HyperNeWS1.4 User Guide

The Turing Institute

September 1990

HyperNeWS was written by Arthur A. van Hoff. The graphical design was masterminded by Frank Bollen. The manual was written from the contributions of Danny Pearce, Jim Rudolf, Arthur van Hoff, Suhayya Abu-Hakima, Sunil Sharma and Tim Niblett. Special thanks to Douglas Scoular for help with distribution of HyperNeWS.

We would also like to thank all the users of HyperNeWS who have provided useful feedback.

The General Manager
The Turing Institute
George House
36 North Hanover Street
Glasgow G1 2AD
United Kingdom
Tel: +44 41 552 6400
Fax: +44 41 552 2985
E-mail: newsdev@turing.ac.uk

Contents

List of Figures

1 Introduction

1.1 Contents of this Document

Welcome to the HyperNeWS User Guide. The purpose of this document is to help you get started with HyperNeWS. A companion document, the HyperNeWS Reference Manual, discusses advanced topics and some reference material you may find useful later. This User Guide consists of:

- Section 1, "Introduction," is the section you are reading now.
- Section 2, "Philosophy," gives a short discussion on the philosophy and some of the HyperNeWS design decisions.
- Section 3, "Architecture," talks about the structure of HyperNeWS.
- Section 4, "Getting Started," (finally!) gets down to actually doing something with HyperNeWS.
- Section 5, "Stack Descriptions," describes the standard HyperNeWS stacks that you'll use most often.
- Section 6, "System Objects," discusses the default objects, such as buttons and menus, that come with HyperNeWS.
- Section 7, "Advanced Editing," goes into more detail and shows you how to make customized stacks.
- Section 8, "Message Passing," shows you how system objects, cards, and stacks communicate with each other.
- Section 9, "Writing HyperNeWS Methods," introduces you to PostScript programming in order to give your stacks added functionality.

If you are anxious to get started in HyperNeWS and you don't want to read about the philosophy or structure of HyperNeWS, you may want to proceed directly to Section 4, and come back to the other sections later.

The companion document, the HyperNeWS Reference Manual, contains sections that discuss writing advanced HyperNeWS methods, creating your own object types, and interfacing with client programs. It also contains an exhaustive list of the HyperNeWS system methods and instance variables...for those enquiring minds that need to know.

At some point you may notice that this document is remarkably similar to the documentation online under HyperNeWS. There is good reason for this, as both versions are generated from the same source. So if you are using HyperNeWS and you don't have this guide nearby, you can find the same information in the

Hyperman stack on the system. References to other sections or figures (such as, "See Section 2") will appear in red online, and double-clicking on one will take you to that particular part of the manual.

HyperNeWS version 1.4 has been placed on various anonymous FTP sites in the United States and Europe. It is also available via mail server in the United Kingdom. For further information, please contact the Turing Institute at the e-mail or postal address listed on page 2.

HyperNeWS is the copyright of The Turing Institute. It is free to be used and modified for academic, research, and personal endeavors. A brief acknowledgement is required in any research paper or other publication where this software has made a significant contribution. The Turing Institute offers hardcopy manuals, HyperNeWS on 1/4" cartridge tape (for those without network access), and consulting services. HyperNeWS may also be licensed for use in commercial products. Please contact the Turing Institute for prices and conditions.

1.2 Changes in Release 1.4

HyperNeWS1.4 has undergone some noticeable and some not-so-noticeable changes since release 1.3. Here is a short list of those changes:

- The new release requires OpenWindows2.0. In the past we have tried to maintain HyperNeWS on the available environments as long as it was practical. In this case, OpenWindows2.0 uses the olwm window manager, and Sun is no longer supporting pswm which was used in OpenWindows1.0. There are enough differences between pswm and olwm that it was not practical for us to attempt to support both versions of OpenWindows.

- The functionality of GoodNeWS has been merged with HyperNeWS. Now you will see just one style of terminal emulator, and exiting HyperNeWS can be a one-step process unlike release 1.3 where you had to exit HyperNeWS and GoodNeWS in separate steps.

- The installation process and the filesystem structure have changed. A configure script should help to make installation easier, and there is now only 1 *make install* that needs to be done, versus two in release 1.3. You can specify where you want the binaries, libraries, and fonts to go. The directories are no longer required to be something like *bin-sun4-xnews*. The filesystem has been split into a source/binary style, so you can remove the source tree after installation, and the files required to run HyperNeWS will have been installed in the binary tree.

