical copy of the original. It is offset by a few pixels. Duplication of objects is useful when you want to create a row of buttons each of the same size.

Deleting Objects



To delete an object simply hit the **Delete** key while the object is selected. You can also select **Clear** from the **Edit** menu.



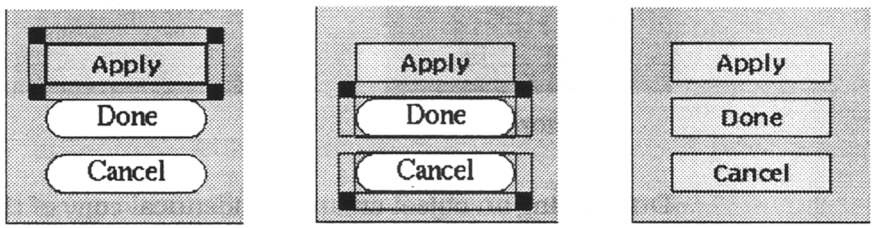
Selecting **Cut** from the **Edit** menu, or using the **Cut** key, copies the object to the clipboard before deleting it.

Copying and Pasting Object Properties

Often you want to copy the *style* of an object to other objects. It is possible to copy an object's properties to the clipboard. Make sure that the object of the desired style is selected and select **Copy Object Props** from the **Edit** menu.

Figure 114

Copying properties.



copy properties | now select and paste | the result

Now select one or more other objects and select **Paste** from the **Edit** menu. The properties of the selected object will be set to be identical to the original.

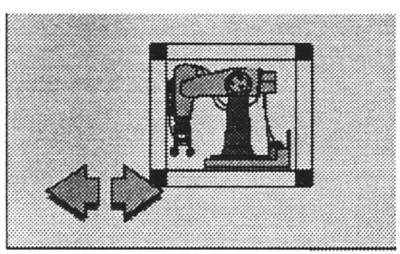
Pasting Text and Graphics

You can paste text directly into a stack when the stack is in edit mode. The text is inserted either as a field object (if it is a single line of text) or as a text object (if it contains newline characters).

Pasting a drawing into a stack creates a button of the right size to hold the drawing (see "Drawing Buttons" on page 6-10).

Figure 115

Pasting a drawing into a stack.

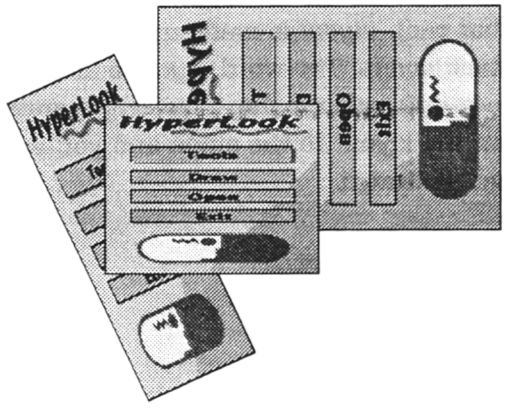


Copy as Drawing

You can copy an object *as a drawing* to the clipboard by selecting the object and using **Copy Object as Drawing** from the **Edit** menu. An Encapsulated PostScript drawing is generated by drawing the object and capturing the PostScript instructions that are executed.

Figure 116

A stack, after being copied as a drawing, really is a drawing!



You can also copy a whole stack as a drawing. When no object is selected, select **Copy Stack as Drawing** from the **Edit** menu. Most illustrations in this manual were created this way (apart from some screen snap shots).

Due its complex nature, the *copy as drawing* operation may take some time[1]. Keep an eye on the SystemStatus stack (see "The SystemStatus Stack" on page 2-6). It will tell you when the drawing is ready. Then you can paste it into a graphics editor.

1. It can take from a few seconds to a full minute for complex stacks.

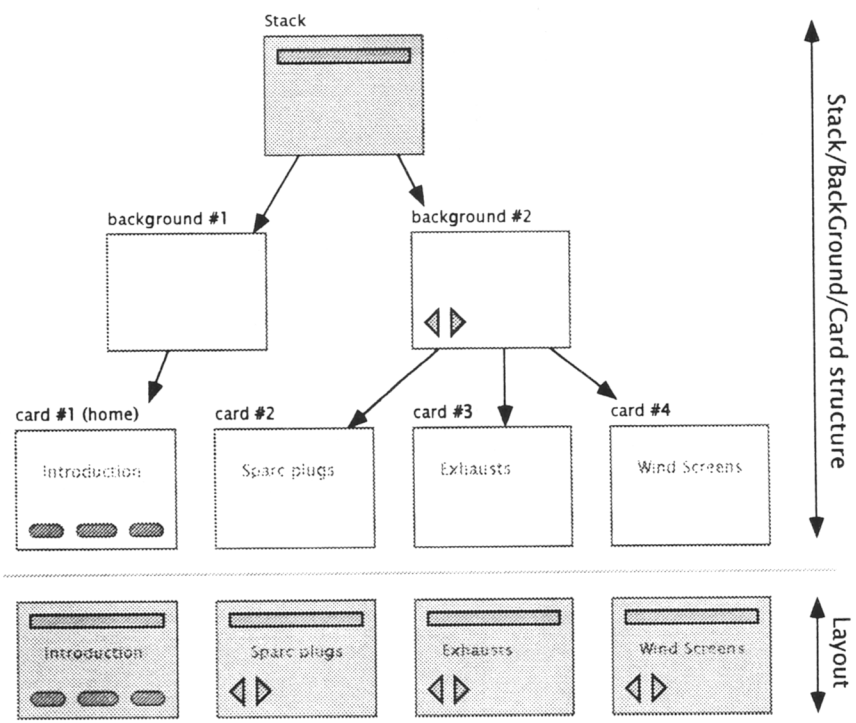
Cards and BackGrounds

Each stack can have multiple cards and each card has a background which it can share with other cards. See "Using Stacks" on page 3-7 to see how to move between cards in a stack. The stack, background and cards are arranged in a tree structure as shown in Figure 117.

You can see that each *layout* is made up of objects from the card, background and stack layers. The word *layout* is normally dropped, because there is a one-to-one relationship between layouts and cards. Trace a path back from the layout to the stack and you will be able to work out how each layout is drawn.

Figure 117

An example of how cards and backgrounds are organized within a stack.



The objects of a stack can be on one of three layers:

1. *Stack layer.* Objects on this layer appear on every card. It is often used to hold components such as resize corners.
2. *Background layer.* Objects on this layer appear only if the current card has this background. Multiple cards can share the same background. This layer is therefore used to hold objects which should appear on a set of cards.
3. *Card layer.* Objects on this layer appear only if the card is visible.

Objects on the stack layer *always* appear behind objects from the background or card layer. Objects from the background layer always appear behind objects from the card layer but in front of objects from the stack layer. See "Generic Properties" on page 6-3 on how to specify the layer of an object.

Adding and Deleting Cards

To add a card simply select **`New Card`** from the **`Card`** menu. This adds a card just after the current card.

The new card will have the same background as the current card. This means that the stack and background objects of the current card will appear on the new card too. The card itself will be empty.

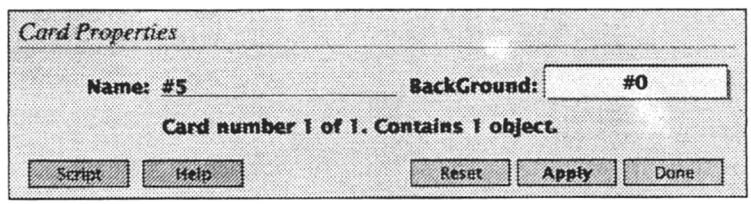
To delete the current card select **`Delete card`** from the **`Card`** menu. A stack must contain at least one card.

Card Properties

Selecting Card Info from the Card menu show the card properties stack.

Figure 118

The properties of a card.



After changing the properties of the card, press the **Apply** button. You can undo the changes made to the properties by pressing **Reset**. When you are finished press **Done**.

Don't worry about the **Script** and **Help** buttons they are explained later.

Card Properties Explained

The **Name** field lets you specify the name of the card.

The **BackGround** pulldown lets you select either **New BackGround** to give the card a completely new background, or the name of an already existing background. See "Using BackGrounds" later in this section on how backgrounds are used.

Copying Cards to the Clipboard

You can copy a card and its contents to the clipboard by selecting **Copy Card** from the edit menu.

Pasting the card into a stack (when in edit mode) causes the card to be added after the current card of the stack. The card will have the same background as the current card.

Using BackGrounds

Backgrounds are useful if you want some objects to appear on a group of cards. Figure 117 on page 5-24 shows an example of an application where the next and previous buttons (the arrows) appear on every card of the stack except the first one.



This arrangement is achieved as follows:

1. Create a new stack.
2. Edit the first card of the stack.
3. Add a new card to the stack, then go to the new card and give the new card a new background (see "Card Properties" explained earlier).
4. Create next and previous buttons and set their layer property to BackGround (see "Generic Properties" on page 6-3).
5. Add two new cards to the stack. They will both use the background with the previous and next buttons.

The advantage of using backgrounds is that the objects on the background are shared between cards[1]. If you make changes to an object on a background these will affect all the cards that share the background[2].

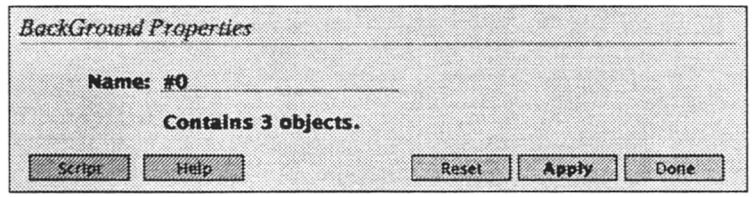
You can assign a new background to the current card by selecting **New Background** from the **Card** menu. A new background can also be assigned from the card properties (see "Card Properties" earlier in this section).

BackGround Properties

Selecting **Background Info** from the **Card** menu shows the properties of the current background.

Figure 119

The properties of a background.



The only property that you can change for a BackGround is the name. Enter the name in the **Name** field.

1. This saves memory!
2. This saves time!

After changing the properties of the background, press the **Apply** button. You can undo the changes made to the properties by pressing **Reset**. When you are finished press **Done**.

Don't worry about the **Script** and **Help** buttons as they are explained later.

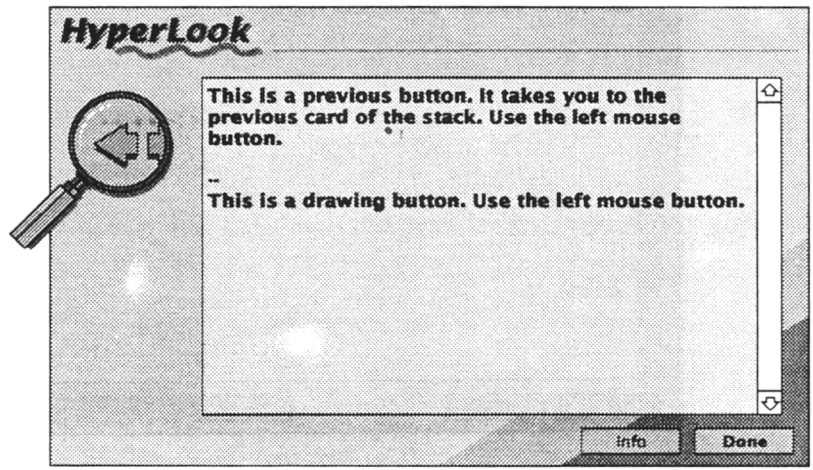
Defining Help



You can add help text to any HyperLook object. This text is shown when the **Help** key is pressed while the mouse is over the object.

Figure 120

Getting help text on an object.



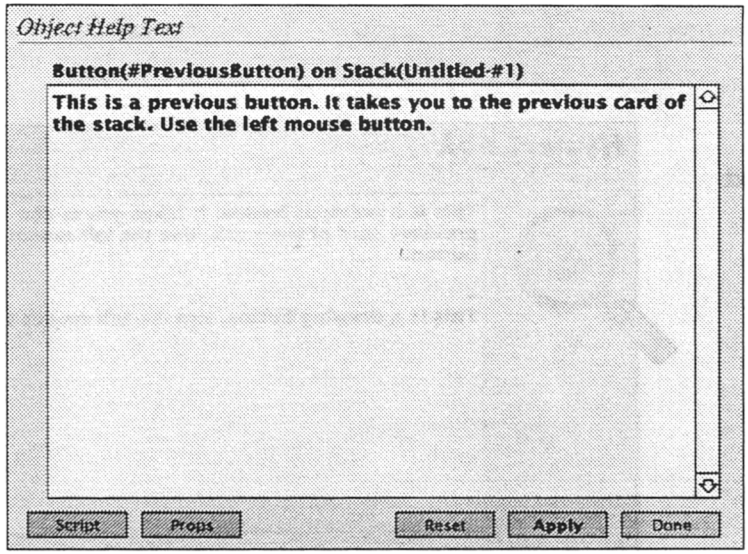
The section "Using Help" on page 3-3 describes how to get help on HyperLook objects while you are in browse mode. In this section, you'll find out how to change the help text for HyperLook objects. You do this in edit mode, and you don't need to do any programming!

Editing Help Text

When you press the **Help** key in *edit* mode, a text editor is shown which lets you edit the help text of the selected object.

Figure 121

Editing the help text of an object.



After editing the help text, press the **Apply** button. Press **Reset** to undo the changes that you have made. When you have finished editing the help text, press **Done**.

You can also edit the help text editor by pressing the **Help** button on the property stack of an object, card, background or stack.

Looking for Help

Every object, including stacks, cards and backgrounds, can have help text. When the Help key is pressed, the text is located in the following order:

1. The object under the cursor
2. The current card
3. The current background
4. The stack

If no help is found, then some generic help is shown for the object or the current card. If help text is found at more than one level, the first help found is shown.

The way help text is located lets you specify generic help for a stack. If no help text is defined for an object, the current card and the current background, then the stack's help text is shown. It also lets you define help text per card or per background.

To edit the help text of a stack, background or card, press **Help** in the relevant property window. It is really very easy to add help to your application.



Black and White Stacks

HyperLook stacks that are designed for a color screen may look bad on black and white screens. This is because colors with different hues can have equal intensities. Also, the stipple patterns which are used to represent colors of varying intensities can look bad on low resolution screens.

You can design black and white versions of your stacks, and save them in a sub directory of your resource directory called **mono**. The resource manager will add the **mono** directory to the resource directory list only when HyperLook is running on a black and white screen.

See "The Resource Manager" on page 3-23, which explains how to add and remove resource directories from the resource directory list (see also **hlpath(1)** on page C-5).

You don't need to worry about the **mono** directory if you design your application for black and white screens only.

Customizing Your Editing Environment

HyperLook lets you customize your stack editing environment in a number of ways. You can define new types of stacks, new types of object, and you can customize the stack editor.

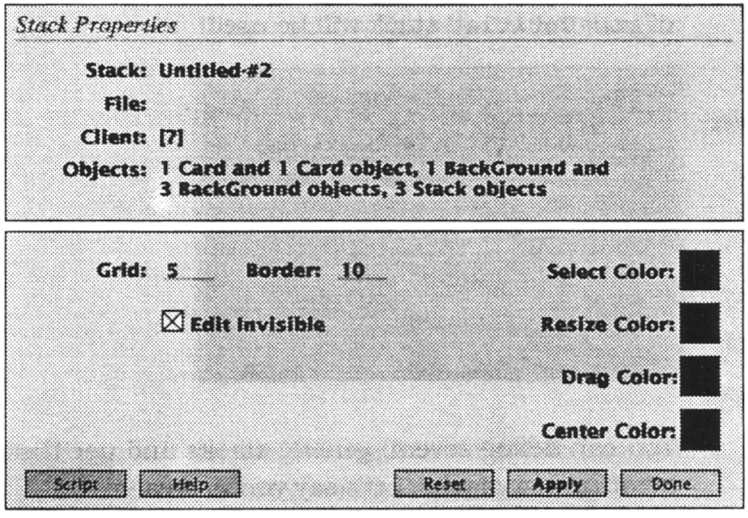
This is very useful, for example, if you want to impose a company style on all stacks and objects.

Stack Editor Properties

Selecting **`Editor Info`** from the **`Edit`** menu shows the editor properties for the stack. Each stack has its own editor properties. This means that you can configure each stack differently.

Figure 122

The editor properties of a stack.



Stack Editor Properties Explained

The **`Grid`** numeric field lets you set the grid size which is used when dragging and resizing objects in the stack.

The **`Border`** numeric field defines how big the resize corners of selected objects are. The wider the resize corners, the easier it is to resize objects.

The **Edit Invisible** checkbox specifies whether invisible objects can be selected and edited.

The **Select Color**, **Resize Color**, **Drag Color**, and **Center Color** color selectors let you choose various colors which are used by the editor to draw the selection borders etc.

Creating Your Own Untitled Stack

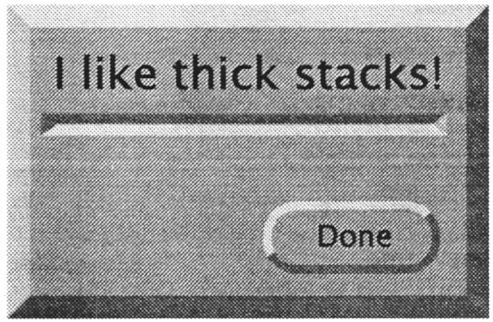
HyperLook lets you re-define the **Untitled** stack This means that you can create your own style of stacks.

Create a stack to your own specification, then save it using **Save As ...** from the **Stack** menu, with the name **Untitled**.

Now when you press the **New** button in the OpenStack stack a new copy of *your* **Untitled** stack will be used!

Figure 123

Creating your own style of stack.



You can define several generic stacks and use them as templates when creating new stacks. Let's say you've created a stack called **Template**. To create a copy of this stack, use the **OpenStack** dialog (see "Creating a New Stack" on page 5-5), type **Template** in the **Name** field and press **New**.

Creating Your Own Object Warehouse

When designing your own systems with HyperLook, you may want to create your own set of object templates. To create your object warehouse, create a stack with template objects. Each card of the stack should hold a set of related objects.

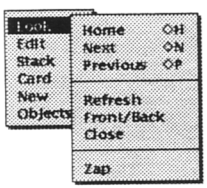
The names of the template objects should reflect what they do, since the name will be used as a menu item (see "Object Name Property" on page 6-4 on how to set the object name). Objects that have a name which starts with "#" are not considered as template objects.

Copy the **Install** button from the **Warehouse** stack and paste it into your warehouse stack. Now you are ready to install the template objects in to the **New** menu.



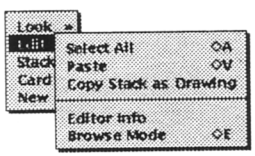
Edit Mode Menus

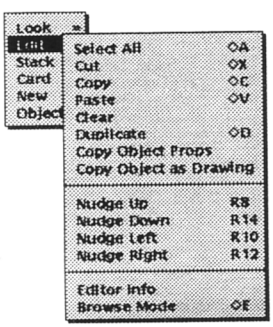
Some of the menus explained here provide different choices depending on whether *no* object is selected, *one* object is selected or *several* objects are selected.

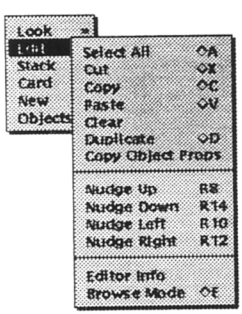


Look

The **Look** menu is explained in "The Stack Menu" on page 3-36.







Edit

Select All

Select all the object on the stack, current background and current card (see "Selecting Objects" on page 5-15).

Cut
Copy
Paste
Clear
Duplicate

Clipboard editing operations. Only **Paste** is available when no objects are selected (see "Copying and Pasting Objects" on page 5-20).

Copy Object Props

Copy the properties of an object to the clipboard (see "Copying and Pasting Object Properties" on page 5-22).

Copy Object as Drawing
Copy Stack as Drawing

Copy an object (if one object is selected) or copy the stack to the clipboard as a drawing (see "Copy as Drawing" on page 5-23).

Nudge Up
Nudge Down
Nudge Left
Nudge Right

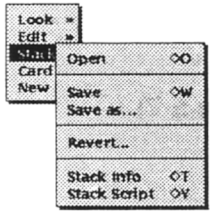
Nudge the selected objects in the indicated direction (see "Moving Objects" on page 5-16).

Editor Info

Show the editor properties of the stack (see "Stack Editor Properties" on page 5-33).

Browse Mode

Switch the stack out of edit mode (see "Edit Mode" on page 5-3).



Stack

Open

Show the OpenStack dialog (see "Opening Stacks" on page 3-8).

Save
Save as...

Save the stack (see "Saving Stacks" on page 5-6).

Revert

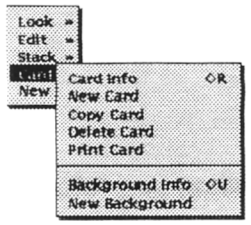
Revert the stack to the previously saved version (see "Reverting Stacks" on page 5-6).

Stack Info

Show the stack properties (see "Stack Properties" on page 5-8).

Stack Script

Show the stack's script (see "Scripting" on page 7-1).



Card

Card Info

Show the properties of the current card (see "Card Properties Explained" on page 5-26).

New Card

Create a new card (see "Adding and Deleting Cards" on page 5-25).

Copy Card

Copy the current card to the clipboard (see "Copying Cards to the Clipboard" on page 5-26).

Delete Card

Delete the current card (see "Adding and Deleting Cards" on page 5-25).

Print Card

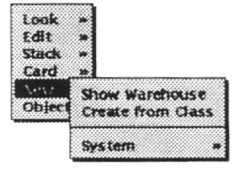
Print the current card (see "Printing Cards" on page 5-7).

Background Info

Show the properties of the background of the current card. See "BackGround Properties" on page 5-27.

New Background

Give the current card a new background (see "Using BackGrounds" on page 5-26).



New

Show Warehouse

Show the object warehouse (see "Using the Object Warehouse" on page 5-13).

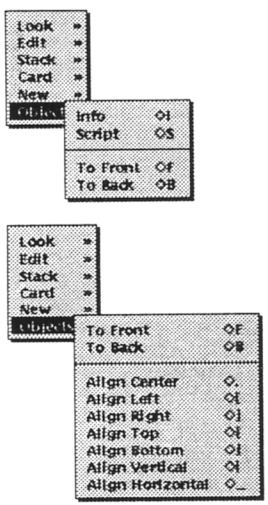
Create from Class

Create an object from a resource class (see "Scripting" on page 7-1).

System

Create a new system object (see "Creating New Objects" on page 5-12).

More submenus can be added to the **New** menu (see "Using the Object Warehouse" on page 5-13).



Object(s)

Info

Show the object properties. This menu item is only available if a single object is selected (see "Object Properties" on page 6-1).

Script

Show the script of the selected object. This menu item is only available if a single object is selected (see "Scripting" on page 7-1).

To Front
To Back

Send objects to the front or to the back (see "Front to Back Order" on page 5-18).

Align Center
Align Left
Align Right
Align Top
Align Bottom
Align Vertical
Align Horizontal

Align multiple objects. This menu item is only available when more than one object is selected (see "Aligning Objects" on page 5-18).