- Some GoodNeWS applications have been ported to stacks. These include the performance monitor and the clipboard. The Clipboard stack is a multi-card clipboard that can hold 16 different pieces of information at once. The user's interface is described in Section 5.6, and the programmer's interface is described in the Reference Manual. As of yet, we have not ported the LaTeX previewer, *dvinews*.

- The close button in the upper-right corner of some stacks will now iconify the stack. The icons are themselves small stacks, and you move them around like you would any stack, using the AdjustButton. Click on the icon once with the PointButton to open the application again.

- Every stack now has a resizable attribute, which is initially set to true. Along with that are values that let you specify the minimum allowed width and height of the stack when a stack is resizable. If you try to make a stack smaller than it is allowed, the rubber band box will not get any smaller than its minimum allowed size.

- When GoodNeWS went away, so went the GoodNeWS background menu. In a step towards better integration with OpenWindows, you will get the OpenWindows menu when you click the MenuButton over the background if the system stack is in browse mode. If the system stack is in edit mode, you'll see the edit menu as before.

- To exit HyperNeWS, you have two choices. If you select *Exit* from the OpenWindows menu, you will exit the entire window system. But if you click on the *Exit HyperNeWS* system button, you will leave HyperNeWS but remain in OpenWindows. This is very convenient if you are developing an application and you want to restart HyperNeWS without having to restart OpenWindows too.

- The documentation has been split into two parts, the User Guide and the Reference Manual. The former presents introductory topics, while the latter has more advanced material. It is not possible to generate a hardcopy version of the online manual, but they are available from the Turing Institute at minimal cost.

- The process of saving stacks and drawings as ASCII files has been changed so that lines will not exceed 80 characters and the order in which objects are saved will be consistent. In the past, some long lines would be truncated if you tried to mail a stack. These changes will make it possible to control stack development using SCCS. Unless you actually look at the saved stack file, you probably won't notice this change.

- The way Canvas objects are drawn and updated has been made more straightforward. This may affect your existing stacks that have canvases. See the Reference Manual for a description of how Canvas object scripts should be structured.

- All stacks have been modified so they will look better when HyperNeWS is viewed on a monochrome display.
- There are a few small bugs and quirks to be aware of:
  - HyperDraw may have trouble importing some EPSF drawings.
  - HyperNeWS will not exhibit blinding speed on a 3/50 with only 4MB. In fact, the terminal emulator may give you fits.
  - OpenWindows2.0 beeps once as it is starting up. This is normal. It does not mean something has gone wrong.
  - Sometimes the cursor, represented as a HyperNeWS hand or an OpenWindows arrow, does not go away when you exit OpenWindows. It will remain on your screen until you reboot or restart OpenWindows.
  - Sun has acknowledged a problem with input focus when you have both NeWS and X applications running. So even if you have follow-mouse for an OpenWindows focus setting, you may sometimes have to click in the window to get the focus to change when you switch between X and NeWS applications.
  - Depending upon your hardware, you may get a message when you start OpenWindows2.0 saying something about needing a GFX revision of SunOS in order to work properly. We haven't noticed any unusual behavior when this message appears.
- Despite the few glitches mentioned above, numerous bugs have been fixed, many of which were found by users. Keep those comments and suggestions coming!

2 Philosophy

2.1 Background

HyperNeWS development began in October 1987 at the Turing Institute, a not-for-profit organization involved in research and training in Artificial Intelligence and related disciplines. We had a requirement for a user interface development tool as a vehicle for the implementation of knowledge-based systems. We had previously developed a system called *PanelTools* which was SunView-based and had a Prolog programming interface. We found this system rather limited for a variety of reasons, and were looking to develop something more powerful.

Two developments at around this time fired us with the enthusiasm to develop HyperNeWS. The first was Sun's NeWS windowing system, which offered a giant leap forward over SunView or the then-emerging X-Windows, and the second was Apple's HyperCard, which provided an interface tool which could be used without programming.

There seemed an opportunity to combine the NeWS system and the idea of open access computing to provide a genuinely useful tool with greater flexibility than HyperCard. We believe that HyperNeWS is such a tool.