6

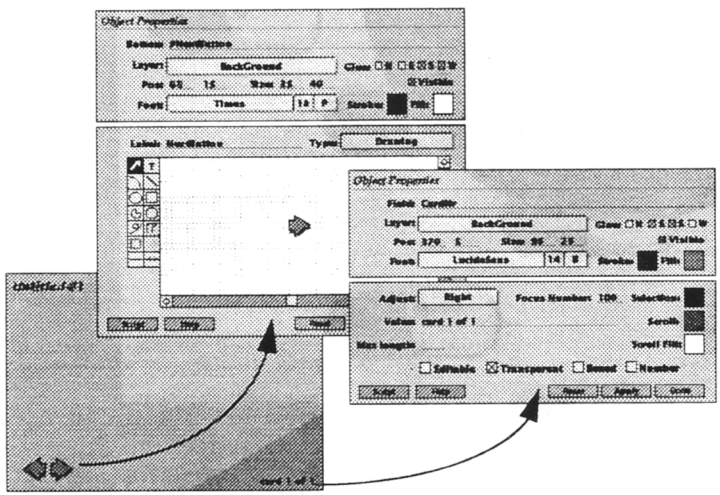
Object Properties

This chapter explains the properties of the HyperLook user interface objects. In Chapter 5, "Editing Stacks" you found out how to create and edit objects interactively.

In this chapter you will find out how to change an object's properties so that it looks just the way you want it to. You have the freedom to change object properties such as, color, font, shape and size. Every HyperLook object has properties which can be changed.

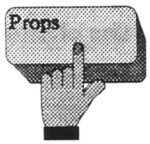
Figure 124

Every object has properties.



Before reading this chapter, you should be familiar with Chapter 3, "Using HyperLook" and Chapter 5, "Editing Stacks".

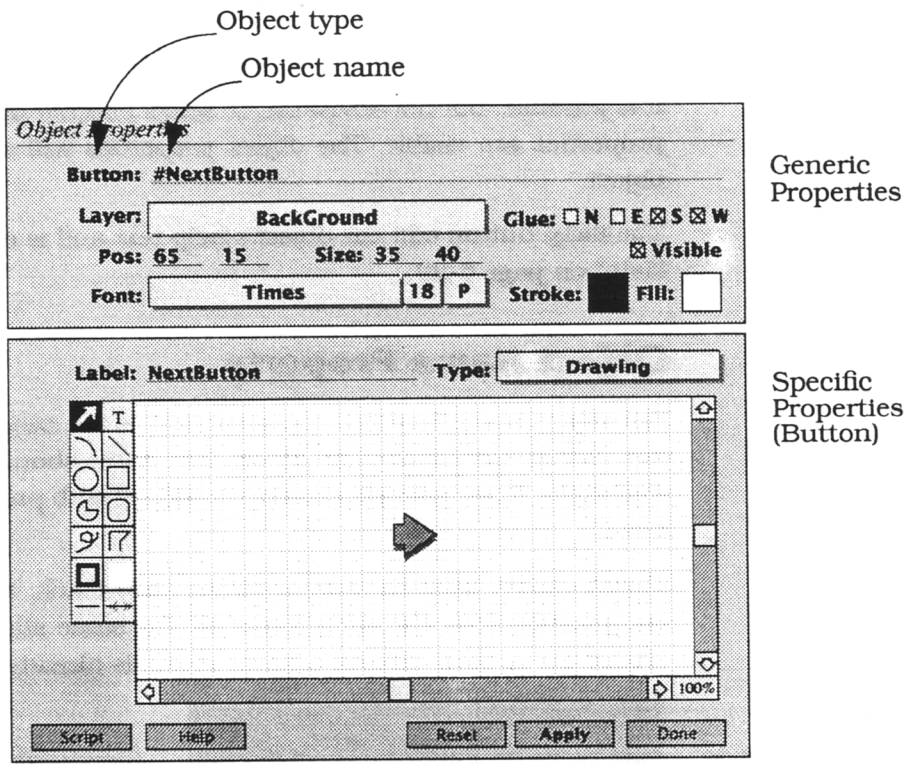
Generic Properties



When editing a stack you can edit the properties of an object by selecting the object and then choosing **Info** from the stack's **Object** menu. You can also double click on the selected object or press the **Props** function key on the keyboard, while the object is selected. This shows the object's property stack.

Figure 125

The properties of a Button object.



The property stack is divided into two sections. The top half contains properties that apply to every object in the system (generic properties). The bottom half contains the properties that apply only to the type of object which you have selected (specific properties).

The generic properties are explained next. After that you'll find out about the specific properties of each type of object.

Editing Properties

When editing properties, no changes are made to the selected object until you press the **Apply** button. The properties are then applied to the object and the object is given the new values.

You can throw away changes made to the properties by pressing **Reset**. When you have finished editing the properties press **Done**. Remember to press **Apply** if you want to make the changes permanent. You will be warned if you press **Done** without first pressing **Apply**.

The **Script** button shows the object's script. See Chapter 7, "Scripting".

It is possible, but not advisable, to select another object while an object's properties are visible. The object properties will still apply to the first object.

The **Help** button edit the object's help text and is explained in "Defining Help" on page 5-29.

Object Name Property

The object name field (the top most field in the **ObjectProps** stack), lets you change the name of an object. The name should contain only letters and digits. This makes it easier to interface with programming languages like C.

Object names must be unique within each stack. You cannot define two components with the same name on the same stack. A warning is displayed if you have chosen a name which is already in use.

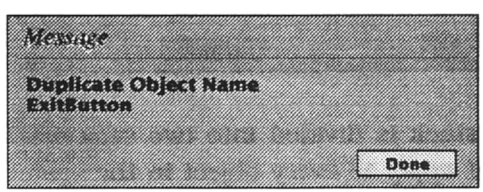


Figure 126

Somebody tried to create a second object called **ExitButton**.

Layer Property

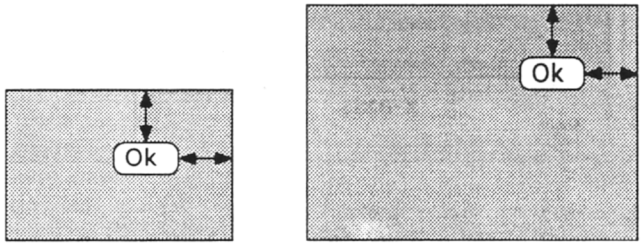
The **Layer** pulldown lets you select the layer of the object. You can select **Stack**, **Background** or **Card**. See "Cards and BackGrounds" on page 5-24 on how these layers affect the object's behavior.

Glue Property

The **Glue** checkboxes[1] let you specify the way an object behaves when the stack is resized (see "Making a Stack Resizable" on page 5-11 on how to make a stack resizable). You can define how the object is *glued* to the edges of the stack.

Figure 127

An object glued to the north and to the east edge of the stack.

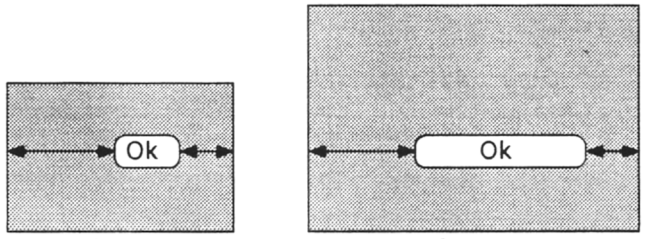


For example, an object which is glued to the north edge and to the east edge of the stack will remain in the same position relative to the north east corner of the stack.

If an object is glued to opposing edges, i.e. the west and the east edge, it will be stretched in order to satisfy the glue constraints.

Figure 128

An object glued to the east and to the west side of the stack.



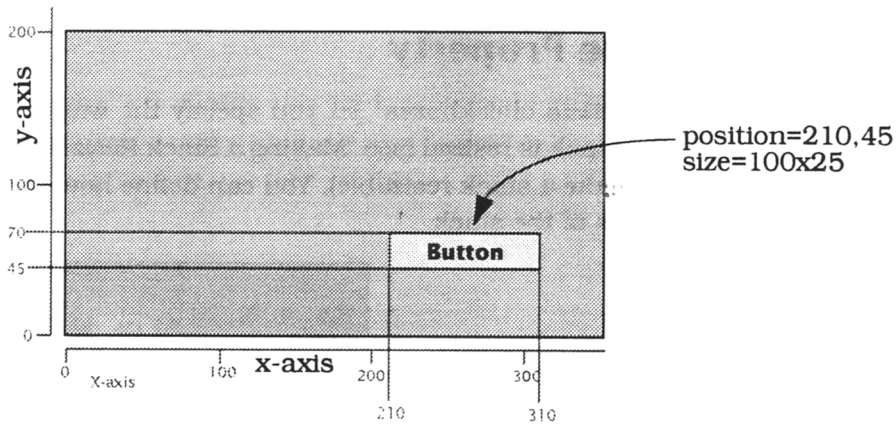
1. **N** = north, **E** = east, **S** = south, **W** = west

Position and Size Properties

The **Pos** and **Size** numeric fields let you specify the exact position and size of the object. See "Moving Objects" on page 5-16 and "Sizing Objects" on page 5-17 on how to change these parameters interactively.

Figure 129

The coordinate system of a stack.



The coordinates are relative to the bottom left corner of the stack.

Visible Property

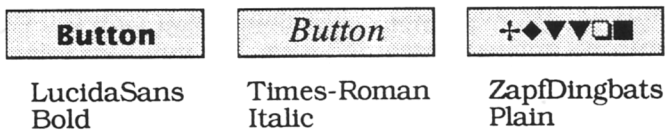
The **Visible** checkbox lets you select whether an object is visible or not. It is sometimes convenient to make an object invisible. The user can not interact with an invisible object in any way, until you make it visible again!

Font Property

The **Font** pulldowns let you change the font, font size and font style[1] of the object. See "Changing the System Fonts" on page 3-31 to see how to change the list of fonts.

Figure 130

Buttons with different fonts.

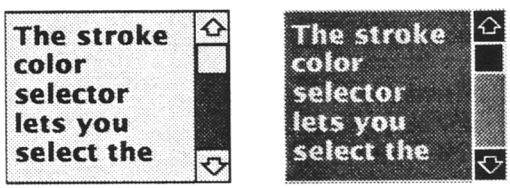


Color Properties

The **Stroke** color selector lets you select the stroke (line and text) color of the object. The **Fill** color selector lets you select the fill (interior) color of the object.

Figure 131

Changing the color properties.



See "The Color Pallet" on page 3-20 on how to change the default set of colors.

These are the generic object properties. They apply to all object types. Let's now look at specific object properties for different types of object.

1. **P** = plain, **B** = bold, **I** = italic, **BI** = bold and italic

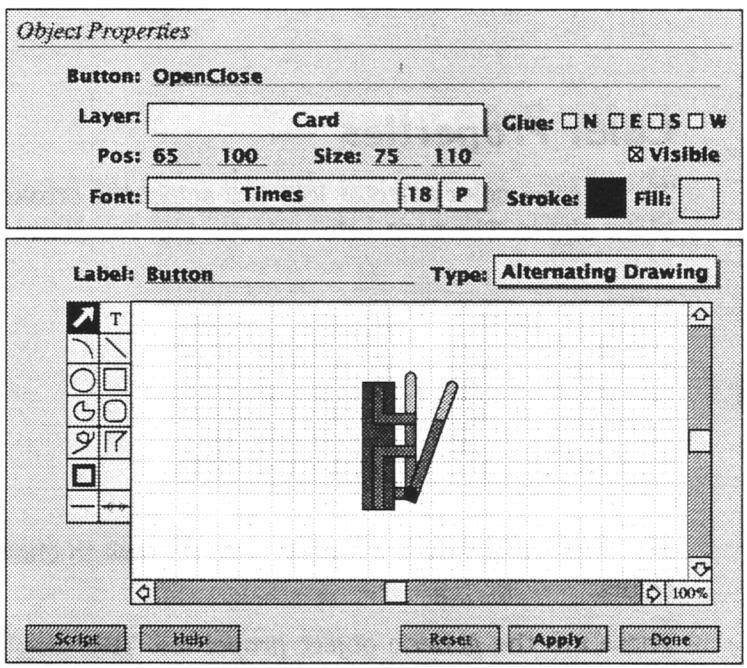
Button Properties

Buttons do something when you press them and they are used a lot in HyperLook. Lots of different types of button are provided, but you can also draw your own with the graphics editor.

Buttons are activated by pressing them with the left mouse button. The button's action is executed when the mouse is released inside the perimeter of the button.

Figure 132

The properties of a Button object.



We're only interested in the specific button properties, so concentrate on the bottom half of Figure 132.

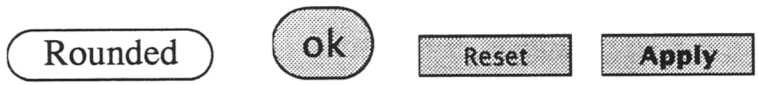
The **Type** pulldown lets you select the type of button. You can choose from **Rounded Button**, **Square Button**, **Transparent Button**, **Checkbox**, **Small Checkbox**, **Drawing** and **Alternating Drawing**. The button types are discussed below.

Push Buttons

Rounded push buttons and square push buttons are the most common types of button. When the mouse is clicked in the button, it highlights. When the mouse is released, the button returns to its normal state.

Figure 133

Examples of rounded and square buttons.



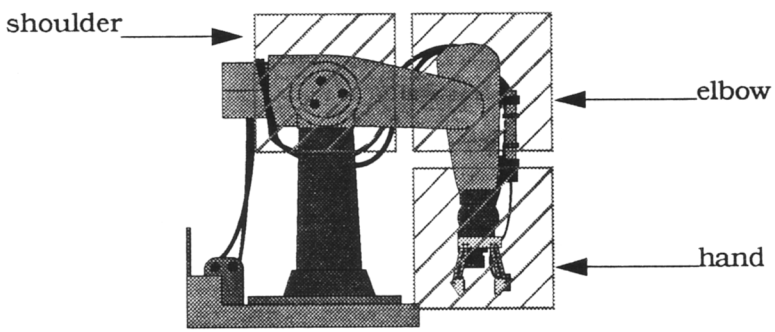
Fill in the **Label** field to specify the text displayed in the button.

Transparent Buttons

Transparent buttons are used to detect mouse clicks in an area. You can overlay them on a drawing of some sort to detect a click in a certain part of the drawing.

Figure 134

Transparent buttons on top of a drawing.



Transparent buttons act as normal buttons but they have no visual appearance, so you can't see them. You can specify a label for a transparent button if you want to, but you won't see it!

Checkboxes

A checkbox button is used to indicate a Boolean state, on or off. They come in two sizes, big or small.

Figure 135

Examples of checkboxes.

☐ Married ☒ **Happy** ☒ **Visible**

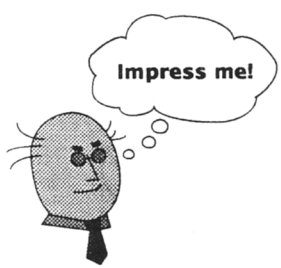
The value of a checkbox is `0` for false (not highlighted) and `1` for true (highlighted). Fill in the **`Label`** field to specify the text printed next to the text box.

Drawing Buttons

A drawing button is most frequently used to display a drawing in a stack. Pasting a drawing from a graphics editor into a stack automatically creates a drawing button containing the drawing (see "Pasting Text and Graphics" on page 5-22).

Figure 136

A drawing button.



The graphics editor lets you create or edit the drawing for the button. Note that you sometimes need to adjust the size of the button before you can see the whole drawing.

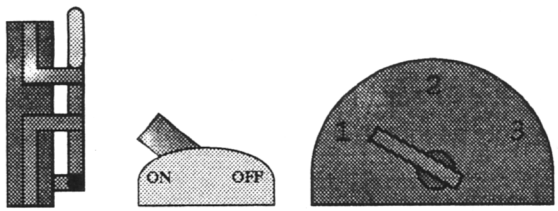
Alternating Drawing Buttons

An alternating drawing button contains multiple drawings, which are used to display different states of the button. There is one drawing for each state of the button, and each time you press the button the drawing changes from one to the next.

In this way it is easy to create graphically pleasing buttons like switches and levers.

Figure 137

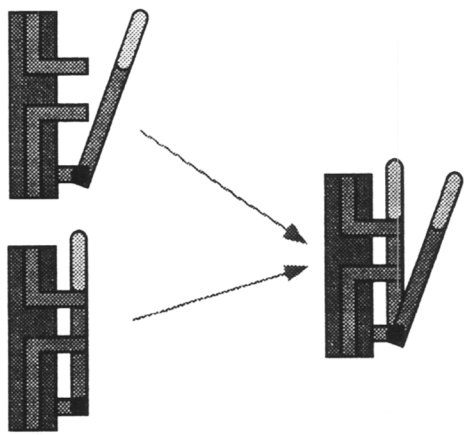
Examples of alternating drawing buttons.



To specify the states for the button you need to draw each state in the graphics editor on the button properties stack. Each state should be a single drawing object, so group objects together so that you have one object per state. Place all the states on top of each other starting with the initial state at the bottom of the pile.

Figure 138

Button states drawn on top of each other.



Now you have given HyperLook all the information it needs to create an alternating drawing button.

Note that the button will redraw between states using the fill color of the button as the background of the drawing. Make sure that the fill color matches the background color of the window!

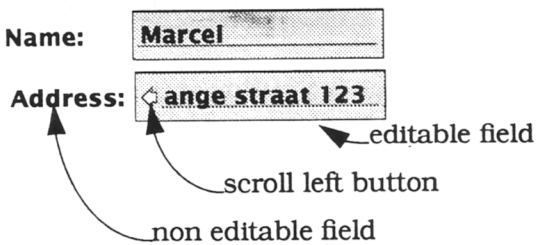
Field Properties

A field is a single line of text which can scroll horizontally. It is used for labels or to let the user enter a single line of text.

You usually finish editing a text field by hitting the **Return** or **Esc** key, then the action associated with the field is then executed.

Figure 139

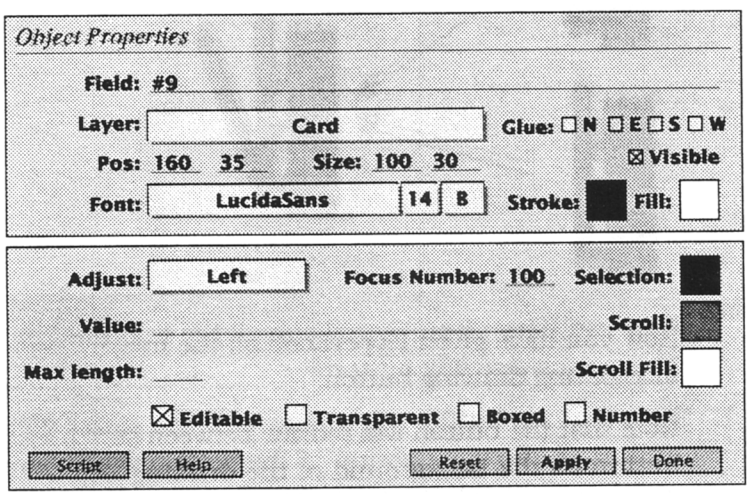
Examples of field objects.



See "Editing Text" on page 3-14 on how to edit text using keyboard short cuts. You can copy and paste text to and from fields using the clipboard, see "The System Properties" on page 3-31. See "Text Properties" on page 6-15 if you want to use a multi line text editor.

Figure 140

The property stack of a Field object.



Field Properties Explained

We're only interested in the specific properties of field objects, so concentrate on the bottom half of Figure 140.

The **Value** field lets you enter the text to be displayed in the field. You can do this even if the text field itself is not editable.

The **Transparent** checkbox lets you choose whether the text field is transparent or opaque. A transparent field can not be edited.

The **Boxed** checkbox lets you choose whether a box is drawn around the text field. The border color of the box is the same as the text color.

The **Selection**, **Scroll** and **Scroll Fill** color selectors let you change the colors of selected text and scroll buttons.

The **Editable** checkbox determines whether the user can type into the field directly. If the field is not editable it is usually used for display only.

The **Adjust** pulldown lets you select how the field is adjusted. You can choose **Left**, **Center** or **Right**.

The **Max length** field lets you limit the length of the text field to a maximum number of characters. If you don't enter a value here, then the maximum length is 65000 characters.

Numeric Fields

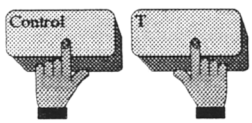
The **Number** checkbox is used to specify whether or not a field is numeric. A numeric field restricts the user to entering numbers. Numbers can start with a "**-**" and may contain a decimal point.

Controlling the Input Focus

The object which is receiving typed characters is said to hold the input focus. It's useful to be able to control the input focus so that user of your application can enter text in a logical sequence.

The **Focus Number** field lets you specify the focus number of the field. This number is used to establish some ordering between text objects when more than one editable text object is visible. The order is used to decide the sequence in which input focus is passed around between objects.

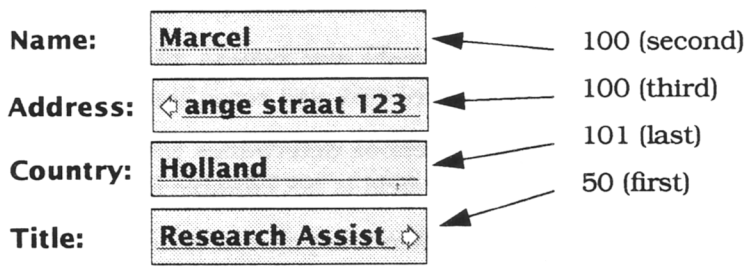




When editing text in a stack, typing **Control-T** forwards the input focus to next editable text object in the stack. You can also force the input focus to go to a particular text object by clicking on it. This applies to any object which can accept typed characters.

Figure 141

The focus number affects the order. The order is: Title, Name, Address, Country.



The object with the *smallest* focus number gets the initial focus, that is when the stack or card is opened. After that, the focus is forwarded to the text object with the next higher focus number.

If there is more than one text object with the same focus number, then the front to back ordering is used.

The default input focus number is 100. This number is chosen so that there are plenty of lower and higher input focus numbers. Leaving the input focus number field empty means that the object does not compete for the input focus. It means that you have to click on it to give it the input focus.

Text Properties

Text objects are used for multi-line text editing. They provide editing facilities that allow editing of text up to 65000 characters long.

Figure 142

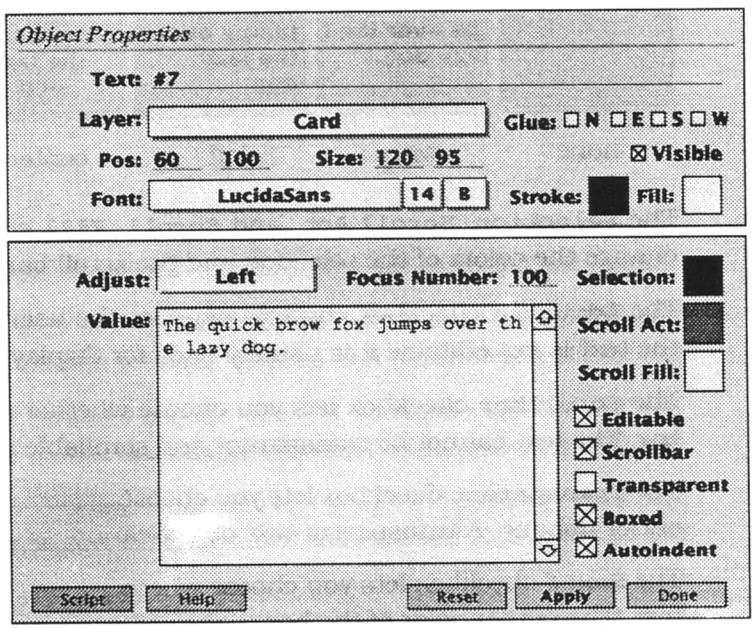
Some examples of text objects.



In this section we're concerned with text properties, not text editing. See "Editing Text" on page 3-14 to see how to edit text using keyboard short cuts. See "The Clipboard" on page 3-18 to see how to copy and paste text.

Figure 143

The properties of a Text object.



Text Properties Explained

We're only interested in the specific properties of text objects, so concentrate on the bottom half of Figure 143.

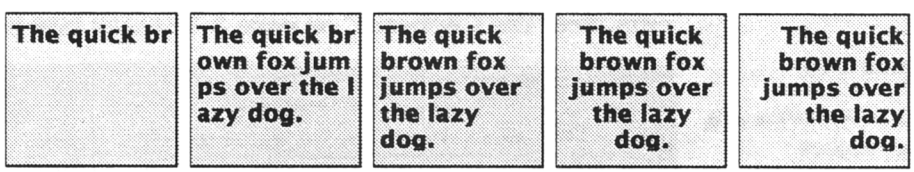
The **Value** field lets you enter the text for the text object. You can do this even if the text object is not editable.

The **Adjust** field lets you select the way the text is formatted. You can choose from:

- **None**. No formatting, lines are broken only where you type a newline.
- **Wrap**. Lines are broken at the last character that fits.
- **Left**. Lines are broken at word boundaries. The text is left justified.
- **Center**. Lines are broken at word boundaries. The text is centered.
- **Right**. Lines are broken at word boundaries. The text is right justified.

Figure 144

Different text adjustments.



The **Selection**, **Scroll Act** and **Scroll Fill** color selectors let you change the colors of the selection and the scroll bar.

The **Editable** checkbox controls whether the user can edit the text. If the text is not editable it is usually used for display only.

The **Scrollbar** checkbox lets you choose whether the text has a scrollbar. The text cannot be transparent and scrollable at the same time.

The **Transparent** checkbox lets you choose whether the text is transparent or opaque. A transparent text object cannot be editable.

The **Boxed** checkbox lets you choose whether a box is drawn around the text. The border color of the box is the same as the text color.

The **AutoIndent** checkbox lets you select whether the text is in *auto indent* mode. If the text is in *auto indent* mode, typing a newline will indent the new line the same as the previous line. This feature is useful for entering program text.

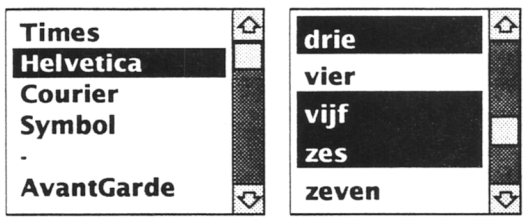
See "Controlling the Input Focus" on page 6-13 for how to use the **Focus Number** field to control the input focus.

List Properties

List objects are used to select items from a list. List items are useful if you want to present a long list of items, for example, fonts.

Figure 145

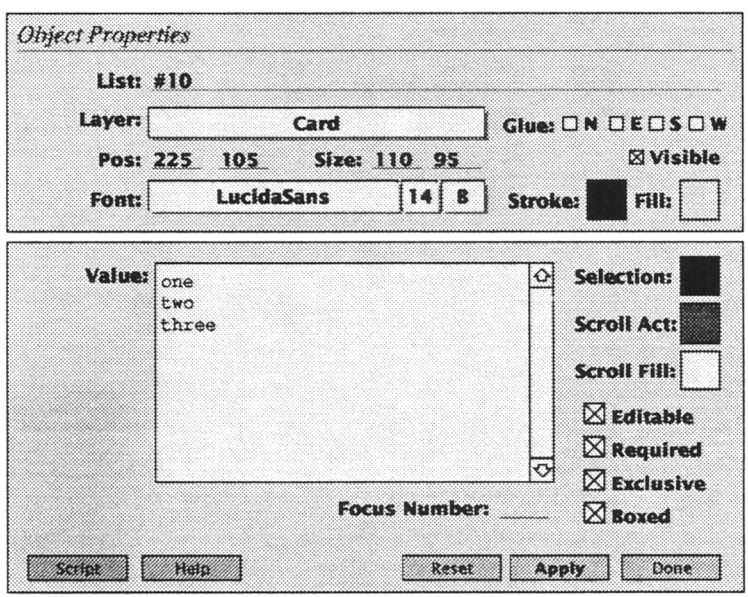
Some examples of List objects.



List objects are either exclusive or nonexclusive. An exclusive list object can only have one item selected at any time. A nonexclusive list object can have many items selected. See "Scrolling Lists" on page 3-16 on how to use List objects with keyboard short cuts.

Figure 146

The properties of a List object.



List Properties Explained

We're only interested in the specific properties of list objects, so concentrate on the bottom half of Figure 146.

Section "Generic Properties" on page 6-3 explains how to change the generic properties of a List object.

The **Value** field lets you specify the items in the list. You can specify any number of items separated by newlines.

The **Selection**, **Scroll Act** and **Scroll Fill** color selectors let you change the colors of the selection and the scroll bar.

The **Editable** checkbox determines whether the user can change the selection by clicking on items.

The **Required** checkbox indicates whether at least one item must be selected or not.

The **Exclusive** checkbox lets you choose whether the list is exclusive or nonexclusive. An exclusive list can have at most one item selected.

The **Boxed** checkbox lets you choose whether a box is drawn around the list. The border color of the box is the same color as the items.

See "Controlling the Input Focus" on page 6-13 on how to use the **Focus Number** field to control the input focus.

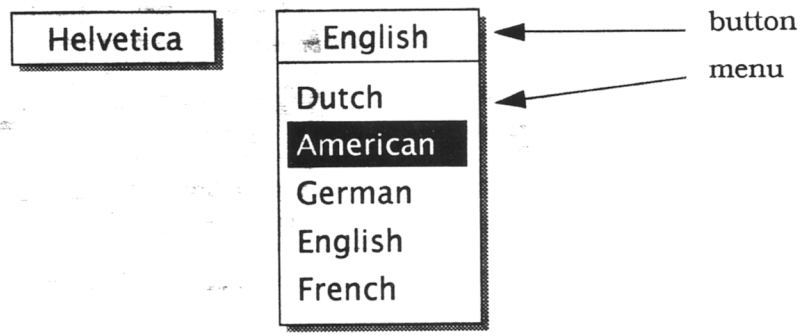


PullDown Properties

Pulldown objects are buttons with a pulldown menu. Pulldowns are displayed as a square button, usually with a shadow.

Figure 147

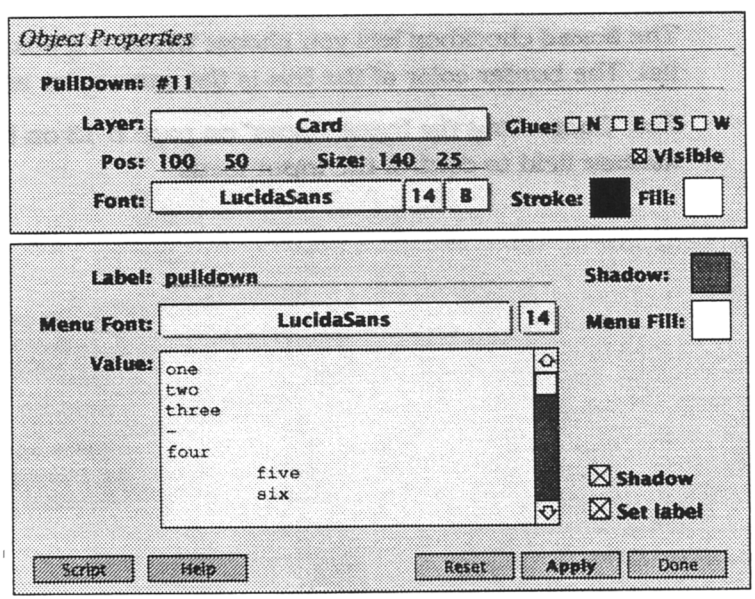
Some examples of Pulldown objects.



PullDown objects are used to select from a menu. A good example of a pulldown object is the font selector pulldown in the object properties stack.

Figure 148

The properties of a PullDown object.



PullDown Properties Explained

We're only interested in the specific properties of PullDown objects, so concentrate on the bottom half of Figure 148.

The **Value** field lets you specify the items in the pulldown menu. You can specify any number of items (one line per item). A single "–" item will put a horizontal line in the menu. See also "Pull Right Menus in PullDown Objects" later in this section.

The **Label** field lets you enter the text displayed in the pulldown button. The **Set Label** checkbox indicates whether the label of the pulldown button should change when a selection is made from the menu. If Set Label is true, the label will display the most recent selection.

The **Shadow**, and **Menu Fill** color selectors let you change the colors of the shadow and the menu.

The **Menu Font** pulldowns let you select the font which is used in the menu.

The **Shadow** checkbox determines whether the pulldown button has a shadow to distinguish it from ordinary square buttons.

Pull Right Menus in PullDown Objects

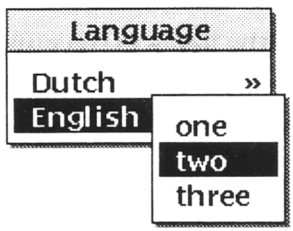
You can specify pull right menus for PullDown objects by indenting the items in the **Value** field accordingly. Items which are indented by equal amounts are assumed to be part of the same menu. For example:

```
Dutch
   een
   twee
   drie
English
   one
   two
   three
```

This results in three menus. The top level pulldown menu contains two items, **Dutch** and **English**. Each item has a pull right menu with three items.

Figure 149

An example of a pull right menu in a PullDown object.

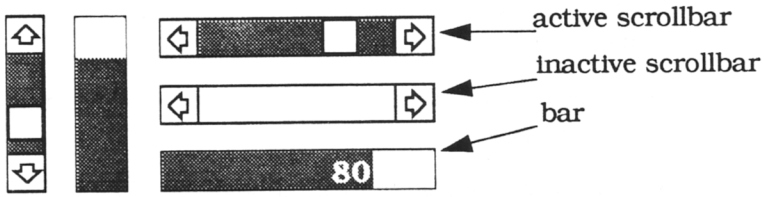


Slider Properties

Slider objects let you select a numeric value. Sliders come in two flavors: *bar* and *scrollbar*. Bar sliders are usually used to select a value from a range. Scrollbars are used to scroll data such as text or graphics.

Figure 150

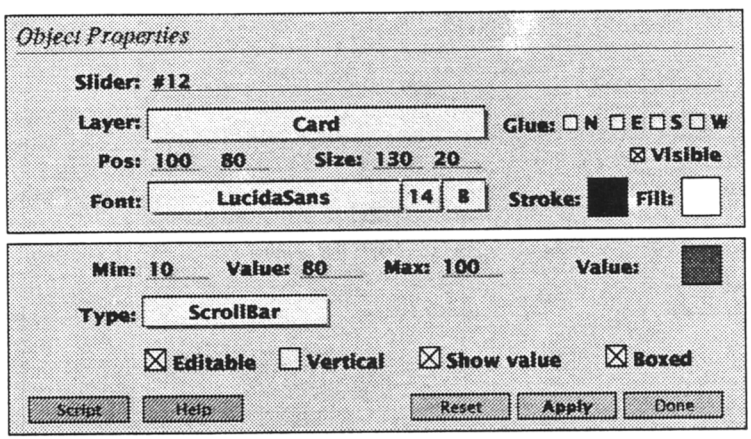
Some examples of Slider objects.



See "Moving Around" on page 4-8 to see how to use scrollbars.

Figure 151

The properties of a Slider object.



Slider Properties Explained

We're only interested in the specific properties of slider objects, so concentrate on the bottom half of Figure 151.

The **Min**, **Max**, and **Value** numeric fields let you specify the minimum, maximum and current value of the slider itself. If the minimum value is greater or equal to the maximum, or if the current value is outside the legal range, the slider is made inactive.

The **Value** color selector lets you change the color of the bar or scrollbar.

The **Type** pulldown lets you select the type of slider, either **Bar** or **ScrollBar**.

The **Editable** checkbox determines whether the user can change the value of the slider.

The **Vertical** checkbox indicates whether the slider moves vertically or horizontally.

The **Show Value** checkbox lets you choose whether a **Bar** slider shows its current value.

The **Boxed** checkbox lets you choose whether a box is drawn around the Slider.

ColorSelect Properties

ColorSelect objects let you select a color from a popup color pallet. The currently selected color is displayed in the color rectangle. See "The Color Pallet" on page 3-20 on how to change the colors on the popup color pallet.

Figure 152

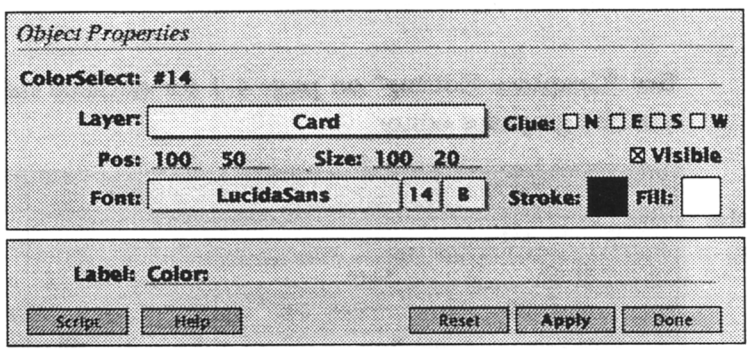
Some examples of ColorSelect objects.



An example of how to use a popup color pallet is described in "Color Selectors" on page 3-17.

Figure 153

The properties of a ColorSelect object.



ColorSelect Properties Explained

We're only interested in the specific properties of ColorSelect objects, so concentrate on the bottom half of Figure 153.

ColorSelect objects are really very simple. All you can change is the **Label**. The size of the color rectangle is controlled by changing the size of the object.

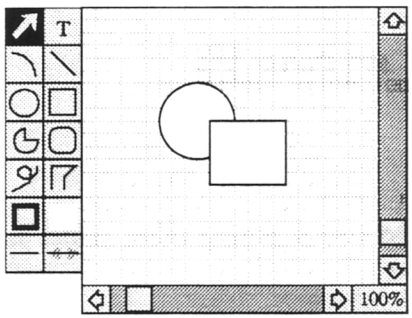
To change the colors appearing in the popup menu, use the ColorPallet stack and modify the specific entries as described in "The Color Pallet" on page 3-20.

DrawTool Properties

The DrawTool object is a complete graphics editor in the shape of a HyperLook object. It lets you incorporate the graphics editing features in your own application.

Figure 154

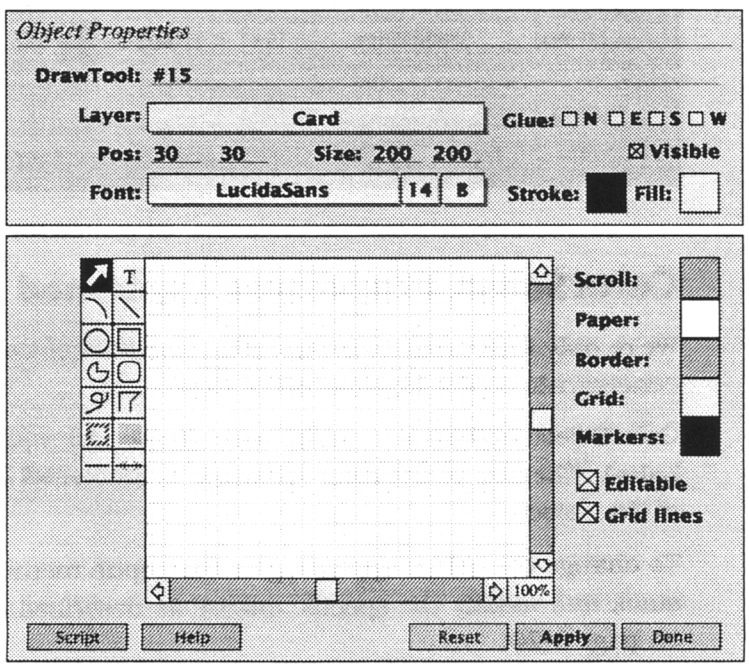
An example of a DrawTool object.



See "Graphics Editing" on page 4-1 for detailed information on how to use the graphics editor.

Figure 155

The properties of a DrawTool object.



DrawTool Properties Explained

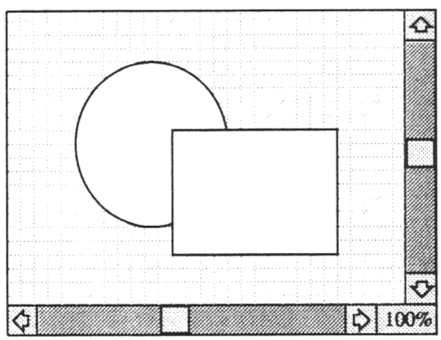
We're only interested in the specific properties of DrawTool objects, so concentrate on the bottom half of Figure 155.

The graphics editor on the properties stack lets you edit the drawing of the DrawTool object. You can do this even if the object is not editable.

The **Editable** checkbox determines whether the user can change the drawing in the DrawTool object. A non editable DrawTool object does not have the tool pallet and its menu is restricted.

Figure 156

An example of a *non* editable DrawTool object.



The **GridLines** checkbox lets you choose whether the grid lines are drawn or not.

The **Scroll** color selector lets you choose the color of the scrollbars.

The **Paper**, **Grid** and **Border** color selectors let you choose the color of the paper, the grid lines, and the borders of the paper.

The **Markers** color selector determines the color of the markers used to indicate the selected object.

7

Scripting

This chapter explains how to program HyperLook objects. Programming a HyperLook object is called scripting. The language that is used is PostScript.

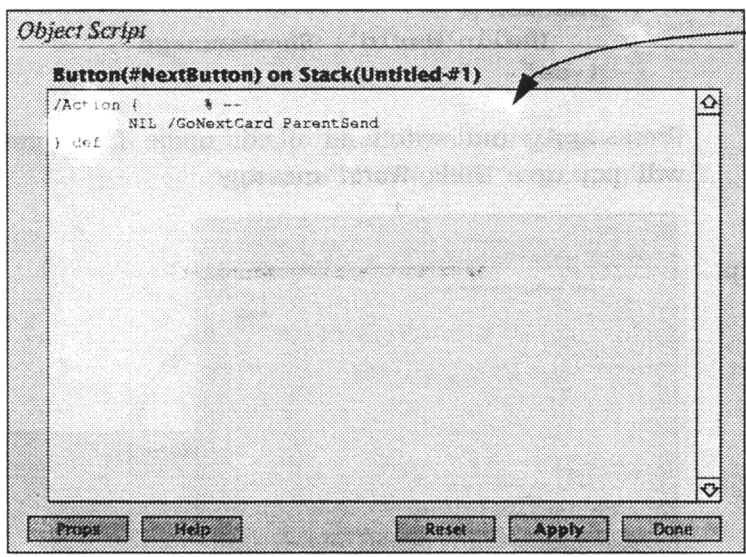
Writing scripts is useful if you want to define bits of program which are executed when, for example, a button is pressed. But you are not restricted to simple programs alone, you can redefine most behaviors of an object by writing scripts. You can even create new classes of object.

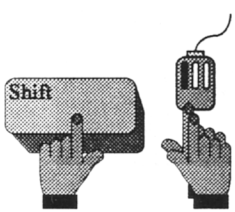
Editing Scripts

To edit the script of an object, switch the stack to edit mode and select the object. Now select **Script** from the **Object** menu. The script property stack is shown.

Figure 157

The Script properties of a button.





The script of an object can also be shown by double clicking the object while holding down the **Shift** key.

The scripts properties of Stacks, BackGrounds, Cards and objects are shown by pressing the **Script** button on the appropriate property stack.

When you are finished editing the script, press the **Apply** button. This evaluates the script and assigns it to the object. You can throw away any changes that you have made by pressing **Reset**. Press **Done** when you are finished.

Pressing the **Props** button shows the object properties of the object. These are explained in Chapter 5, "Editing Stacks" and Chapter 6, "Object Properties". The **Help** button is explained in "Defining Help" on page 5-29.

An Example of a Script

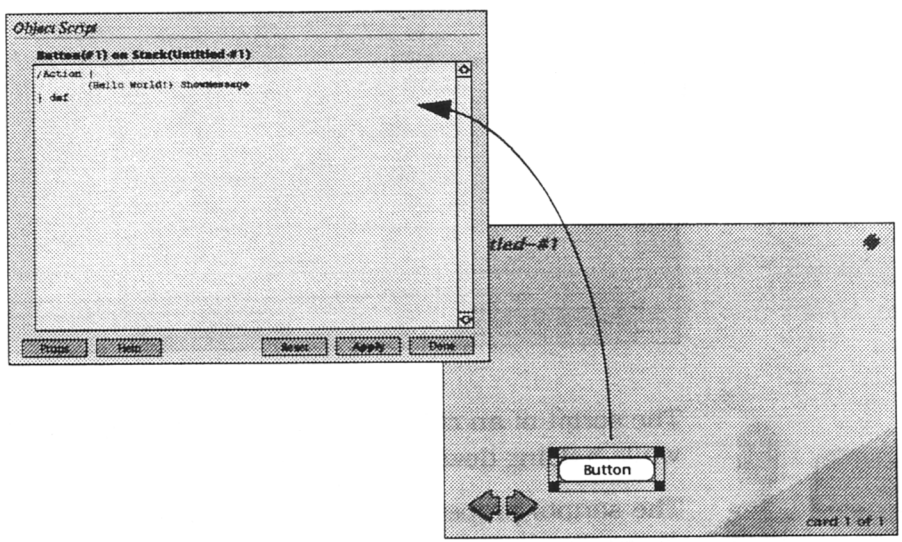
Now try writing your own script. Create a new button and get its script properties. Enter the following script (see "Editing Text" on page 3-14 for some useful editing hints):

```
/Action {
   (Hello World!) ShowMessage
} def
```

Press **Apply** and switch out of edit mode. If you now press the button it will pop up a "*Hello World*" message.

Figure 158

Editing the script of a button.



A script should contain only PostScript variable and function definitions. It should not contain code which gets executed when the script is applied. See "The OnOpen and OnClose Methods" on page 7-25 on how to perform initializations.

This is because when a stack is loaded from disk for the first time, its script evaluated in a strange environment. Not all operations are guaranteed to work in that state. Please read on, later on in this chapter more information about writing scripts is given. And there are lots of examples too!

The PostScript Language

The programming language used in scripts is called PostScript. If you want to learn more about PostScript, read the Adobe PostScript manuals and the NeWS® manuals (see the end of this section for details)[1].

It is beyond the scope of this manual to give a comprehensive introduction to the PostScript language. It assumes that you are familiar with PostScript.

But don't panic if you don't know PostScript. Just read on and copy the examples that are given. First, a little bit of background on the language itself.

Postfix Notation

The most striking aspect of the PostScript language is its notation. PostScript uses postfix notation. That means that the arguments of an operator come *before* the operator name. For example, C uses *infix* notation:

```
1 + 2
```

In PostScript this is written differently:

```
1 2 add
```

Postfix notation is not hard to learn. You have to get used to it, but once you understand the principles behind it, it's very easy.

Stack Operations

Most PostScript functions use the *operand stack*. This is a stack of things like temporary values and arguments to function. Take the following expression:

```
6 - (2 + 3)
```

1. NeWS is a registered trademark of Sun Microsystems Incorporated.



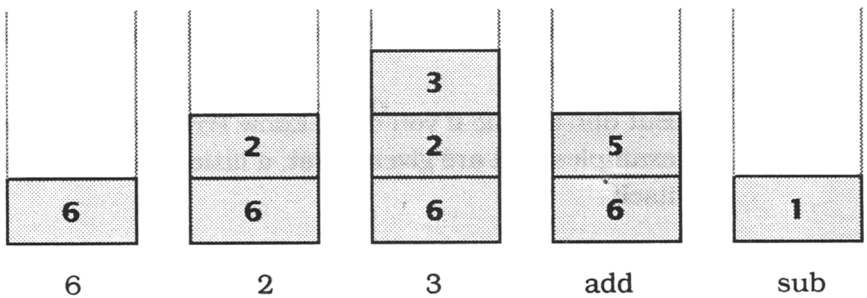
The PostScript equivalent of this expression is as follows:

```
6 2 3 add sub
```

To understand this you could imagine the numbers being pushed on the stack as shown in Figure 159.

Figure 159

The PostScript operand stack.



You never need to use parenthesis to group sub expressions in PostScript. All PostScript expressions are unambiguous (even without parenthesis).

PostScript provides lots of operators which let you manipulate the items on the stack. For example, **pop** discards the top item of the stack, **dup** duplicates the top item of the stack, and **exch** swaps the top two items of the stack.

Notation

A brief note on the notation of variables and functions is useful because you will see it used a lot in examples that follow.

A PostScript function is defined as follows (note that the '**%**' sign starts a comment):

```
/functionname {    % arguments -- results
   ... Postscript Code ...
} def
```

In the case of multiple arguments or results, they are specified in bottom to top order as they occur on the operand stack. A PostScript variable is defined as follows:

```
/variablename initialvalue def
```

More Information on PostScript

There are four sources of information (apart from this manual) that you may find useful when writing HyperLook scripts.

- **The PostScript Language Reference Manual**[1]. It explains the PostScript language and all the PostScript operators. The PostScript interpreter used by OpenWindows is almost 100% compatible with Adobe PostScript (level 1).
- **The NeWS programmer's manual**. This manual lists the differences between Adobe PostScript and the PostScript used by OpenWindows. It also describes the many enhancements and utilities included in to support interactive programming.
- **The Hyperlook documentation browser**. This on-line tool lets you access the documentation on all the PostScript functions that HyperLook provides (see "The Documentation Browser" on page 7-10).

1. ISBN: 0-201-10174-2

Classes

A *class* defines the behavior of an object. All objects in the HyperLook system are implemented using classes. HyperLook is an *object oriented* system.

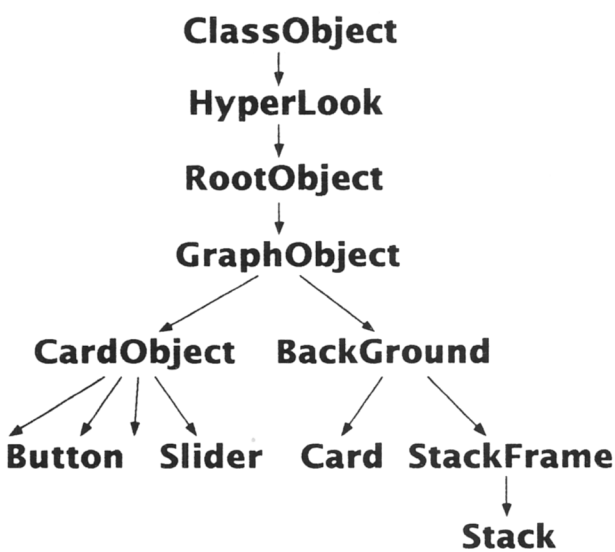
A class contains *variables* and *methods*. Methods are functions which are defined in the context of the class. A method can always access other methods and variables of the class.

An *instance* is an actual object created from the class. Take for example the system stack, which contains multiple instances of the Button class. A script defines methods and variables that apply only to a single instance.

Each class is part of the class hierarchy. Figure 160 shows part of the HyperLook class hierarchy. The class hierarchy defines how methods and variables are inherited.

Figure 160

Part of the HyperLook class tree.



A button, for example, inherits its variables and methods from the classes **`ClassObject`**, **`HyperLook`**, **`RootObject`**, **`GraphObject`**, **`CardObject`** and **`Button`**.

Super Classes and Sub Classes

The parent of a class is called a *super* class. For example, class **CardObject** is a super class of class **Button**.

The child of a class is called a *sub* class. For example, class **BackGround** has two sub classes: class **Card** and class **StackFrame**.

A super class's methods are inherited by its sub classes. This means that if class **CardObject** has a method called **Draw**, it will be inherited by all of its sub classes. Therefore class **Button** and class **Slider** therefore both have a **Draw** method

The methods defined by a super class can be redefined. For example, class **Button** can redefine methods which are defined in class **CardObject**. This doesn't change the methods in class CardObject, it just overrides it for instances of class Button. Scripts often redefine class methods in a particular instance.

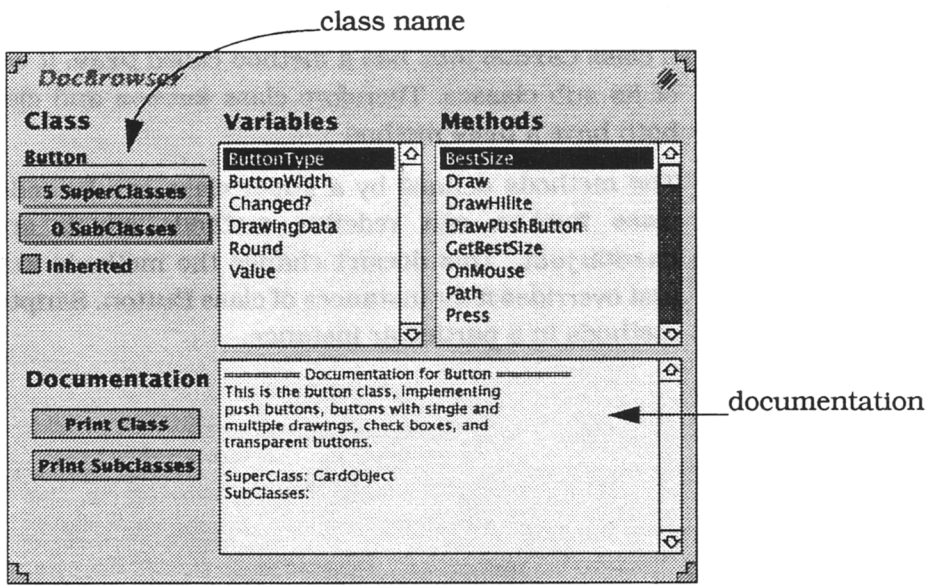


The Documentation Browser

HyperLook provides a tool which lets you examine the class hierarchy and the methods and variables of the HyperLook classes. It is called the documentation browser. Select **DocBrowser** from the **Tools** menu in the system stack to show the documentation browser.

Figure 161

The documentation browser.



The documentation browser lets you examine the documentation of methods and variables of HyperLook classes. Every HyperLook class method and variable has some documentation describing briefly what it does.

Using the Documentation Browser

To view the documentation on a class you must enter the name of the class in the **Class** field and type return. The methods and variables of the class are then displayed in the **Variables** and **Methods** lists.

The **Inherited** checkbox determines which methods and variables are shown. If the checkbox is checked, all methods and variables that are inherited from the super classes are displayed. If it is not checked, only those methods and variables which are defined by the class itself are displayed.

You can also select classes from the **SuperClasses** and **SubClasses** pulldown items. The **SuperClasses** pulldown contains all the super classes of the current class. The **SubClasses** pulldown contains all the sub classes.

Printing Class Documentation

You can print the class documentation by pressing **Print Class** ("Specifying a Printer" on page 3-32). This prints an alphabetic listing of the variables and methods of the current class. There are many pages of source documentation associated with each class, the printouts can take up a lot of paper!

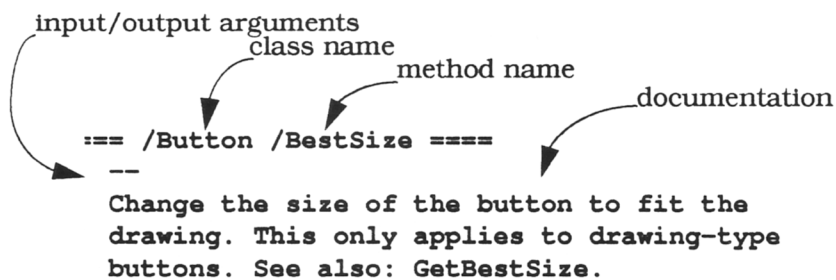


Figure 162

An example of some documentation for the method **BestSize** of class **Button**.

The header sometimes contains the words *persistent* or *instance*, these indicate the type of methods or variable. Don't worry about them, they are only significant for advanced programmers.

Pressing **Print SubClasses** prints documentation for the current class and all its subclasses. The **Inherited** checkbox is turned off to save trees.

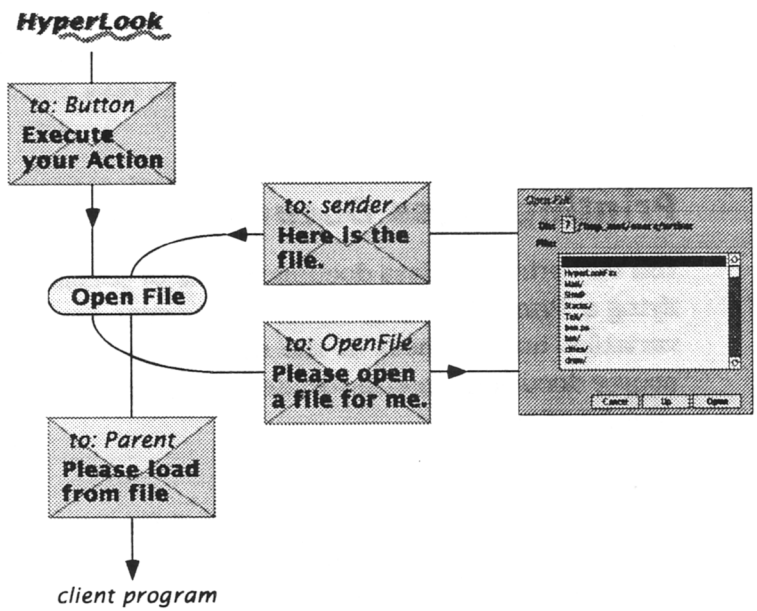
To print the class documentation of *all* HyperLook classes, select the **HyperLook** class and press **Print SubClasses**.

Messages

HyperLook uses message passing to communicate between objects and between objects and clients. Figure 163 shows the flow of messages when a button is pressed to open a file.

Figure 163

Communication between a Button, a stack and a client.



The HyperLook system sends a message to the **`Open File`** button when it is pressed. The button responds by asking the OpenFile stack to open a file, and the resulting file name is send back to the button. The button then passes the message onto its parent.

Messages and Methods

Every message has a name. For example, the HyperLook system sends a **`Draw`** message to a button when it needs updating. The button has a *method* called **`Draw`** which draws the stack.

The methods of an object are defined in its *class* or in its script. You can define new methods by editing the script of the object.

Most programming languages let you write functions which can be accessed from anywhere, they are defined globally. Object oriented systems let you define functions which are known only in the scope of an object.

This not only means that objects can have different methods with the same names, but also that a message can mean different things to different objects. For example, the **Draw** method of a button draws a button while the **Draw** method of a slider draws a slider.

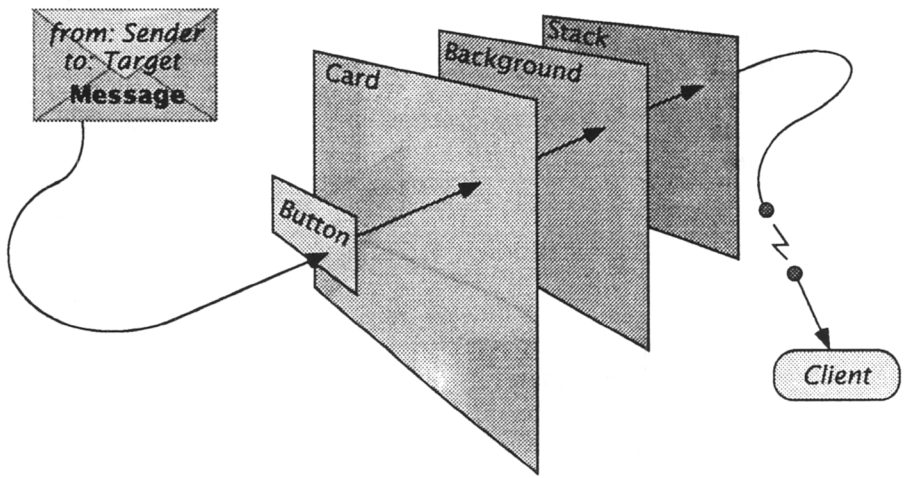
Sending a message to an object really means calling a function in the *context* of the object. The *context* of an object is defined by its class[1].

The Message Hierarchy

What happens when a message is sent to an object and the object does not define the method? In some systems this would cause an error! Not in HyperLook.

Figure 164

The HyperLook message hierarchy.



When a message is not defined by an object, it is passed on to the *parent* of the object. If the parent of the object does not define the message, it is sent on to its parent, and so on. Eventually the end of the chain is reached and the message is either send to a client program or it is lost.

1. The *class* of an object and the *type* of an object really mean the same thing.

This path is traversed to find the receiver of a message:

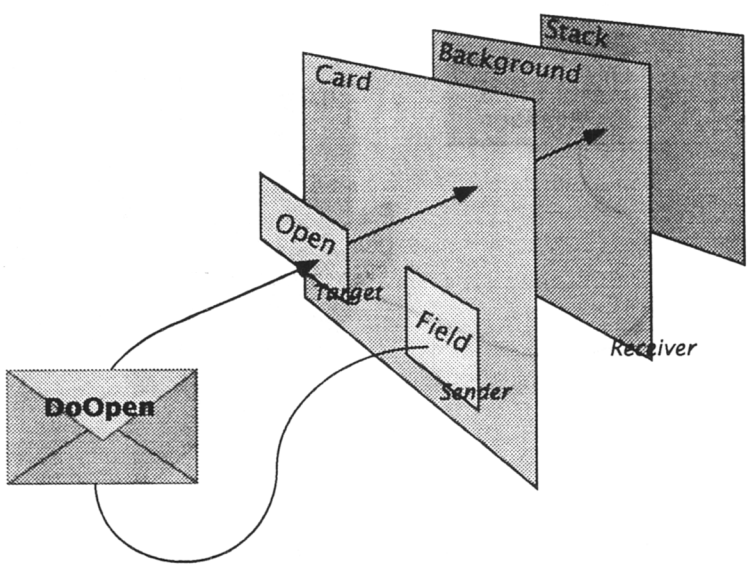
1. Component (Button, Slider, etc.)
2. Card
3. Background
4. Stack
5. Client (only when connected)

The message is delivered to the first object in the chain that understands the message. If the message is not delivered, it is discarded, or it is sent to the client. You can also send messages directly to a card, background, stack or client.

A message is sent from the **Sender** to the **Target**. The **Target** is the object which was identified by the sender as the initial target of the message. If the target can't handle the message, it is passed on to its parent until it's handled by the **Receiver**.

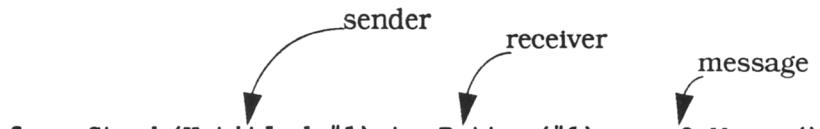
Figure 165

A message is send by the **Sender**, it is aimed at the **Target** and it is handled by the **Receiver**.



Finding Out About Messages

It's possible to print all messages on the console. See "Logging Messages" on page 3-33 to find out how to switch this feature on. As you use the system, you will see all the messages being generated.




```
from Stack(Untitled-#1) to Button(#1) --> OnMouse()
from Button(#1) to Button(#1) --> Action(0)
from Button(#1) to Stack(Message) --> Init((Hello
world!))
```

Figure 166

Part of the message log that is printed on the console when a button is pressed.

Writing Scripts

HyperLook objects know about certain messages. In this section examples are given of how to use the most common ones.

See also Appendix B, "Methods" for an overview of the most important HyperLook methods which are discussed in this chapter.

The Action Method

```
/Action        % value --
```

The **Action** method is called when the *action* of the object should be performed. This always happens after the user has changed the state of the object. It happens, for example, when a button is pressed or when the value of a slider is changed

The most common use of the **Action** method is to define the action of a button. Try this out by creating a button and giving it the following script[1]:

```
/Action {      % number -- number(ignored)
   beep
} def
```

When you press the button, the **Action** is executed and the **beep** operator causes the keyboard to beep.

Here is an example of how to use the **Action** message of a slider. Create a slider and give it the following script:

```
/Action {      % nr --
   50 gt {
        (More than 50) ShowMessage
   } {
        (Less than or equal to 50) ShowMessage
   } ifelse
} def
```

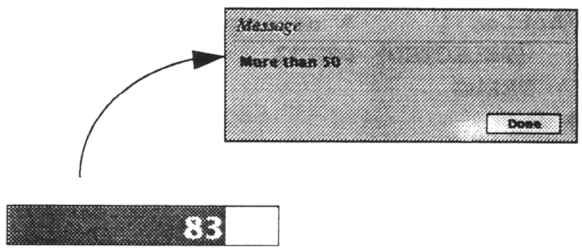
1. The arguments and results of the **Action** method can be ignored safely.



The **ShowMessage** function shows a pop up message window containing the argument string. When you have applied the script try, changing the value of the slider.

Figure 167

A slider in action.



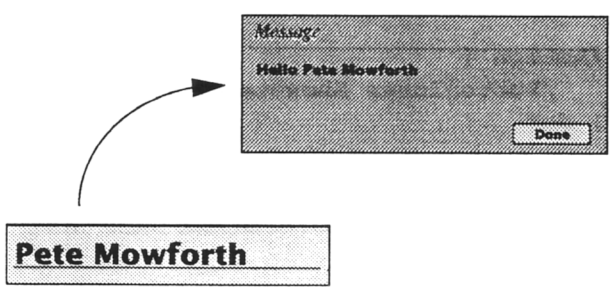
The **Action** message of text and field objects is called with a string as an argument. Here is an example of an **Action** method for a field object:

```
/Action {      % string --
   (Hello ) exch append ShowMessage
} def
```

When you type a string into the field and hit return, the **Action** method is called, with the string that you typed as an argument, so a message is shown.

Figure 168

A field in action.



The following script can be used for any object. Try assigning it to a PullDown object. It uses the **DEBUG** function to print a debugging message on the console.

```
/Action {      % string --
   (SELECTED) DEBUG
   DEBUG
} def
```

When you select an item from the pulldown menu a debug message is printed on the console:

```
DEBUG: SELECTED
DEBUG: two
```

You can use the **DEBUG** function whenever you want to print debugging information.

Manipulating Stacks

Now that you know how to program a button action, it is easy to show a stack when a button is clicked. Let's say you want to show the ButtonIdeas stack when button is clicked:

```
/Action {
   /ButtonIdeas ShowStack
} def
```

The **ShowStack** function loads a stack (if it is not already loaded) and shows it. You can hide the ButtonIdeas stack using the **HideStack** function:

```
/Action {
   /ButtonIdeas HideStack
} def
```

If you want to be really adventurous, you can hide the current stack. The variable **MyStack** returns the current stack:

```
/Action {
   MyStack HideStack
} def
```




See "Opening Stacks" on page 3-8 on how to open your stack again.

Sending Messages

The **Send** operator is used to send a message to an object. Sending messages is explained in more detail in "Messages" on page 7-12. It takes the following arguments:

```
/Send          % arg-array message target --
```

Create a button and give it the following script:

```
/Action {      % number -- number(ignored)
   [] /IconifyStack Self Send
} def
```

When you press the button it will send a message called **IconifyStack** to itself. Because a button does not know how to handle it, the message is passed on to its parent. The stack is iconified because the message is handled by the stack.

Messages can be sent directly to the stack or client by using the **MyStack** or **MyClient** variables as a target. There are various other ways in which messages can be sent:

```
[] /Message Self Send     % to the object itself
[] /Message MyStack Send  % directly to the stack
[] /Message ParentSend    % to the parent of the object
[] /Message ClientSend    % directly to the client
[] /Message /name Send    % to a named object
```

The target can be specified as a stack name or object name. Here is an example of a button script that iconifies the system stack.

```
/Action {
   [] /IconifyStack /system Send
} def
```

You can also send message between objects on the same stack. Create a slider and call it **Foo**. A button on the same stack can now send a message to the slider as follows:

```
/Action {
   [50] /SetValue /Foo Send
} def
```

This sends a **SetValue** message to the slider called **Foo**. The message has one argument, the number **50**.

Addressing Objects

Every stack name in the HyperLook system must be unique. Every object on a stack must have a name which is unique within that stack. This makes the addressing of objects very easy[1]:

```
/system FindStack DEBUG
DEBUG: instance(class(Stack))
```

Finds the stack named **system**.

The **FindObject** function finds an object on a stack. It takes a stack name and an object name as an argument. It provides you with an unambiguous way to address an object:

```
/system /Exit FindObject DEBUG
DEBUG: instance(class(Button))
```

This finds the object called **Exit** on the stack called **system**.

The **Send** primitive uses the **LookupObject** function to address objects. This function first tries to find the object on the current stack. If that fails it assumes that you are addressing a stack. For example:

```
/system LookupObject DEBUG
DEBUG: instance(class(Stack))
```

1. The output of **DEBUG** will appear on the console.

In this case it returned a stack. Addressing an object with **LookupObject** is also possible. For example:

```
/CardNr LookupObject DEBUG
DEBUG: instance(class(Field))
```

Here is a button script that address an object unambiguously using **FindObject**:

```
/Action {
   [] /Press /system /Exit FindObject Send
} def
```

The **Press** method only applies to buttons. It highlights the button and performs the button action just as if the user had pressed it.

Linking Buttons to Cards

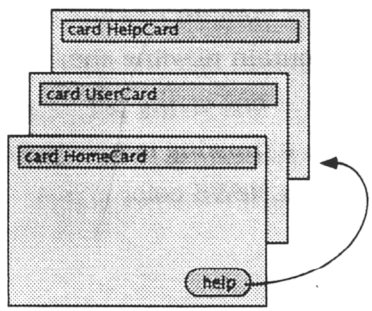
Now that you know how to send messages and how to address objects, it is easy to link a button to a card. Here is the script of a "next card" button. When the button is pressed the next card is shown:

```
/Action {
   [] /GoNextCard ParentSend
} def
```

The following button script is an example of how a button can go to a particular card of a stack. Create a new card and call it **HelpCard**.

Figure 169

A button can be linked to any card of the stack.



You can now create a button (on another card) that shows the card named **HelpCard**.

```
/Action {
   [/HelpCard] /GotoCard ParentSend
} def
```

You can also open a stack at a particular card. This is done by first sending a **GotoCard** message to the stack and then showing the stack using **ShowStack**.

To try this out, assign the following script to a button. It will open the **ButtonIdeas** stack at card number 2 (make sure the stack is hidden first).

```
/Action {
   [2] /GotoCard /ButtonIdeas Send
   /ButtonIdeas ShowStack
} def
```

The SetValue Method

Every object has a **SetValue** method. The value of the object depends on its type. The value of a text object is a string, while the value of a slider is a number. Here a a list of the value types for the HyperLook objects:

- **Button** -- *integer* -- the state of the button.
- **Slider** -- *integer* -- the value of the slider.
- **Field** -- *string* -- doesn't contain newline characters.
- **Text** -- *string* -- may contain newline characters.
- **List** -- *array-of-strings* -- one string per item.
- **DrawTool** -- *drawing* -- a drawing is a dictionary object.
- **ColorSelect** -- *color* -- a NeWS color object.



This button script sets the value of a text object called **MyText**.

```
/Action {
   [(Hi There,\nhave a nice day!)]
   /SetValue /MyText Send
} def
```

Note that the **\n** in the string inserts a newline in the text (see the Adobe PostScript manual).

Below is an example of a script for a slider. Every time the slider is changed by the user, it sends a **SetValue** message to a slider called **MyBrother**, keeping it up to date with the same value:

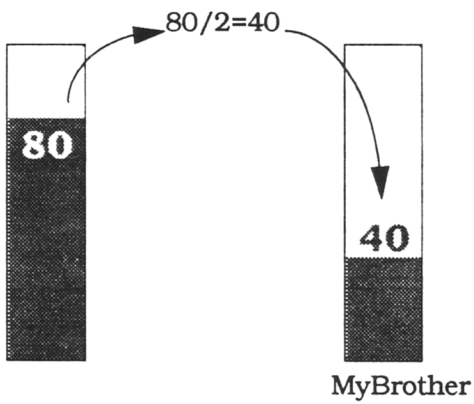
```
/Action {  % nr --
   [exch] /SetValue /MyBrother Send
} def
```

Alternatively you can change the value before setting the other slider. In this example the value is halved:

```
/Action {  % nr --
   [exch 2 idiv] /SetValue /MyBrother Send
} def
```

Figure 170

The value of the right slider is half the value of the left slider.



Getting the Value of an Object

You may have noticed that the **Send** operator does not return values. That means that you can't use it to access the value of an object. You must use the **send**[1] operator to access the value of an object's variable.

The definition of the send operator is as follows:

```
/send      % ...args message object -- results...
```

Note that the arguments are not in an array but simply on the stack. Also notice that the **send** operator can return results.

In the following example a slider accesses the value of another slider in the **Action** method:

```
/Action {
   /Value /MyBrother LookupObject send SetValue
} def
```

It first uses the **LookupObject** function to address the other slider before it gets the value of the slider using **send**. It then uses the **SetValue** method to set its own value to be equal to the value of the other slider.

The result is that the slider does not let the user change its value. It always resets itself to the value of the other slider.

The OnHelp Method



Section "Defining Help" on page 5-29 explains how to assign help text to an object. It is also possible to perform an action when the user presses the Help key over an object.

When this happens, an **OnHelp** message is sent to the appropriate object. This object can respond with any action it likes. For example:

```
/OnHelp {      % --
   (Sorry, no help today...) ShowMessage
} def
```

1. Notice the difference in capitalization between **Send** and **send**.

Pressing the **Help** key over an object with this script will show the message stack instead of the default help stack.

The Show and Hide Methods

Sending a **Hide** message to an object makes it invisible. Sending a **Show** message shows it again. To illustrate this, you need to create a check box and another object of arbitrary type called **Foo**.

Give the checkbox the following script:

```
/Action {       % state --
   0 eq {
       [] /Hide /Foo Send
   } {
       [] /Show /Foo Send
   } ifelse
} def
```

Switching the checkbox on shows the **Foo** object, and switching the checkbox off hides the **Foo** object.

The OnOpen and OnClose Methods

The **OnOpen** method of an object is called when the object first becomes visible. The **OnClose** method of an object is called when the object is removed from the screen.

You can use the **OnOpen** method to perform some initialization that may be needed. It can be used, for example, to give the object an initial value.

The following script for a field object sets the initial value of the field to today's date, this can be useful in database applications:

```
/OnOpen {       % --
   SystemDate SetValue
} def
```

The **SystemDate** operator returns a string containing today's date. The use of the OnClose methods is illustrated in the following script:

```
/OnClose {     % --
   (I'm leaving now!) ShowMessage
} def
```

You can use it for any object. It shows a message as soon as the object disappears of the screen.

Defining Your Own Methods

So far only system defined methods were discussed. You are not restricted to using these alone. You can add new methods to objects yourself.

Here is a script for a slider object that defines a method called **SetRootValue** for the object:

```
/SetRootValue {    % nr --
   sqrt SetValue
} def
```

The new method sets the value of the slider to the square root of a number. You can invoke this methods by sending a **SetRootValue** message to the object. For example, a button script may contain the following if the slider is called **Foo**:

```
/Action {      % --
   [25] /SetRootValue /Foo Send
} def
```

Using IncludeScript

You may find that your script gets too long to edit efficiently with the script properties editor. You can store the script in a file which you can edit with your favorite editor.



To do this, you must create a file containing the script in a resource directory. Usually this would be the same directory as the one where you save the stack containing the object. Now put the following in the script of the object:

```
(myfile.script) IncludeScript
```

This will read the object's script from the file **myfile.script** when you press the **Apply** button on the script property stack. You must apply the script every time you've made changes to the file.

Remember to keep the object and the script file together. The script file must be accessible when the object is loaded. The object will be loaded incorrectly if the script cannot be found.



Scripting Examples

This section contains a simple example and two more advanced examples.

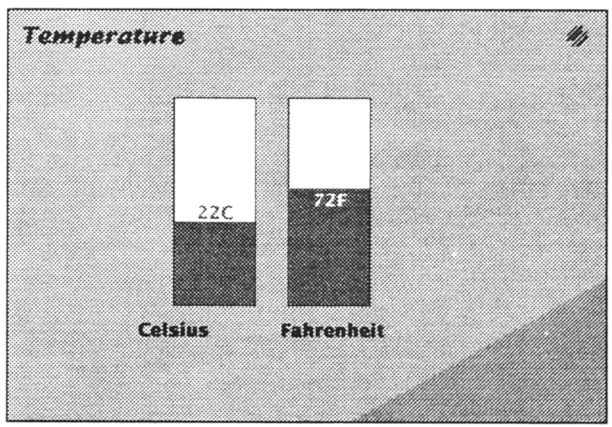
Fahrenheit and Celsius

In this example two sliders are programmed to represent temperatures in Fahrenheit and Celsius. When either of the sliders is changed the other slider's value is updated to the corresponding value.

For this example you need to create a stack containing two vertical bar sliders. Name the one on the left **Celsius**, the one on the right **Fahrenheit**. Set the range of the **Celsius** slider to be **-100** to **100**. The range of the **Fahrenheit** slider should be the same.

Figure 171

The Temperature example stack.



Assign the following script to the **Celsius** slider:

```
/ConvertValue {% val -- str
   (%C) sprintf
} def

/Action {% val --
   [exch 1.8 mul 32 add] /SetValue /Fahrenheit Send
} def
```

The **ConvertValue** method lets you modify the string that is displayed in the slider. The **sprintf**[1] operator is used to convert the integer value to a string and append a **C** for Celsius. The action method sets the value of the **Fahrenheit** slider to **C*1.8+32**.

The script of the **Fahrenheit** slider is very similar. It sets the value of the **Celsius** slider to **(F-32)/1.8**:

```
/ConvertValue {% val -- str
   (%F) sprintf
} def

/Action {% val --
   [exch 32 sub 1.8 div] /SetValue /Celsius Send
} def
```

You can now modify one slider, and the other slider will be automatically updated to the corresponding value.

Let's make a small modification to the **SetValue** method so that the updating of the sliders happens immediately instead of when you release the mouse. Add this definition of **SetValue** to both scripts.

```
/SetValue {% val --
   round dup Value ne {
       /SetValue super send
       [Value] /Action Self Send
   } {pop} ifelse
} def
```

The **SetValue** method is called continuously when you are changing the value of the slider. It now sends an **Action** method to itself to update the other slider every time its own value changes.

1. See the NeWS programmer's manual.



Let's make one last change to **SetValue** method of the sliders, to make them change color when the temperature is below freezing. Change the SetValue method in the script of the **Celsius** slider to:

```
/Freezing  0 def
/Red       1 .5 .5 rgbcolor def
/Blue      .5 .6 1 rgbcolor def

/SetValue {% val --
   round dup Value ne {
       Value Freezing gt {Red} {Blue} ifelse
       /ValueColor exch def
       [Value] /Action Self Send
   } {pop} ifelse
} def
```

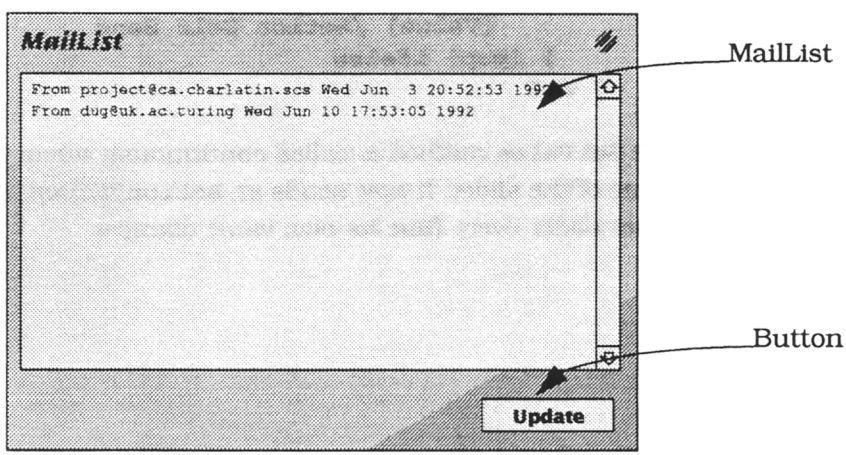
Change the script of the **Fahrenheit** slider accordingly (you need to change the **Freezing** variable to **32** instead of **0**).

Listing Mail Messages

In this more advanced example, a button is programmed to read the output produced by the Unix program called **from**. This standard Unix program lists the messages in your mail box.

Figure 172

The MailList example stack.



For this example you need to create a stack containing a button labelled **Update** and a list object called **MailList**.

Now give the **Update** button the following (rather horrible) script:

```
/Action {
   [(from) pipe pop {
       dup 256 string readline {exch} {
           pop pop exit
       } ifelse
   } loop]
   [exch] /SetValue /MailList Send
} def
```

The script uses the **pipe** operator to fork a **from** process. It then proceeds by reading one line of output until the end of file is reached. The resulting strings are left on the stack to form an array of messages which is then assigned to the **MailList** object using the **SetValue** method.

If you press the **Update** button the list of incoming mail messages is updated from the output of the **from** program.

Animated Slider

In this advanced example the **OnOpen** and **OnClose** methods are used to start and stop a timer which is called at regular intervals to increment the value of a slider. Create a bar slider and give it the following script:

```
/Timer null def
/Delay ScrollThresh def
/OnOpen {      % --
   /Timer {
       Value 1 add MaxValue mod SetValue
   } Delay StartTimer def
} def
/OnClose {     % --
   Timer StopTimer
   /Timer null def
} def
```

The **ScrollThresh** variable used to initialize the **Delay** variable. It is done this way because the time format varies between versions of OpenWindows.



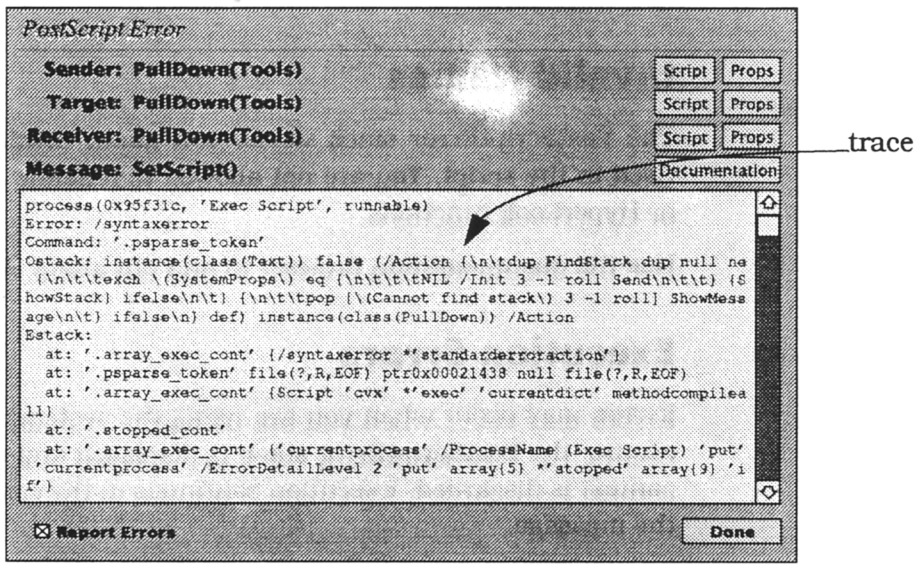
In the OnOpen method, a timer is started using **`StartTimer`**, which increments the value of the slider periodically. The **`OnClose`** method stops the timer.

Dealing with Errors

When the script is applied to the object, it is evaluated by the PostScript interpreter. If an error occurs when the script is applied, the PostScript error stack is shown (see also "Report Errors" on page 3-33).

Figure 173

The PostScriptError stack.



Pay attention to the trace when an error occurs[1]. It tells you the name of error and the name of the operator that caused the error. The **Receiver** field indicates in which context the error occurred.

Before you can apply the script correctly, the error must be fixed. So press **Done** on the PostScript error stack and edit the script until the problem is solved.

The properties or the script of either the **Sender**, the **Target** or the **Receiver** of the message can be shown by pressing the corresponding **Props** or **Script** button.

1. See the NeWS Programmer's Manual for more information.

Syntax Errors

A common error, when writing scripts, is to leave "{" and "}" or "(" and ")" brackets unbalanced, this results in a **syntaxerror**. When this error occurs, press **Done**. Fix the unbalanced brackets in the script before applying it again.

Invalid Names

The PostScriptError stack warns you about invalid names which are used in the script. You are not allowed to redefine PostScript operators or HyperLook functions.

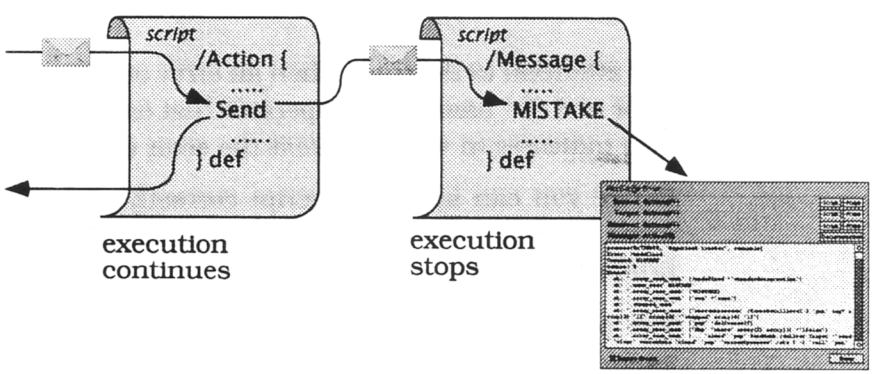
Use the documentation browser to find out about HyperLook functions.

Execution Errors

Errors may occur when you are using the system. As soon as an error occurs, the PostScriptError stack is displayed and the current message context is discarded. Execution continues in the context of the sender of the message.

Figure 174

The sender of a message is not affected when an error occurs at the receiving end.



The PostScript execution trace can help you find the cause of the problem. Only the most recent error is displayed if multiple errors occur. Switch the **Report Errors** check box off to stop errors from being displayed in the PostScriptError stack.

Unix Scripting Utilities

HyperLook provides some Unix utilities which give you control over objects from the Unix shell.

The HyperLook PostScript Shell

The **hlpsh** program (see **hlpsh(1)** on page C-6) lets you execute PostScript commands in the HyperLook environment. For example, type the following in a Unix shell:

```
hlpsh -i
Welcome to HyperLook
(Hello World!) ShowMessage
^D
```

This uses the **ShowMessage** function to display a string in the message window.

You can use the **hlpsh** program to create new HyperLook Unix commands. The following **sh** program iconifies all HyperLook stacks:

```
#!/bin/sh
hlpsh << EOF
StackDict {
   [] /IconifyStack 3 -1 roll Send
   pop
} forall
EOF
```

Sending Messages from the Shell

The **hlsend** program (see **hlsend(1)** on page C-7) is used to send message from a Unix shell to HyperLook objects. The format of the command is:

hlsend *object message args....*



Client Programming

8

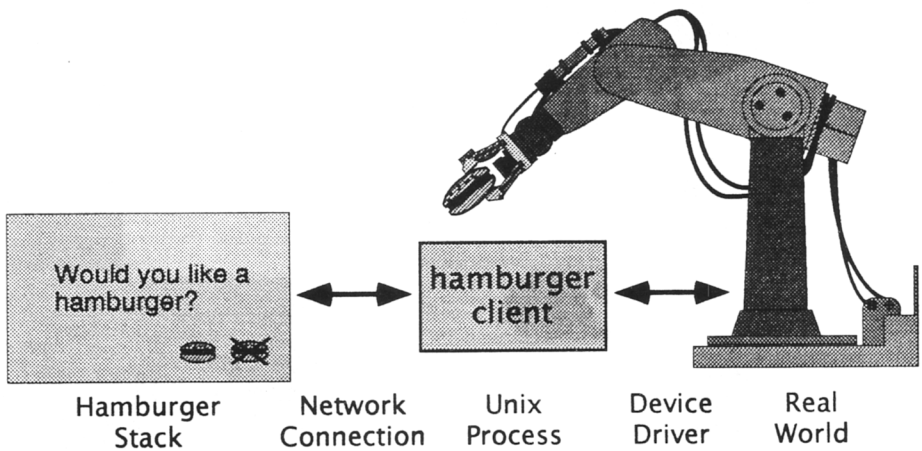
The HyperLook Client Interface

HyperLook objects live in the OpenWindows server. The HyperLook client interface allows Unix processes (called "clients") to connect to OpenWindows via the network, and communicate with HyperLook objects.

Clients can send messages to HyperLook stacks and objects. A client can also receive and respond to messages received from HyperLook. The way messages are passed to the user is discussed in "Messages" on page 7-12.

Figure 175

Interfacing HyperLook to the real world.



High Level Message Passing

HyperLook clients communicate at a high level by passing messages and arguments back and forth to HyperLook. This insulates your application from the look and feel, so you can easily reconfigure the user interface of your application without modifying the client, even while it's running.

Clients need not worry about details such as layout, drawing, and mouse tracking. They need only know about the high level messages which are relevant to the task at hand.

Because of this clean separation between HyperLook and the application, you can create a highly interactive graphical user interface before writing a line of code. The user interface will always be easy to change later on. In fact, the same client can run with any number of different user interfaces.

Client interface programming is very easy. There are only a few routines that you need to know about. Making an application respond to or send a message only requires a few lines of code.

Appendix D, "Library functions" contains a complete list of all client interface library routines.



Client Interface Overview

Below is the skeleton of a HyperLook client program[1]. It is explained in more detail in the rest of this section.

```
#include <hyperlook.h>

int handle_message(msg, argc, argv)
hl_any *msg, *argv[];
int argc;
{
    hl_send0("StackName:ObjectName","Message",NULL);
    return 1;
}

main()
{
    if (!hl_start("ClientName")) {
        fprintf(stderr,"Cannot connect to HyperLook!\n");
        exit(1);
    }
    hl_path(NULL);
    hl_connect("StackName");
    hl_show("StackName");
    hl_register(handle_message,
            "StackName:ObjectName","Message");
    hl_listen(hl_forever);
    hl_stop();
}
```

Initializing the Client Interface

Before communicating with HyperLook, a client must connect to the OpenWindows server and initialize the interface by calling the function `hl_start()`.

1. The source to this example can be found in "demos/client/skeleton.c" in the HyperLook directory.

The function **hl_path()** is called to tell HyperLook where to find the application's resources (see "The Resource Manager" on page 3-23). Passing **NULL** means that the resources are found in the directory where the client was started.

Before the client exits, it calls **hl_stop()** to clean up and disconnect from HyperLook.

Connecting to Stacks

To receive messages from a HyperLook stack, the client must be connected to it. A stack is connected to a client using **hl_connect()**. The **hl_show()** function displays the stack *and* connects it to the client.

Any number of stacks can be connected to one client, but each stack can have at most one client.

Receiving Messages from HyperLook

In order to receive messages from HyperLook, a client must register message patterns and handlers with **hl_register()**. A handler is called when a message matching a requested pattern is sent to the client.

In order to process messages, you call **hl_listen()**. Any messages that can't be handled by stacks to which you're connected are sent from HyperLook to the client, and are matched against its registered message patterns. If any patterns match, the appropriate handler functions are called with the message arguments, otherwise unmatched messages are ignored.

When a message is received, the handlers for the message are called in the same order as they were registered. Only the handlers that match the message and object name are called. Each handler has a chance to examine the message and arguments and decide whether or not to handle it.

If a handler returns `0` the next matching handler is called. If a handler returns `1`, then no other handlers are called. If a handler returns `-1`, no other handlers are called and **hl_listen()** returns.



Sending Messages to HyperLook

Clients can send messages to HyperLook objects by calling **hl_send0()**. This function takes a name identifying the target object, a message name, and any number of message arguments (terminated by **NULL**).

The message is sent from the client to HyperLook, and it is delivered to the appropriate object using HyperLook message passing (See "Messages" on page 7-12).

The message arguments passed to the C handler functions and **hl_send0()** are **hl_any** structures. These structures represent PostScript data structures such as names, strings, numbers and arrays. The HyperLook client interface provides a number of functions to create these data structures in C.

PostScript Data Structures in C

The **hl_any** data structure is declared in **hyperlook.h** as follows (see **hl_any(3)** on page D-3 for a more detailed description)[1]:

```
typedef enum {
    hl_number_type,
    hl_boolean_type,
    hl_string_type,
    hl_name_type,
    hl_array_type,
    hl_null_type,
} hl_any_type;

typedef struct hl_any {
    hl_any_type type;
    short len;
    union {
        double number;
        int boolean;
        char *string;
        char *name;
        struct {
            int argc;
            struct hl_any **argv;
        } array;
    } u;
} hl_any;
```

This data structure is used to represent the PostScript messages and arguments that are received from HyperLook. The same data structure is used to construct arguments to messages which are sent back to HyperLook.

There are a number of convenient library routines that let you create **hl_any** data structures. Using these routines you can create the same data structures that you can in PostScript. This lets you use PostScript methods (described Chapter 7, "Scripting") from C.

1. The complete include file is found in "include/hyperlook.h" in the HyperLook directory.

The Hamburger Client

This section gives a complete example of how to program the HyperLook client interface. It explains the client interface in detail[1].

The application is *very* simple. It is a program to ask whether you would like a hamburger. The application exits when you accept the offer.

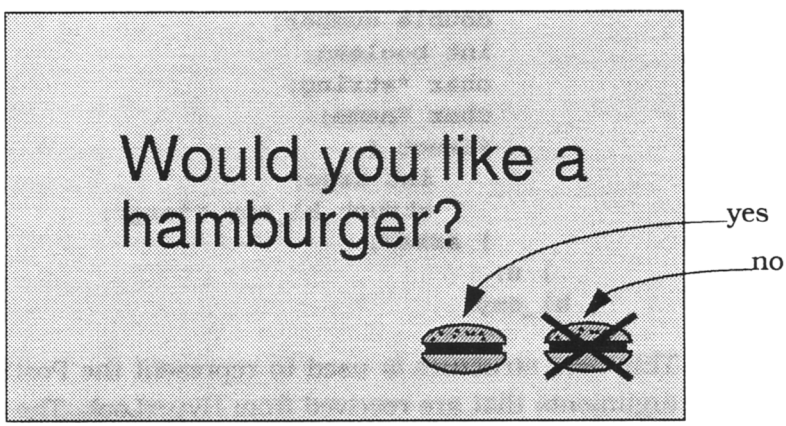
Creating the Hamburger Stack

A client needs to connect to a stack to receive messages, so the first thing to do is to build a stack (see Chapter 5, "Editing Stacks").

Figure 176 show a picture of the stack called **Hamburger**. It has a text object asking the question, and two drawing buttons. The left button is named **yes** and the right is named **no**.

Figure 176

The Hamburger stack.



1. The source to this example can be found in "demo/client/hamburger.c" in the HyperLook directory.

The Hamburger Code

Here is the complete listing of the Hamburger client program.

```
#include <hyperlook.h>

int no()
{
    hl_send0("Message", "Init",
        hl_new_string("Please reconsider my offer!"),NULL);
    return 1;
}

int yes()
{
    hl_send0("Message", "Init",
        hl_new_string("Have a nice day!"),NULL);
    return -1;
}

main(argc, argv)
int argc;
char *argv[];
{
    hl_verbose = 1;

    if (!hl_start(argv[0])) {
        fprintf(stderr,"Cannot connect to HyperLook!\n");
        exit(1);
    }
    hl_path(NULL);
    hl_connect("Hamburger");
    hl_show("Hamburger");

    hl_register(no,"Hamburger:no","Action");
    hl_register(yes,"Hamburger:yes","Action");
    hl_listen(hl_forever);

    hl_hide("Hamburger");
    hl_stop();
}
```

Called when the **no** button is pressed. It shows a message window.

Called when the **yes** button is pressed. It returns **-1** so **hl_listen** will terminate.

Set **hl_verbose** so that messages are printed on **stderr**.

Connect to HyperLook.

The **Hamburger** stack is loaded from the current directory.

Process messages until **yes()** returns **-1**, then clean up.

The Hamburger Code Explained

The first thing that the client must do is to initialize of the interface. It calls **hl_start()** and passes it the name of the client. This client will be known by that name to HyperLook.

It then adds the current directory to the client resource path, by calling **hl_path()**. HyperLook can now find the **Hamburger** stack which is located in the same directory as the program.

Next, it connects to the stack, and shows it. It is not necessary to call **hl_connect()** because **hl_show()** also connects to the stack. Now the **Hamburger** stack is displayed.

Before calling **hl_listen()** to process incoming message, the client registers two message handlers, **yes()** and **no()**, which will be called when the corresponding buttons are pressed. Finally it calls **hl_listen()** to process incoming messages.

When you click one of the buttons, a message named **Action** is sent to the button. In this example, neither the buttons, the card, the background, nor the stack have **Action** handlers. So the **Action** message is passed on to the client, and the corresponding message handler is called.

The handler sends a message to the HyperLook **Message** stack, telling it to display a string to the user. When the user finally presses the **yes** button, the **yes()** message handler is called. This handler returns **-1** which causes **hl_listen()** to terminate.

Finally, the client calls **hl_hide()** to remove the stack from the screen, followed by **hl_stop()** to clean up and break the connection with HyperLook.

HyperLook Include File

```
#include <hyperlook.h>
```

In order to use the HyperLook client interface, you must include the "hyperlook.h" include file. You'll find it in the "include" sub directory of the HyperLook directory.



Debugging

```
hl_verbose = 1;
```

Setting the **hl_verbose** flag causes diagnostics to be written to the client's **stderr** file. The diagnostics include a log of all messages sent to the client. A typical message looks like this:

```
hl_dispatch: message(Hamburger:Button(no),Action,[0])
```

The above means that an **Action** message is received with a single argument, the number zero. The target of the message is a **Button** object named **no** on the **Hamburger** stack.

Initialization

```
if (!hl_start(argv[0])) {
    fprintf(stderr,"Cannot connect to HyperLook!\n");
    exit(1);
}
```

This initializes the client interface, connects to OpenWindows server, and loads HyperLook if necessary. The argument is the name of the program, which identifies the client to HyperLook. If **hl_start()** returns 0 it wasn't possible to connect with HyperLook.

```
hl_path(NULL);
```

HyperLook needs to know where to find the client's resources (the Hamburger stack in this case), so **hl_path()** is called with an argument of **NULL**, to add the current directory to the client resource path. The argument to **hl_path()** can also be a string with an absolute directory name.

```
hl_connect("Hamburger");
```

This connects the client to the Hamburger stack, loading it from the resource directory if necessary. Once a client is connected to a stack, any messages the stack can't handle are sent to that client.

```
hl_show("Hamburger");
```



This shows the Hamburger stack on the screen. It also implicitly connects the client to the stack, if it's not already connected[1].

Registration

```
hl_register(no,"Hamburger:no","Action");
hl_register(yes,"Hamburger:yes","Action");
```

The above code registers the **no()** and **yes()** message handlers, so they're called whenever an **Action** message is received from the **yes** or **no** buttons on the **Hamburger** stack.

The message address is specified as "**stackname:objectname**". In this example the stack is called Hamburger and the buttons are called **no** and **yes**.

Notification

```
hl_listen(hl_forever);
```

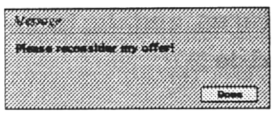
Calling **hl_listen()** enters the HyperLook notifier, which processes incoming messages and calls the appropriate handlers. **hl_listen()** exits when a handler returns **-1**.

The argument specifies how long the notifier should wait before calling the timeout handler. In this case we aren't interested in a timeout, so we pass **hl_forever**, so the timeout handler[2] is never called.

1. Strictly speaking, it is not necessary to call **hl_connect** in this example.
2. Use the **hl_register_timeout()** function to register a timeout handler.



Message Handlers

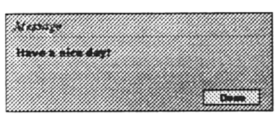



```
int no()
{
   hl_send0("Message", "Init",
    hl_new_string("Please reconsider my offer!"),NULL);
   return 1;
}
```

This message handler is called in response to an **Action** message, when the **no** button (with the picture of the crossed out hamburger) is pressed. It doesn't care about the message arguments, so they're not declared. It sends an **Init** message to the HyperLook **Message** stack (part of the HyperLook system), with one argument: a string to display to the user.

The message arguments passed to **hl_send0()** are of type **hl_any** (which is what **hl_new_string()** returns). A small message window is shown displaying the string. The **Message** stack is a facility provided by HyperLook to display messages.

The C message handlers return an integer to let **hl_listen()** know whether the message was handled. The **no** message handler returns **1**, which means that it was able to deal with the message and that no other handlers should be considered. Had it returned **0**, other matching handlers would have been given a chance.




```
int yes()
{
   hl_send0("Message", "Init",
       hl_new_string("Have a nice day!"),NULL);
   return -1;
}
```

The **yes** handler is similar to the **no** handler, except for its return value, which is **-1**. This causes **hl_listen()** to stop processing messages and return to its caller.

Cleanup

```
hl_hide("Hamburger");
```

Once the user presses the **yes** button, **hl_listen** returns, and the Hamburger stack is removed from the screen using **hl_hide()**.

```
hl_stop();
```

Calling **hl_stop** disconnects the client from the OpenWindows server and terminates the HyperLook connection.

Compiling the Hamburger

The source to this example is found in "**demo/client/hamburger.c**". You can compile the program (if you have write permission in this directory) by typing:

```
make hamburger
```

You can run the program, once the program has been successfully compiled, by typing:

```
hamburger
```

The program is compiled as follows. First **hamburger.c** is compiled to **hamburger.o**:

```
cc -c hamburger.c -I$HLHOME/include \
       -I$OPENWINHOME/include/wire \
       -I$OPENWINHOME/include/NeWS \
       -I$OPENWINHOME/include
```

The **-I** flags are used to specify the directories where the C preprocessor should find the include files. Then the program is linked with the following command:

```
cc -o hamburger hamburger.o -L$HLHOME/lib -lhl
```

This creates a binary called **hamburger**. The **-L** and **-l** flags are needed to link the HyperLook client interface library into the program.

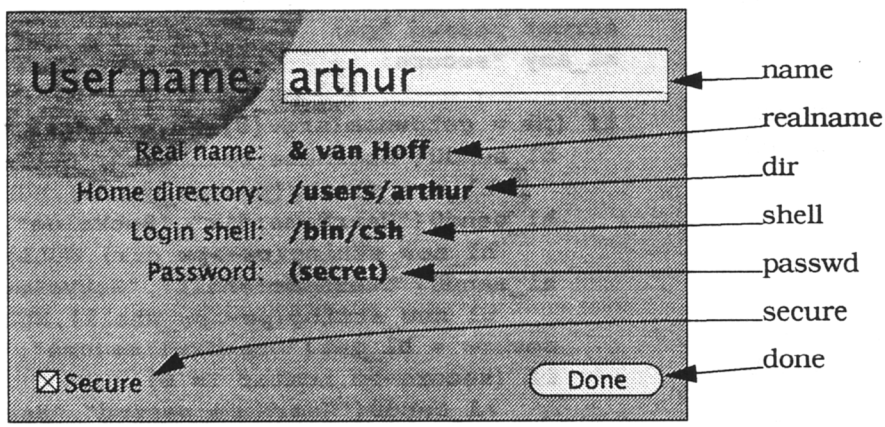
The UserName Client

This section discusses a more sophisticated example. The program provides a simple interface which lets you access information from the Unix password database[1].

When you enter a user name followed by return, the information about the user is displayed in the stack. The **getpwnam(3V)** Unix function call is used to access the information from the Unix password database.

Figure 177

The username stack.



The password is shown encrypted. If the **Secure** checkbox is marked, it should show the string "*(secret)*" instead of the encrypted password.

The program exits when you press the **Done** button.

The stack is called **UserName**, and it contains several fields and a button. The **name** field at the top right is the only editable one. The other fields are called **realname**, **dir**, **shell**, and **passwd**. The checkbox is called **secure**. The button is called **done**.

1. The source to this example can be found in "demo/client/username.c" in the HyperLook directory.

The UserName Code

The complete username program is listed below.

```
#include <pwd.h>
#include <hyperlook.h>

int request(msg,argc,argv)
hl_any *msg, *argv[];
int argc;
{
    struct passwd *pw;
    hl_any *secure;

    if (pw = getpwnam(argv[0]->u.string)) {
        hl_send0("UserName:realname","SetValue",
            hl_new_string(pw->pw_gecos),NULL);
        hl_send0("UserName:dir","SetValue",
            hl_new_string(pw->pw_dir),NULL);
        hl_send0("UserName:shell","SetValue",
            hl_new_string(pw->pw_shell),NULL);
        secure = hl_get("UserName:secure","Value");
        if (secure->u.number != 0)
            hl_send0("UserName:passwd","SetValue",
                hl_new_string("(secret)"),NULL);
        else
            hl_send0("UserName:passwd","SetValue",
                hl_new_string(pw->pw_passwd),NULL);
        hl_free(secure);
    } else {
        hl_send0("UserName:realname","SetValue",
            hl_new_string("-- not found --"),NULL);
        hl_send0("UserName:dir","SetValue",
            hl_new_string(""),NULL);
        hl_send0("UserName:shell","SetValue",
            hl_new_string(""),NULL);
        hl_send0("UserName:passwd","SetValue",
            hl_new_string(""),NULL);
    }
    hl_send0("UserName:name","SelectAll",NULL);
    return 1;
}
```

This function is called when the user has entered a name.

Call **getpwnam** to get the user info.

Show the password if it can be disclosed.

Clear the fields if the user is not found.

Select the name that the user typed.

This function is called when the **Done** button is pressed.

```
int done()
{
    return -1;
}

main(argc,argv)
int argc;
char *argv[];
{
    hl_verbose = 1;

    if (!hl_start(argv[0])) {
        fprintf(stderr,"Cannot connect to HyperLook!\n");
        exit(1);
    }

    hl_path(NULL);
    hl_show("UserName");

    hl_register(request,ANY,"Action");
    hl_register(done,"UserName:done","Action");
    hl_listen(hl_forever);

    hl_hide("UserName");
    hl_stop();
}
```

Connect to HyperLook.

Show the stack.

Register message handlers.

The UserName Code Explained

The structure of the program is very similar to the previous example. The client shows the UserName stack (automatically connecting to it), and registers two message handlers.

The **request()** handler is called when a **Action** message is received from the **name** field. The handler uses the string argument of the message and looks up the user name in the password database.

If the name is found, **SetValue** messages are sent to the data fields of the **UserName** stack.



Before the **passwd** field is set, the value of the **Secure** checkbox is examined to determine whether the encrypted password should be displayed. The program uses the **hl_get()** routine to get the value of the checkbox.

The program exists when an **Action** message is received from the **done** button.

Message Handler Arguments

When **hl_listen()** calls a handler, it passes three arguments to the handler.

```
int request(msg,argc,argv)
hl_any *msg, *argv[];
int argc;
{
   /* message handler */
}
```

The first argument, **msg**, is an **hl_any** data structure which contains the complete message that **hl_listen()** received from HyperLook. The **argv** argument is an array of length **argc**, containing pointers to **hl_any** data structures, which are the arguments to the message.

The **request()** handler is called in response to an **Action** message from a **field** object. It has one argument, the string that the user typed. The string data can be accessed as follows:

```
argv[0]->u.string
```

Compiling the UserName client

The source to this example is found in **demo/client/username.c**. You can compile the program (if you have write permission in this directory) by typing:

```
make username
```

You can run the program, once the program has been successfully compiled, by typing:

```
username
```




A

Glossary

Background

A place holder for components on a stack. Every card has at least one BackGround. A BackGround can be shared by several cards. Therefore objects on a background can be visible on more than one card. The parent of a BackGround is the stack.



Button

A user interface component that performs an action when it is pressed. Buttons can have different styles, for example: push button, checkbox, drawing button, etc.

Card

A card is like a page in a book. Every stack contains at least one card. Only one card of a stack can be visible at any time. You can flip between cards like turning the pages of a book. The parent of a card is a BackGround.



Checkbox

A checkbox is a user interface component that can either be *on* or *off*. It is on when it is crossed. It is just another style of button.

Class

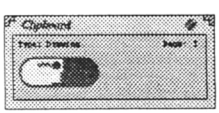
A class contains the code describing an object type. Classes inherit functionality from their super classes.

Client

A client is a Unix process that is connected to HyperLook. It can communicate with stacks and objects by sending and receiving messages.

Client Interface

A set of C routines which let you write client programs that communicate with HyperLook.



Clipboard

A place holder for data when it is moved between stacks and applications. It can contain graphics, text, images, etc.



Color: 

ColorSelect

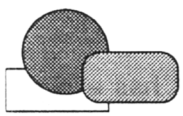
A user interface component that lets you select a color from a color pallet.

Component

A user interface object such as a button, field or slider.

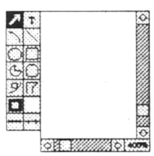
Desktop

The OpenWindows user interface metaphor for the screen. It contains windows and icons to represent files and applications.



Drawing

A HyperLook PostScript graphics object that can be edited using the HyperLook graphics editor. It can be translated to Encapsulated PostScript format.



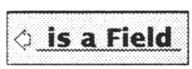
DrawTool

A user interface component that lets you edit graphics interactively. It is HyperLook graphics editor in a single object.



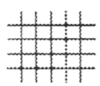
Encapsulated PostScript

A standard format which is used when PostScript files are transported between applications. Most desktop publishing and graphics editing applications know how to import and export this format.



Field

A user interface component that lets you edit a single line of text.

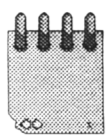


Grid

A grid defines the accuracy at which editing operations are performed.

Group

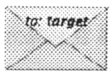
A group is a set of graphical objects which can be manipulated as if they are a single object.



Icon

A small graphical representation of a stack or an application.





Message

Messages are passed between HyperLook objects and clients as a means of communication.

Method

A method is a function which is defined in a class or script. The function is executed in the context of the object.

NeWS

Network extensible Windowing System. This is the name for the PostScript interpreter which is part of OpenWindows.

Object Oriented

A way to structure programs and data. Each object has a class which defines its functionality. Classes are organized in a hierarchical structure that allows for inheritance of functionality.



OPENLOOK®

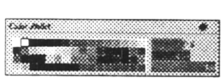
A user interface "look and feel" using lots of bevelled edges[1]. This is the default "look and feel" of OpenWindows.

OpenWindows

The environment in which HyperLook runs. It manages the windows and icons which you normally see on the screen.

Operator

A PostScript function.



Pallet

A set of things from which you can choose. For example, a color pallet.



Polygon

A polygon is a graphical object made of straight lines only.

PostScript

An interpreted language that was especially designed for graphics.

1. OPENLOOK is a registered trademark of Unix systems Laboratories Incorporated.



PullDown

A button with a menu associated to it. When the button is pressed, the menu is shown and you can select an option.

Resource

A file used by HyperLook. Resources are accessed using the resource manager.



Script

A script contains code that customizes an object. Each HyperLook object can have a script. Scripts contain method and variable definitions that can redefine and extend the default behavior of objects.



Scrollbar

A scrollbar is a user interface component that is used to view data. Its value reflects which data is visible.



Slider

A slider is a user interface component that lets you select a numeric value from a range.



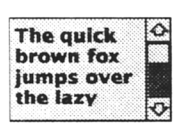
Spline

A spline is a smooth polygon.



Stack

A HyperLook window. It contains components, cards and backgrounds.



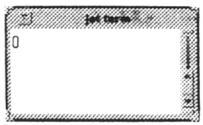
Text

A text object is a component that lets you edit multiple lines of text. It is a simple text editor.

Unix

The operating system which runs on Sun computers. It is the software that supports all programs on your computer.





Window

An area of the screen usually dedicated to a single application. Windows can be moved and resized. You can turn them into icons if they take up too much screen space.

Zoom

The magnification at which graphics are displayed. Zooming *in* means that the graphics are enlarged. Zooming *out* means that they get smaller.

B

Methods and Variables

HyperLook Methods and Variables

This Appendix briefly mentions the most important functions, methods and variables which are used by HyperLook. The list is by no means complete. Use the documentation browser ("The Documentation Browser" on page 7-10) to examine the full list of documented methods and variables.

These methods are defined in the HyperLook class and are always available.

```
/DEBUG              % anything --
/ShowMessage        % string|array-of-strings --

/ShowStack          % stack|stackname --
/HideStack          % stack|stackname--

/FindStack          % stackname -- stack|null
/FindObject         % stackname objectname -- object|null
/LookupObject       % name -- object|stack|null

/Send               % array-of-arg message target --
/ParentSend         % array-of-arg message --
/ClientSend         % array-of-arg message --
```

Stack Methods

These methods can only be used in the script of a stack, or sent to a stack.

```
/Move              % x y --
/ToFront           % --
/ToBack            % --
/IconifyStack      % --
/DeIconifyStack    % --
/FlipIconifyStack  % --

/GoHomeCard        % --
/GoLastCard        % --
/GoNextCard        % --
/GoPreviousCard    % --
/GotoCard          % number|name --
```

Card Object Methods and Variables

These variables can be referenced in the script of every card object, e.g. Buttons, Sliders etc:

```
/Value          % -- value
/Width          % -- number
/Height         % -- number
/Color          % -- color
/FillColor      % -- color
/FontName       % -- name
/FontSize       % -- number

/MyStack        % -- stack
/MyClient       % -- file
/MyCanvas       % -- canvas
```

These methods can be used in the script of every card object, e.g. Buttons, Sliders etc:

```
/Draw           % --
/Damage         % --

/Move           % x y --
/Reshape        % x y width height --
/Show           % --
/Hide           % --

/OnAscii        % char --
/OnMouse        % --
/OnHelp         % --
/OnOpen         % --
/OnClose        % --
```

Button Methods

```
/Action         % number --
/SetValue       % number --
/Path           % --
/Press          % --
```




Field Methods

```
/Action       % string --
/SetValue     % string --
/Selection    % -- string
/SelectAll    % --
```

Text Methods

```
/Action       % string --
/WordAction   % string --
/SetValue     % string --
/Selection    % -- string
/SelectAll    % --
```

List Methods

```
/Action       % array-of-strings --
/SetValue     % array-of-strings --
/Select       % array-of-strings --
/Selected     % -- array-of-strings
```

PullDown Methods

```
/Action       % string --
/SetValue     % string|array-of-strings --
```

Slider Methods

```
/Action       % number --
/SetValue     % number --
/SetRange     % min val max --
/ConvertValue % number -- string
```

DrawTool Methods

```
/Action       % drawing --
/SetValue     % drawing --
/GetValue     % -- drawing
```

ColorSelect Methods

```
/Action        % color --
/SetValue      % color --
```

C Unix Commands

drawps(1)

NAME

drawps

Convert HyperLook drawings to PostScript format.

SYNOPSIS

drawps [-epsf] [-scale f] [-hdr hdr] [file]

DESCRIPTION

drawps is a filter to translate HyperLook drawings produced with the **hyperdraw(1)** graphics editor into PostScript or Encapsulated PostScript format.

Input is taken from the standard input, or from the specified file. The output is produced on the standard output.

OPTIONS

-epsf

Generate Encapsulated PostScript headers. These files can be imported by other programs.

-scale f

Scale the drawing by the indicated factor. The factor must be a floating point number greater than 0.

-hdr hdr

Use this header when generating Encapsulated PostScript files. Some programs insist on a particular header string. The default is:

```
!PS-Adobe-2.0 EPSF-1.2
```

EXAMPLES

To print a drawing at 50% magnification use:

```
drawps -scale 0.5 logo.draw | lpr -Plw
```

ENVIRONMENT

drawps requires that the **$HLHOME** environment variable is set.

FILES

$HLHOME/lib/ps/drawlib.ps
PostScript printer initialization file.

$HLHOME/lib/ps/lib.ps
Initialization file.

SEE ALSO

hyperdraw(1), **hyperlook(1)**.



exithyperlook(1)

NAME

exithyperlook

Exit HyperLook.

SYNOPSIS

```
exithyperlook [ -f ]
```

DESCRIPTION

The **exithyperlook** command exits the HyperLook system. The **ExitHyperLook** stack is shown to let the user save stacks before really exiting.

OPTIONS

-f

Exit without asking. The user gets no chance to save stacks.

EXAMPLES

To restart HyperLook quickly without asking the user:

```
exithyperlook -f ; hyperlook -c
```

SEE ALSO

hyperlook(1).



hlpath(1)

NAME

hlpath

Manage HyperLook resource directories.

SYNOPSIS

hlpath [-a] [-r] [-ra] dir ...

DESCRIPTION

The **hlpath(1)** command adds or removes user or application resource directories to HyperLook. The default is to add user resource directories.

HyperLook resource directories are searched when looking for resources such as stacks. The default user resource directory is **~/stacks**.

OPTIONS

-a

Add the specified application resource directories.

-r

Remove the specified user resource directories.

-ar

Remove the specified application resource directories.

FILES

HyperInit.ps

This PostScript file is executed if it exists in the resource directory when it is added to the HyperLook resource directory list. It allows applications to perform initializations.

./mono

When a resource directory is added when HyperLook is running on a monochrome display, and the resource directory has a sub directory called **mono**, then this directory is added before the resource directory itself. This allows you to design interfaces for both monochrome and color screens.

SEE ALSO

hyperlook(1).



hlpsh(1)

NAME

hlpsh

The HyperLook version of **psh(1)**.

SYNOPSIS

hlpsh [-s] [-i] file ...

DESCRIPTION

hlpsh uses **psh(1)** to interpret PostScript commands from the specified files or from standard input. The PostScript environment is setup so that the HyperLook dictionary is on the top of the dictionary stack.

If HyperLook is not running, it will be started.

OPTIONS

-s

Work silently. If HyperLook is not running, exit immediately.

-i

Run in interactive mode. Report errors to the current terminal instead of the console.

EXAMPLES

This example illustrates the loading and displaying of a stack from PostScript:

```
hlpsh -i
Welcome to HyperLook version 1.5 (TNT 2.0)
/system FindStack dup =
dictionary[40/50000]
ShowStack
^D
```

SEE ALSO

hyperlook(1), **psh(1)**.

hlsend(1)

NAME

hlsend

Send a message to an object in HyperLook.

SYNOPSIS

hlsend object message arg ....

DESCRIPTION

hlsend uses **hlpsh(1)** to send a message to an object in HyperLook. The object is specified either as a stack name, or as **stackname:objectname**. The message is specified as a single word, without the leading slash. The message arguments must be in PostScript format.

EXAMPLE

To move the system stack use:

```
hlsend system Move 100 100
```

To set the value of a field in a stack use:

```
hlsend SystemStatus:status SetValue "(Hello world)"
```

SEE ALSO

hyperlook(1), **hlpsh(1)**.

hyperdraw(1)

NAME

hyperdraw

HyperLook graphics editor.

SYNOPSIS

hyperdraw [filename]

DESCRIPTION

HyperDraw is a PostScript graphics editor. It is part of the HyperLook system.

The **hyperdraw** command starts HyperLook if needed. If a drawing file is specified, it is loaded into the editor. The file name must end in **.draw**.

EXAMPLES

To edit a drawing from the HyperLook home directory use:

```
hyperdraw $HLHOME/draw/garfield.draw
```

SEE ALSO

hyperlook(1).

hyperlook(1)

NAME

hyperlook

install_hyperlook

HyperLook startup script.

SYNOPSIS

hyperlook [-c] [-fix]
install_hyperlook

DESCRIPTION

HyperLook is an object oriented PostScript based user interface development system. It uses the stack and card model for the design of user interfaces by direct manipulation. User interface components can be programmed using PostScript as a scripting language.

The **hyperlook** command starts HyperLook; no action is taken if HyperLook is already running. OpenWindows is started if it is not yet running.

Use **install_hyperlook(1)** to install the HyperLook icons in the **filemgr(1)** data base.

OPTIONS

-c

Startup quickly, not showing the introduction stack.

-fix

Restart and fix things when HyperLook has crashed (not that it has ever happened).



ENVIRONMENT

The **hyperlook(1)** startup script recognizes the following shell variables. Some of these are used when starting OpenWindows.

HLHOME
The HyperLook home directory.

OPENWINHOME
The OpenWindows home directory.

OPENWINOPTIONS
Options passed to OpenWindows at startup. See **openwin(1)**.

DISPLAY
The display on which HyperLook should run. HyperLook will use the color framebuffer if one is available.

FILES

~/stacks
The default directory where user resource files are saved. The most common resource file is a **.stack** file containing a HyperLook stack.

~/stacks/.hyperlook
The file where the HyperLook user preferences are saved.

~/stacks/.hyperlook-init
This file is executed at startup. It should only contain **sh(1)** commands to show stacks and perform initializations.

~/stacks/HyperInit.ps
This PostScript file is executed when the **~/stacks** directory is added by the resource manager. See also **hlpath(1)**.

SEE ALSO

exithyperlook(1), **hidestack(1)**, **hlpath(1)**, **hlsend(1)**, **hlpsh(1)**, **hyperdraw(1)**, **install_hyperlook(1)**, **showstack(1)**.

DIAGNOSTICS

Make sure that you don't have any dangerous files in your home directory when starting HyperLook. Remove **.xinitrc**, **.startup.ps**, **.user.ps**, **.init.ps**, if you have problems starting HyperLook.



Start OpenWindows first and start HyperLook using the **hyperlook(1)** command from the console, if all else fails.

showstack(1)

NAME

showstack

hidestack

Show and hide HyperLook stacks and objects.

SYNOPSIS

```
showstack stack
hidestack stack
```

DESCRIPTION

The **showstack** command displays a HyperLook stack (window) and brings it to the front. If the stack is iconified, the icon is opened. The **hidestack** command hides a stack.

The stack is specified either by its name (without the **.stack** file name extension) or by using its file name. If the stack is specified as a name it must be accessible by the HyperLook resource manager. If the stack is specified as a file name its directory will be added to the user resource directory list.

The stack is loaded from a resource directory if it is not already in memory. See **hlpath(1)** on how to add a resource directory.

SEE ALSO

hlpath(1), **hyperlook(1)**.

DIAGNOSTICS

The **hidestack** command loads the stack before hiding it.



D

Library Functions

hl_alloc(3)

NAME

hl_alloc
hl_destroy
hl_use
hl_free
hl_constant
Manage **hl_any** data structures.

SYNOPSIS

```
#include <hyperlook.h>

void *hl_alloc(bytes)
int bytes;

void hl_destroy(ptr)
void *ptr;

void hl_use(any)
hl_any *any;

void hl_free(any)
hl_any *any;

void hl_constant(any)
hl_any *any;
```

DESCRIPTION

The **hl_alloc()** and **hl_destroy()** routines are used for low level memory management. You can redefine these routines in your program to use your own memory manager.

The **hl_use()** routine increments the reference count of an **hl_any** data structure. **hl_free()** decrements the reference count of an **hl_any** data structure. If it reaches **0** the data structure is actually deleted. **hl_constant()** freezes the reference count of an **hl_any** data structure so that it is never deleted.

SEE ALSO

hyperlook(1), **hl_any(3)**, **hl_send(3)**.



hl_any(3)

NAME

hl_any
hl_new_null
hl_new_boolean
hl_new_number
hl_new_string
hl_new_string_body
hl_new_name
hl_new_array
hl_new_array_body
Create **hl_any** data structures.

SYNOPSIS

```
#include <hyperlook.h>

hl_any *hl_new_null()

hl_any *hl_new_boolean(flag)
int flag;

hl_any *hl_new_number(n)
float n;

hl_any *hl_new_string(str)
char *str;

hl_any *hl_new_string_body(len)
int len;

hl_any *hl_new_name(str)
char *str;

hl_any *hl_new_array(obj0, ..., NULL)
hl_any *obj0, ...;

hl_any *hl_new_array_body(len)
int len;
```



DESCRIPTION

The **hl_new** routines allocate **hl_any** data structures which are used to represent data for communication with HyperLook. The **hl_any** data structure is defined in the file **include/hyperlook.h** in the HyperLook directory.

The **hl_new_null()**, **hl_new_boolean()**, **hl_new_number()**, **hl_new_name()**, **hl_new_string()** routines allocate a data structure containing the appropriate data type.

hl_new_string_body() allocates a data structure containing a string of length **len** containing **'\0'** characters.

hl_new_array() allocates an array of **hl_any** objects. The members of the array are specified in the argument list which must be terminated by a **NULL**. **hl_new_array_body()** allocates an array of length **len** containing **NULL** pointers.

The **hl_any** data structure is defined as:

```
typedef struct hl_any {
   hl_any_type type;
   short len;
   union {
       double number;
       int boolean;
       char *string;
       char *name;
       struct {
           int argc;
           struct hl_any **argv;
       } array;
   } u;
} hl_any;
```

The **len** field is the length of strings and names. The **type** field indicates how the other fields are used. It is of type **hl_any_type**:

```
typedef enum {
    hl_number_type,
    hl_boolean_type,
    hl_string_type,
    hl_name_type,
    hl_array_type,
    hl_null_type,
} hl_any_type;
```

EXAMPLES

Create an array of objects which can be sent to HyperLook:

```
hl_any *a = hl_new_array(
    hl_new_number(30), hl_new_number(40.5),
    hl_new_name("Dug"), hl_new_string("Scoular"),
    hl_new_array(hl_new_boolean(1),NULL),
    NULL);
```

The PostScript version of the array is:

```
[30 40.5 /Dug (Scoular) [true]]
```

Elements of the array can be accessed using the **hl_any** data structure:

```
printf("n = %d\n",a->u.array.argc);
printf("x = %g\n",a->u.array.argv[0]->u.number);
printf("y = %g\n",a->u.array.argv[1]->u.number);
printf("name = %s %s\n",a->u.array.argv[2]->u.name,
    a->u.array.argv[3]->u.string);
```

FILES

include/hyperlook.h
: The include file containing all the HyperLook client interface definitions.

SEE ALSO

hyperlook(1), **hl_print(3)**, **hl_free(3)**, **hl_use(3)**, **hl_send(3)**.

hl_exists(3)

NAME

hl_exists

Check the existence of a HyperLook object.

SYNOPSIS

```
#include <hyperlook.h>

int hl_exists(target)
char *target;
```

DESCRIPTION

hl_exists() checks the existence of a HyperLook object. The object is addressed in the same way as **hl_send(3)**.

EXAMPLES

The following program checks for the existence of an object called "**name**" on a stack called "**entry**".

```
if (hl_exists("entry:name")) {
    /* it's there! */
    ...
}
```

RETURN VALUES

If the object exists, **1** is returned. If not, or if the connection to HyperLook is lost, **0** is returned.

SEE ALSO

hyperlook(1), **hl_send(3)**.

hl_flush(3)

NAME

hl_flush

hl_flush_input

Flush HyperLook input and output.

SYNOPSIS

```
#include <hyperlook.h>

void hl_flush()

void hl_flush_input()
```

DESCRIPTION

hl_flush() flushes all pending output to HyperLook. Output is automatically flushed before reading input (in **hl_listen(3)**).

hl_flush_input() causes all pending input to be discarded.

SEE ALSO

hyperlook(1), **hl_start(3)**, **hl_listen(3)**.

hl_get(3)

NAME

hl_get

hl_put

Access to instance variables.

SYNOPSIS

```
#include <hyperlook.h>

hl_any *hl_get(target,param);
char *target, *param;

int hl_put(target,param,value);
char *target, *param;
hl_any *value;
```

DESCRIPTION

hl_get() provides direct access to instance variables of HyperLook objects. The **target** specifies the object in the same way as the first argument to **hl_send(3)** does.

The **param** argument is the name of the variable which is accessed. **hl_get** returns an **hl_any** data structure containing the variable's value. This data structure must be de-allocated using **hl_free(3)**.

hl_put() assigns a new value to an instance variable of a HyperLook object. The new value is specified as an **hl_any** data structure which must be de-allocated using **hl_free(3)** if it is not used again.

RETURN VALUES

hl_get() returns a **NULL** pointer when an error occurs or if HyperLook is not started. It returns **hl_null** if the parameter does not exist.

hl_put() returns **1** on success, and **0** on failure. Failure only occurs if the connection to HyperLook is lost.

EXAMPLES

Get the "**Editable?**" parameter of a text object called "**name**" on a stack called "**entry**":

```
hl_any *b = hl_get("entry:name","Editable?");
if ((b->type == hl_boolean_type) &&
    (b->u.boolean != 0)) {
   ...
}
hl_free(b);
```

Set the "**Editable?**" flag of the same object:

```
hl_any *b = hl_new_boolean(1);
hl_put("entry:name","Editable?",b);
hl_free(b);
```

SEE ALSO

hyperlook(1), **hl_start(3)**, **hl_send(3)**.

DIAGNOSTICS

hl_get() and **hl_put()** provide unrestricted access to HyperLook objects. This means that they can easily be misused. Use them with care!



hl_listen(3)

NAME

hl_listen
hl_register
hl_register_any
hl_register_quit
hl_register_timeout
hl_register_ioerror
HyperLook message handling.

SYNOPSIS

```
#include <hyperlook.h>

int hl_listen(delay)
int delay;

void hl_register(handler,obj,message)
int (*)() handler;
char *obj;
char *message;

void hl_register_any(handler,match)
int (*)() handler;
hl_any *match;

void hl_register_quit(handler)
int (*)() handler;

void hl_register_timeout(handler)
int (*)() handler;

void hl_register_ioerror(handler)
int (*)() handler;
```

DESCRIPTION

hl_listen() is the main HyperLook event loop. It reads messages from HyperLook and calls the appropriate handlers.

hl_register(), **hl_register_any()**, **hl_register_timeout()**, and **hl_register_ioerror()** install handlers for incoming messages. The handlers are called when a message is received from HyperLook.

The **delay** argument to **hl_listen** is a delay in milliseconds (1000ms is 1 second). If no input is received for **delay** milliseconds, an **hl_timeout** message is generated. You can use **hl_forever**, to make it never time out.

hl_listen doesn't return upon timeout. It calls the timeout handler. Any handler can return **-1** to cause **hl_listen** to return.

The handler argument to the **hl_register()** and **hl_register_any()** is a function returning an integer. Here is an example of a handler function:

```
int button_pressed(message,argc,argv)
hl_any *message;
int argc;
hl_any *argv[];
{
    ....
    return 1;
}
```

When handler returns **0** the message is matched against other handlers. The handler should return **1** if the message is handled and no other handlers should be considered. When the handler returns **-1** the **hl_listen()** event loop returns to its caller.

The **obj** argument of **hl_register()** specifies which object the message responds to. The object can be specified as follows:

```
ANY -- any object
"*" -- any object
"x" -- stack x
"x:*" -- any object on stack "x"
"x:y" -- an object "y" on stack "x"
"*:y" -- any object called "y"
"Button(*)" -- any button
"Button(y)" -- any button called "y"
"x:Button(*)" -- any button on stack "x"
"x:Button(y)" -- a button called "y" on stack "x"
```

In the above examples **Button** is used as an object type. You can use other object types like **Field** and **Slider**, too.

The message argument of **hl_register()** specifies which message is allowed. Any message is matched if you specify the message as **ANY**.

hl_register_any() lets you register for any type of input data. The **match** data structure is matched against the input. Any part of the data structure may contain **ANY** which matches anything.

hl_register_quit() lets you register a handler for **hl_quit** messages. An **hl_quit** message is generated when the connection with HyperLook is lost.

hl_register_ioerror() lets you register a handler for **hl_ioerror** messages. An **hl_ioerror** message is generated when invalid data is read from HyperLook.

RETURN VALUES

hl_listen() returns **1** on normal exit. When an **hl_ioerror** is found it returns **-1**.

EXAMPLES

This is an example of a handler for a Field object. It is registered as:

```
hl_register(search_handler,"*:Field(x)","Action");
```

The **Action** message of a field has one argument (the string). The string is accessed through the **hl_any** data structure.

```
int search_handler(message,argc,argv)
hl_any *message;
int argc;
hl_any *argv[];
{
   char *s = argv[0]->u.string;
   /* start search */
   ....

   return 1;
}
```

SEE ALSO

hyperlook(1), **hl_start(3)**.

DIAGNOSTICS

hl_listen() generates an **hl_quit** message when the connection with HyperLook is lost. An **hl_ioerror** is generated if an I/O error occurs.



hl_path(3)

NAME

hl_path

Register a resource directory.

SYNOPSIS

```
#include <hyperlook.h>

int hl_path(dir)
char *dir;
```

DESCRIPTION

hl_path() registers an application resource directory with the HyperLook resource manager. Once registered, the application can access resources (stacks) in the directory.

The argument **dir** is the full path name of the resource directory. It can be **NULL**, meaning the current directory.

EXAMPLES

The following main program shows a stack which has been saved in the directory from which the program is executed.

```
#include <hyperlook.h>
main()
{
    hl_start("myclient");
    hl_path(NULL);
    hl_show("mystack");
    hl_stop();
}
```

RETURN VALUE

hl_path() returns **1** on success, **0** if the connection to HyperLook is lost.

SEE ALSO

hyperlook(1), **hl_start(3)**, **hl_show(3)**.

hl_print(3)

NAME

hl_print

Print an **hl_any** data structure to a file.

SYNOPSIS

```
#include <hyperlook.h>

void hl_print(fp,any)
FILE *fp;
char *prog;
```

DESCRIPTION

hl_print() prints the **hl_any** data structure specified in **any** to the **fp** file in human readable format. This function should be used for debugging only.

SEE ALSO

hyperlook(1), **hl_verbose(3)**.

hl_ps(3)

NAME

hl_ps

Execute raw PostScript in HyperLook.

SYNOPSIS

```
#include <hyperlook.h>

int hl_ps(code)
char *code;
```

DESCRIPTION

hl_ps() sends a string containing PostScript commands to HyperLook to be executed in the server. You need to call **hl_flush(3)** to flush the output stream after calling **hl_ps()**.

EXAMPLES

The following PostScript example shows the system stack using PostScript only (you can also use **hl_show(3)**).

```
hl_ps("/system FindStack ShowStack");
```

RETURN VALUES

hl_ps() returns 1 on success, 0 if the connection to HyperLook is lost.

SEE ALSO

hyperlook(1), **hl_flush(3)**.



hl_send(3)

NAME

hl_send

hl_send0

Send messages to HyperLook objects.

SYNOPSIS

```
#include <hyperlook.h>

int hl_send(target,message,args)
char *target, *message;
hl_any *args;

int hl_send0(target,message,arg0,....,NULL)
char *target, *message;
hl_any *arg0, ...;
```

DESCRIPTION

The **hl_send()** and **hl_send0()** routines send a message to a HyperLook object. The **message** argument is the name of the message. The **target** argument is the name of the object to which the message should be sent. For example:

```
"x" - a stack called "x"
"x:y" - an object "y" on stack "x"
```

The arguments to **hl_send()** are specified in **args** as an **hl_any** array. The array can be allocated using **hl_new_array(3)**. The argument array must be de-allocated, after sending the message, using **hl_free(3)**.

The arguments to **hl_send0()** are specified in the argument list which must be **NULL** terminated. The arguments are de-allocated by **hl_send0()** using **hl_free()**.

Usually you should use **hl_send0()** to send messages to objects. It uses a more compact notation and it requires no explicit freeing of data structures. If you want to send the same message frequently, or if you want to avoid repeated memory allocations, the you should use **hl_send()**.

RETURN VALUES

Both **hl_send()** and **hl_send0()** return **1** if the message is sent successfully. They return **0** if HyperLook was not started or if the target is invalid.

EXAMPLES

Set the value of a text object called "**name**" on stack "**entry**":

```
hl_any *a = hl_new_array(
   hl_new_string("Some text"),
   NULL);
hl_send("entry:name","SetValue",a);
hl_free(a);
```

The same message can be sent using **hl_send0** as follows (note that the argument string is deallocated automatically):

```
hl_send0("entry:name","SetValue",
   hl_new_string("Some text"),
   NULL);
```

SEE ALSO

hyperlook(1), **hl_any(3)**.

hl_show(3)

NAME

hl_show

hl_hide

hl_connect

Show and hide stacks and objects, and connect to stacks.

SYNOPSIS

```
#include <hyperlook.h>

int hl_show(target);
char *target;

int hl_hide(target);
char *target;

int hl_connect(target);
char *target;
```

DESCRIPTION

The **hl_show()** and **hl_hide()** routines let you show and hide HyperLook objects and stacks. The argument is the name of a stack, like "**mystack**", or an object, like "**mystack:field**". Hiding a stack removes its window from the screen. Hiding an object makes it invisible.

hl_connect() connects the client to a stack directly (without showing it). **hl_show()** implicitly connects to the stack before it is shown. A client must be connected to a stack if it wants to receive messages from the stack. You can't connect to an object.

A client can be connected to any number of stacks. A stack can be connected to only one client.

RETURN VALUES

On success the routines return **1**, on failure **0**. Failure only occurs if the connection to HyperLook is lost.

EXAMPLES

To show a stack called "**entry**" use:

```
hl_show("entry");
```

To connect to the same stack without showing it:

```
hl_connect("entry");
```

To hide an object called "**search**" on stack "**entry**" use:

```
hl_hide("entry:search");
```

SEE ALSO

hyperlook(1), **hl_start(3)**, **hl_send(3)**, **hl_path(3)**.

DIAGNOSTICS

Stacks must be accessible by the resource manager before they can be shown. See **hl_path(3)**.

hl_start(3)

NAME

hl_start

hl_stop

hl_verbose

Connect to HyperLook.

SYNOPSIS

```
#include <hyperlook.h>

int hl_start(prog)
char *prog;

int hl_stop()

int hl_verbose;
```

DESCRIPTION

hl_start() attempts to establish a connection to HyperLook. It should be called before any other HyperLook client interface routine is used. **prog** is the name which is used by HyperLook to identify the client.

hl_stop() terminates the connection with HyperLook.

The **hl_verbose** flag lets you monitor message as they are received. If this flag is non zero message are printed to **stderr**. The default value is **0**.

RETURN VALUES

hl_start() returns **1** when a connection is successfully established, it returns **0** otherwise.

hl_stop() always returns **1**.

EXAMPLES

This is an example of a skeleton main program.

```
#include <stdio.h>
#include <hyperlook.h>

main(argc,argv)
int argc;
char *argv[];
{
   hl_verbose = 1;

   if (!hl_start(argv[0])) {
       fprintf(stderr,"%s: cannot connect to\
           HyperLook\n",
           argv[0]);
       exit(1);
   }

   /* main program */
   ....

   hl_stop();
   exit(0);
}
```

ENVIRONMENT

The HyperLook client interface is build on top of the NeWS wire service. The following environment variables are significant.

HLHOME
The HyperLook home directory.

DISPLAY
The display to which the client connects.

FILES

include/hyperlook.h
The include file containing all the HyperLook client interface definitions.

SEE ALSO

hyperlook(1), **hl_register(3)**, **hl_listen(3)**.

DIAGNOSTICS

`hl_start()` starts HyperLook. It does nothing if HyperLook is running.

libhl.a(3)

NAME

libhl.a

HyperLook client interface library.

DESCRIPTION

The C library **libhl.a** can be linked with any C program to communicate with HyperLook. It contains library routines which are defined in the file **include/hyperlook.h**.

See **hl_start(3)** and **hl_listen(3)** for information on how to use the main functions in this library.

EXAMPLES

To compile a C program called **try.c** you need to specify some include flags and library flags in the makefile:

```
CFLAGS= -I$(HLHOME)/include \
   -I$(OPENWINHOME)/include/wire \
   -I$(OPENWINHOME)/include/NeWS \
   -I$(OPENWINHOME)/include
LDFLAGS= -L$(HLHOME)/lib -lhl

cc -o try try.c $(CFLAGS) $(LDFLAGS)
```

The three OpenWindows include directories are not all necessary. This combination is chosen because it works for all OpenWindows versions.

FILES

include/hyperlook.h
The include file containing all the HyperLook client interface definitions.

lib/libhl.a
The library file containing the HyperLook client interface.

SEE ALSO

hyperlook(1), **hl_start(3)**.

Index

A

Again
function key 3-12, 3-35
Align
graphics editor font menu 4-36
graphics editor menu 4-37
graphics editor menu choices 4-37
stack editor menu choices 5-39
aligning objects 5-18
alternating drawing buttons 6-10
animation 7-31
application resources 3-24
Arrange
graphics editor menu 4-36
arrow keys
and text editing 3-15
auto indentation
text property 6-16

B

BackGround
properties 5-27
Background Info
stack editor menu choice 5-38
background layer 5-25
backgrounds 5-24
using 5-26
black and white stacks 5-32
Browse Mode
stack editor menu choice 5-37
browse mode
cursor 5-3
getting into 5-3
stack menu 3-36
Button
alternating drawing buttons 6-10
checkboxes 6-10
drawing buttons 6-10
properties 6-8
push buttons 6-9
transparent buttons 6-9
using 3-13

C

Card
properties 5-25
stack editor menu 5-38
Card Info
stack editor menu choice 5-38
card layer 5-25
cards 5-24
adding to stack 5-25
copying 5-26
deleting from stack 5-25
selecting background 5-26
stacks and 3-7
Center
graphics editor menu choice 4-36
text adjustment property 6-16
Checkbox
using 3-13
classes 7-8
browsing 7-10
Clear
graphics editor menu choice 4-35
stack editor menu choice 5-36
stack menu choice 3-36
client interface 8-2
addressing 8-12
allocating hl_any data structures D-3
allocating memory D-2
cleaning up 8-14, D-21
compiling 8-14, 8-18, D-24
connecting to HyperLook 8-4, D-21
connecting to stacks 8-5, D-19
data structures 8-7, D-3
debugging 8-11, D-21
existence of objects D-6
flushing data D-7
getting object attributes 8-18, D-8
hamburger example 8-8
handling messages 8-5, D-10
hiding stacks D-19
hl_any data structures 8-7
hl_verbose variable D-21
hyperlook.h 8-7, 8-10
include files 8-10

initializing 8-4, 8-11, D-21
library libhl.a D-24
message handlers 8-13
messages 8-2
monitoring messages D-21
notification 8-12
overview 8-4
PostScript D-16
printing hl_any data structures D-15
receiving messages 8-5
registering handlers 8-12, D-10
resource directories D-14
sending messages 8-6, D-17
showing stacks D-19
skeleton program 8-4
UserName example 8-15
client interface routines
hl_alloc D-2
hl_any 8-6, 8-7, D-3
hl_connect 8-5, 8-10, 8-11, D-19
hl_constant D-2
hl_destroy D-2
hl_exists D-6
hl_flush D-7
hl_flush_input D-7
hl_free D-2
hl_get 8-18, D-8
hl_hide 8-10, 8-14, D-19
hl_listen 8-5, 8-10, 8-12, 8-18, D-10
hl_new_array D-3
hl_new_array_body D-3
hl_new_boolean D-3
hl_new_name D-3
hl_new_null D-3
hl_new_number D-3
hl_new_string 8-13, D-3
hl_new_string_body D-3
hl_path 8-5, 8-10, 8-11, D-14
hl_print D-15
hl_ps D-16
hl_put D-8
hl_register 8-12, D-10
hl_register_any D-10
hl_register_ioerror D-10
hl_register_quit D-10
hl_register_timeout 8-12, D-10
hl_send D-17
hl_send0 8-6, 8-13, D-17
hl_show 8-5, 8-10, 8-11, D-19
hl_start 8-4, 8-10, D-21
hl_stop 8-5, 8-10, 8-14, D-21
hl_use D-2
hl_verbose variable 8-11, D-21
clipboard
multiple pages 3-18
using 3-18
Clipped Group
graphics editor menu choice 4-37
Close
stack menu choice 3-36
close button
in stacks 3-11
color
black and white stacks 5-32
changing 3-20
color pallet 3-20
OpenWindows cubesize 3-21
problems with 2-11
user colors 3-20
ColorSelect
and the ColorPallet 3-17
properties 6-25
using 3-17
console
logging messages 7-15
messages on 2-8, 3-33
constraint editing 5-17
converting drawings
drawps C-2
Copy
function key 3-18, 3-35, 4-22, 5-20
graphics editor menu choice 4-35
stack editor menu choice 5-36
stack menu choice 3-36
copy
using the clipboard 3-18
Copy Card
stack editor menu choice 5-38
Copy Object as Drawing
stack editor menu choice 5-36



Copy Object Props
stack editor menu choice 5-36
Copy Stack as Drawing
stack editor menu choice 5-36
stack menu choice 3-37
copying
cards 5-26
object properties 5-22
objects 5-20
objects and stacks as drawings 5-23
Create from Class
stack editor menu choice 5-39
creating new objects 5-12
-cubesize
OpenWindows option 2-11
cursors
in edit mode and browse mode 5-3
Cut
function key 3-18, 3-35, 4-23, 5-21
graphics editor menu choice 4-35
stack editor menu choice 5-36
stack menu choice 3-36
cut
using the clipboard 3-18

D

Delete Card
stack editor menu choice 5-38
deleting objects 5-21
documentation browser 7-10
double click
using the mouse 3-4
Double Poly Points
graphics editor menu choice 4-37
Draw
button in system stack 2-5
drawing buttons 6-10
drawings
creating from objects and stacks 5-23
mailing to other users 2-10
drawps
Unix command 4-7, C-2
DrawTool
graphics editor object 4-2
properties 6-26
Duplicate
graphics editor menu choice 4-35
stack editor menu choice 5-36
duplicating objects 5-21

E

Edit
graphics editor menu 4-35
stack editor menu 5-36
stack menu 3-36
Edit Mode
stack menu choice 3-37
edit mode 5-3
cursors 5-3
customizing 5-33
getting into 5-3
grid 5-33
invisible objects 5-34
stack menu 5-36
editing
customizing 5-33
Editor Info
stack editor menu choice 5-37
Encapsulated PostScript 4-7, 4-30
End
function key 3-7, 3-35
environment
DISPLAY variable 2-12
HLHOME variable 2-11
OPENWINHOME variable 2-11
OPENWINOPTIONS variable 2-11
path variable 2-11
setting up 2-10
errors
PostScript trace 7-33
syntax errors 7-34
turning on or off 3-33
Exit
button in system stack 2-5
exithyperlook
Unix command C-4
exiting HyperLook 2-6

F

Field
  numeric fields 6-13
  properties 6-12
  using 3-14
File
  graphics editor menu 4-34
file dialog 3-27
  changing directories 3-27
  opening files 3-27, 3-28
  saving files 3-29
  selecting a director 3-30
  short cuts 3-30
file manager
  OpenWindows tool 2-9
Fixed position
  Stack property 5-9
Flip
  graphics editor menu choices 4-38
Flip Poly and Spline
  graphics editor menu choice 4-37
Focus Number
  property 6-13
Font
  function key 3-35
  graphics editor font menu 4-35
fonts
  changing list of 3-31
Front
  function key 3-10
front to back order of objects 5-18
Front/Back
  stack menu choice 3-36
function keys 3-35

G

graphics
  on a stack 6-10
  pasting into a stack 5-22
graphics editor
  aligning objects 4-16
  arrow heads 4-19
  changing the size of objects 4-13
  clipped objects 4-32
  constraint editing 4-15
  converting image formats 4-29
  copy 4-22
  creating multiple objects 4-12
  creating objects 4-11
  cut 4-23
  deleting objects 4-23
  duplicating objects 4-22
  Encapsulated PostScript 4-6
  Encapsulated PostScript objects 4-30
  fill color 4-17
  font of text objects 4-24
  free hand drawing 4-27
  front to back order 4-20
  grid 4-15
  grid lines 4-15
  grouping objects 4-20
  holes in objects 4-32
  hyperdraw Unix command C-8
  icons 4-4
  image filters 4-29
  images 4-29
  line color 4-17
  line width 4-18
  loading drawings 4-6
  magnification 4-5, 4-8
  menus 4-34
  moving around 4-8
  moving objects 4-13
  opening drawings 4-6
  paste 4-22
  pie objects 4-25
  points in polygons and splines 4-27
  polygons 4-26
  printing drawings 4-7
  rotating objects 4-14
  rounded corner rectangles 4-26
  saving drawings 4-6
  saving Encapsulated PostScript 4-6
  scaling text 4-25
  selecting multiple objects 4-10
  selecting objects 4-9
  sizing objects 4-13
  smooth curves 4-28
  snap to grid 4-15
  solid objects 4-31



splines 4-28
starting 4-4
text objects 4-24
tool pallet 4-5
triangle, creating 4-31
undo 4-23
using the clipboard 4-22
zooming 4-8
Grid
graphics editor menu 4-36
Group
graphics editor menu choice 4-37

H

Help
function key 3-3, 3-35, 5-29
help
defining 5-29
editing help text 5-30
locating help text 5-30
using help 3-3
hidestack
Unix command C-12
hiding stacks
hidestack Unix command C-12
hlpath
Unix command 3-24, 5-32, C-5
hlpsh
Unix command 7-35, C-6
hlsend
Unix command 7-35, C-7
Home
function key 3-7, 3-35
stack menu choice 3-36
home card 3-7
hyperdraw
Unix command 4-4, C-8
HyperLook
exiting 2-6
home directory 2-3
icons 2-9
installing 2-3
quitting 2-6
starting 2-2
.hyperlook
parameter file 3-31
hyperlook
Unix command 2-3, C-9
hyperlook.h
client interface include file 8-7

I

Import
graphics editor menu choice 4-34
-includedemos
OpenWindows option 2-11
Info
button in Introduction stack 2-4
stack editor menu choice 5-39
.init.ps
PostScript file 2-8
input focus
order 6-13
install_hyperlook
Unix command 2-9, C-9
invalid names error 7-34
invisible objects 6-6
editing 5-34

L

layer
object property 6-5
Left
text adjustment property 6-16
levers 6-11
List
exclusive 6-18
non-exclusive 6-18
properties 6-18
using 3-16
loading a stack 3-8
logging
switching on or off 3-33
.login
unix file 2-10
Look
stack menu 3-36
lpr
Unix command 3-32

M

mail tool
    OpenWindows tool 2-10
memory
    usage 2-6
menus
    accelerators 3-6
    function keys 3-6
    graphics editor menu 4-34
    in edit mode 5-4
    New, installing sub menus 5-13
    stack menu 3-5, 3-9
    stack menu in browse mode 3-36
    stack menu in edit mode 5-36
    using 3-5
messages 7-12
    hierarchy 7-13
    hlsend Unix command C-7
    logging 7-15
    logging, switching on or off 3-33
mono
    resource directory 5-32
monochrome stacks 5-32
mouse
    use of buttons 3-4
    using 3-4
moving objects 5-16
moving stacks 3-9
multiple screens
    running HyperLook on 2-12
multiple state buttons 6-10

N

New
    graphics editor menu choice 4-34
    stack editor menu 5-38
New Background
    stack editor menu choice 5-38
New Card
    stack editor menu choice 5-38
new objects 5-13
Next
    stack menu choice 3-36
None
    text adjustment property 6-16
-nosunview
    OpenWindows option 2-11
Nudge
    graphics editor menu choices 4-35
    stack editor menu choices 5-37
numeric fields 6-13

O

Object
    stack editor menu 5-39
object name
    object property 6-4
object warehouse 5-13
    customizing 5-34
Objects
    stack editor menu 5-39
objects
    aligning 5-18
    changing size 5-17
    constraint editing 5-17
    copying 5-20
    copying properties 5-22
    deleting 5-21
    duplicating 5-21
    front to back order 5-18
    moving 5-16
    pasting 5-20
    pasting properties 5-22
    selecting 5-15
    selecting multiple 5-15
Open
    button in system stack 2-5
    function key 3-11, 3-35
    graphics editor menu choice 4-34
    stack editor menu choice 5-37
opening a directory 3-30
opening files 3-27, 3-28
opening stacks 3-8
    * in pulldown menu 3-9
OpenWindows
    color cube size 2-11
    function keys 3-35
    mutiple screens 2-12
    options 2-11



starting 2-8
visual modes 2-12

P

pageview
OpenWindows PostScript previewer 3-32
Paste
function key 3-18, 3-35, 4-22, 5-20
graphics editor menu choice 4-35
stack editor menu choice 5-36
stack menu choice 3-36
paste
using the clipboard 3-18
pasting graphics
in edit mode 5-22
pasting object properties 5-22
pasting objects 5-20
pasting text
in edit mode 5-22
PgDn
function key 3-8, 3-19, 3-35
PgUp
function key 3-8, 3-19, 3-35
pop up menus 3-5
PostScript
converting drawings with drawps C-2
documentation 7-7
hlpsh PostScript shell 7-35
hlpsh Unix command C-6
language 7-5
manuals 7-7
postfix notation 7-5
Previous
stack menu choice 3-36
Print
graphics editor menu choice 4-34
printing
class documentation 7-11
drawings from the graphics editor 4-7
previewing the output 3-32
specifying a printer 3-32
programming clients 8-2
properties 6-15
BackGround 5-27
Button object 6-8
Card 5-25
color 6-7
ColorSelect object 6-25
DrawTool object 6-26
editing generic 6-4
Field object 6-12
focus number 6-13
font 6-7
generic 6-3
glue 6-5
layer property 6-5
List object 6-18
object name 6-4
position and size 6-6
PullDown object 6-20
Slider 6-23
Stack 5-8
stack editor 5-33
visible 6-6
pull right menus 3-5
in pulldown objects 6-21
PullDown
properties 6-20
pull right menus 6-21
using 3-17
push buttons 6-9

Q

quitting HyperLook 2-6

R

Receiver 7-14
redrawing stacks 3-12
Reduce Poly Points
graphics editor menu choice 4-37
Refresh
stack menu choice 3-36
Request
button in Introduction stack 2-4
requesting more information 2-4
resizable stacks 5-11

resize corner
adding to a stack 5-11
in stacks 3-11
resizing stacks 3-11
resource directories
mono 5-32
specifying using hlpath C-5
resources
adding directories 3-24
locating 3-25
mono, sub directory 5-32
removing directories 3-24
resource manager 3-23
Retained
Stack property 5-9
Revert
graphics editor menu choice 4-34
stack editor menu choice 5-37
Right
text adjustment property 6-16

S

Save
graphics editor menu choice 4-34
stack editor menu choice 5-37
Save as...
graphics editor menu choice 4-34
stack editor menu choice 5-37
Save EPS...
graphics editor menu choice 4-34
SaveBehind
Stack property 5-9
saving files 3-29
Scale
graphics editor menu choice 4-38
Script
stack editor menu choice 5-39
scripting 7-9
Action method 7-16
addressing objects 7-20
animating slider 7-31
Button Action 7-16, 7-18
Button methods B-4
CardObject methods and variables B-4
celsius slider example 7-28
Checkbox Action 7-25
class tree 7-8
classes 7-8
ColorSelect methods B-6
communicating Sliders 7-23
custom methods 7-26
customizing help 7-24
DrawTool methods B-5
editing scripts 7-3
errors 7-33
example of a script 7-4
examples 7-28
fahrenheit slider example 7-28
Field Action 7-17
Field methods B-5
getting the value of an object 7-24
go to card script 7-22
Hello world example 7-4
hiding a stack 7-18
hiding the current stack 7-18
iconifying a stack 7-19
including scripts 7-26
invalid names error 7-34
linking buttons to cards and stacks 7-21
List methods B-5
mail messages example 7-30
message formats 7-19
message hierarchy 7-13
messages 7-12
methods and messages 7-12
methods and variables, overview B-2
next card script 7-21
OnClose script 7-25
OnOpen script 7-25
printing class documentation 7-11
PullDown Action 7-18
PullDown methods B-5
Receiver 7-14
Sender 7-14
sending messages 7-19
setting the value of a slider 7-20
setting the value of an object 7-22
SetValue 7-22
showing a stack 7-18

Slider Action 7-16
Slider methods B-5
Slider, setting the value of 7-23
Stack methods B-3
stacks, showing and hiding 7-18
super classes 7-9
syntax errors 7-34
Target 7-14
Text methods B-5
Text, setting the value 7-23
values, of objects 7-22
writing scripts 7-16
scripting methods and variables 7-25
Action 7-16
Button method B-4
ColorSelect method B-6
DrawTool method B-5
Field method B-5
List method B-5
PullDown method B-5
Slider method B-5
Text method B-5
ClientSend, operator 7-19, B-2
Color, CardObject variable B-4
ConvertValue, Slider method 7-28, B-5
Damage, CardObject method B-4
DEBUG, operator 7-18, B-2
DeIconifyStack, Stack method B-3
Draw, CardObject method B-4
FillColor, CardObject variable B-4
FindObject, operator 7-20, B-2
FindStack, operator 7-20, B-2
FlipIconifyStack, Stack method B-3
FontName, CardObject variable B-4
FontSize, CardObject variable B-4
GetValue, DrawTool method B-5
GoHomeCard, Stack method B-3
GoLastCard, Stack method B-3
GoNextCard, Stack method B-3
GoPreviousCard, Stack method B-3
GotoCard, Stack method 7-22, B-3
GotoNextCard, Stack method 7-21
Height, CardObject variable B-4
Hide, CardObject method 7-25, B-4
HideStack, operator B-2
IconifyStack, Stack method B-3
IncludeScript, operator 7-26
LookupObject 7-20
LookupObject, operator B-2
Move
Stack method B-3
Move, CardObject method B-4
MyClient, CardObject variable B-4
MyStack, CardObject variable 7-19, B-4
OnAscii, CardObject method B-4
OnClose, CardObject method 7-25, B-4
OnHelp, CardObject method 7-24, B-4
OnMouse, CardObject method B-4
OnOpen, CardObject method 7-25, B-4
ParentSend, operator 7-19, B-2
Path, Button method B-4
Press, Button method 7-21, B-4
Reshape, CardObject method B-4
Select, List method B-5
SelectAll
Field method B-5
Selected
List method B-5
Selection
Field method B-5
Text method B-5
send operator 7-24
Send, operator 7-19, B-2
SetRange, Slider method B-5
SetValue
Button method B-4
ColorSelect method B-6
DrawTool method B-5
Field method B-5
List method B-5
PullDown method B-5
Slider method B-5
Text method B-5

SetValue, Slider method 7-20
Show, CardObject method B-4
ShowMessage operator 7-35
ShowMessage, operator 7-17, B-2
ShowStack, operator B-2
SystemDate, operator 7-25
ToBack, Stack method B-3
ToFront, Stack method B-3
Value, CardObject variable B-4
Width, CardObject variable B-4
WordAction, Text method B-5
Scrollbar
using 3-14
scrollbar
as a Slider object 6-23
scrolling
setting the speed 3-32
scrolling list
using 3-16
Select
graphics editor menu choices 4-35
stack menu choices 3-37
Select All
stack editor menu choice 5-36
selecting a directory 3-30
selecting colors
using ColorSelect objects 3-17
selecting multiple objects 5-15
selecting objects 5-15
Sender 7-14
sh, shell scripts with hlpsh 7-35
shape of stacks
changing 5-10
Show Warehouse
stack editor menu choice 5-38
Show, CardObject method 7-25
showing stacks
showstack Unix command C-12
showstack
Unix command C-12
Size
graphics editor font menu 4-36
sizing objects 5-17
Slider
properties 6-23
using 3-14
Slider object
scrollbar 6-23
Solid Group
graphics editor menu choice 4-37
Special
graphics editor menu 4-37
Stack
properties 5-8
stack editor menu 5-37
stack
BackGroundProps 5-27
ButtonIdeas 5-14
ButtonProps 6-8
CardProps 5-25
ClipBoard 3-18
ColorPallet 3-20
ColorSelectProps 6-25
Confirm 5-7
DocBrowser 7-10
DrawToolProps 6-26
ExitHyperLook 2-6
FieldProps 6-12
Help 1-7, 3-3, 5-29
HelpProps 5-30
HyperDraw 4-4
Introduction 2-2
ListProps 6-18
Notepad 1-4
ObjectProps 5-2, 6-3
OpenDir 3-30
OpenFile 3-27
OpenStack 3-8, 5-5
PostScriptError 3-34, 7-33
PrinterQueue 3-32
PullDownProps 6-20
ResourceMgr 3-23
SaveFile 3-29
SaveStack 5-6
ScriptProps 7-3
SliderProps 6-23
StackEditorProps 5-33
StackMgr 3-22
StackProps 5-8
system 2-5
SystemProps 3-31
SystemStatus 2-6



TextProps 6-15
Untitled 5-5
Warehouse 5-13
Stack Info
stack editor menu choice 5-37
stack layer 5-25
stack manager 3-22
stack menu 3-5, 3-9
in browse mode 3-36
in edit mode 5-4, 5-36
Stack Script
stack editor menu choice 5-37
stacks
black and white 5-32
bringing to the front 3-10
cards and 3-7
cards, adding and deleting 5-25
changing the shape of 5-10
close button 3-11
creating 5-5
creating new style 5-34
creating object warehouse 5-34
designing shape of 5-10
designing your own 5-2
editing 5-5
editing help text 5-31
editor properties 5-33
iconifying 3-11
layers 5-25
mailing to other users 2-10
making resizable 5-11
monochrome 5-32
moving 3-9
moving in edit mode 5-3
new 5-5
opening 3-8
redrawing 3-12
resize corner 3-11
resizing 3-11
reverting 5-6
saving 5-6
structure of cards and backgrounds 5-24
throwing away changes 5-7
Untitled, creating copy of 5-5
zapping from memory 3-22
zapping from the screen 3-12
starting HyperLook
hyperlook Unix command C-9
.startup.ps
PostScript file 2-8
Style
graphics editor font menu 4-35
sub classes 7-9
super classes 7-9
switches 6-11
syntax errors 7-34
System
stack editor sub menu 5-39
system colors 3-20
system properties 3-31
system resources 3-24

T

Target 7-14
Text
graphics editor menu 4-35
properties 6-15
using 3-14
text
editing 3-14
editing keyboard commands 3-15
text editing
key board commands 3-15
text justification
Text property 6-16
Text object 6-15
To Back
graphics editor menu choice 4-36
stack editor menu choice 5-39
To Front
graphics editor menu choice 4-36
stack editor menu choice 5-39
Tools
pulldown menu in system stack 2-5
transparent buttons 6-9
trash
stack manager 3-22
trouble shooting 2-8
Alan Turing 1-11
The Turing Institute 1-11



typing
    input focus 6-13

U

Undo
    function key 3-35, 4-23
    graphics editor menu choice 4-35
Ungroup
    graphics editor menu choice 4-37
Unix commands
    drawps 4-7, C-2
    exithyperlook C-4
    hidestack C-12
    hlpath 3-24, 5-32, C-5
    hlpsh 7-35, C-6
    hlsend 7-35, C-7
    hyperdraw 4-4, C-8
    hyperlook 2-3, C-9
    install_hyperlook 2-9, C-9
    showstack C-12
Untitled stack
    customizing 5-34
user colors 3-20
user resources 3-23
.user.ps
    PostScript file 2-8

V

View
    graphics editor menu 4-36

W

warehouse 5-13
Wrap
    text adjustment property 6-16

X

.xinitrc
    OpenWindows initialization file 2-8

Z

Z2img
    image decompression filter 4-30
Zap
    stack menu choice 3-12, 3-36
zapping stacks 3-12
zapping stacks from memory 3-22
Zoom
    graphics editor menu choices 4-36