The programming and detailed design of HyperNeWS was done by Arthur van Hoff of the Turing Institute, and the initial version (1.0) was completed in April 1988. With the introduction of X/NeWS, the removal of many bugs from NeWS and the experience gained with version 1.0, a rewrite and simplification was done in June 1989. Version 1.3 was virtually the same as its predecessor, with minor revisions to conform to the merged X/NeWS server. The current version, 1.4, has been adapted to run with OpenWindows2.0 and the olwm window manager. Other additions in this version can be found in the Introduction (Section 1.2).

2.2 The Goals of HyperNeWS

Our goals in developing HyperNeWS were:

- To develop a stand-alone interface system.
- To provide a system which can be programmed by direct manipulation.
- To provide high-level interfaces to a variety of languages.
- To provide for transfer of interfaces and interface components between different applications and users in a transparent fashion.
- To provide extensibility without sacrificing portability and reusability.
- To provide full online documentation of HyperNeWS in HyperNeWS.

Let us try to explain the rationale behind these goals:

A stand-alone system

Interface toolkits such as SunView, TNT, or Xtk provide a set of language calls. They do not provide a stand-alone environment in which significant work can be done without programming. HyperCard does provide such an environment, although it is difficult to interface to external languages such as C because of limitations in the Macintosh operating system. A stand-alone environment is essential if the system is to have a gradual "learning curve" for new users. Furthermore, many straightforward applications, such as simple help systems should not require programming in the traditional sense. Another advantage of a stand-alone system which has its own programming language and communication mechanism is that it allows different processes, perhaps written in different languages, to communicate with no overhead on the application side.

Direct manipulation programming

A large amount of the work in designing user interfaces can be done by direct manipulation — drawing, dragging, and sizing objects with the mouse and keyboard. This is preferable to the written specification required in SunView, TNT, or Xtk. To be effective at such manipulation the whole design of the interface tool must be directed towards this end. We had previous experience at grafting a direct manipulation interface onto SunView, with limited success.

A high-level interface

Under a Unix system it is inevitable that users will require interfaces to high-level languages, such as Lisp, Prolog, or C and its variants. Our goal was to provide an abstract interface to these languages, which did not need to specify details about object placement or low-level event handling, but instead focused on the functional characteristics of interfaces. For example, help information about an application should be programmable in the interface, and different types of objects should be capable of fulfilling the same interface functions at different times without having to rewrite the client software.

Component reusability

HyperNeWS is a hybrid object-oriented system in the sense that the basic object types are created with a classing mechanism similar to that of Smalltalk or C++, but user-defined types are created from object templates or prototypes. The use of prototypes has major advantages. First, it is usually the case that a large variety of small variants to standard objects are required. The standard classing mechanism will then produce a large class hierarchy. If I want to send someone an object I have created I have to send the whole class hierarchy. This is usually not possible. Prototypes can be seen as a standard object, with a very small amount of additional or changed behavior – the grain-size is more appropriate. Second, prototypes as deformations of a fixed class hierarchy support the notion of object persistence. A persistent object is one which exists beyond its incarnation in a single process. The most common example in everyday life is the data maintained in databases. Persistent objects in HyperNeWS are stored in ASCII files. It would be extremely cumbersome if every object had to be stored together with its complete class hierarchy. Persistence is the mechanism which allows users to share HyperNeWS objects.

Extensibility

We have talked about the extensibility of HyperNeWS provided by prototypes and persistence. Another aspect of extensibility is the capacity to integrate with existing applications. This is done in HyperNeWS via a special object type, the canvas. Alien applications, obeying their own rules of input with interfaces to any external process can coexist with HyperNeWS interfaces. A simple discipline allows programmers to interface to HyperNeWS services. The HyperNeWS terminal emulator and the HyperDraw stack are examples of applications with canvas objects.

Online documentation

Hypermedia has proven to be a useful educational tool, and because of the structure of HyperNeWS, a hypermedia manual was an attractive idea. The HyperNeWS documentation is kept in an SGML format, and both the online and hardcopy versions of the documentation are generated from the one file. This means that only one version of the manual needs to be maintained, and that we can take advantage of the SGML linking facilities to provide hypermedia links between sections and figures in the online manual.

2.3 Points of Interest

2.3.1 Why PostScript?

The first decision that we took was to implement HyperNeWS entirely in PostScript inside the NeWS server. In some ways this was an absurd decision. Sun, whose engineers designed NeWS, try to persuade you to do as little as possible in PostScript. There are several official reasons why one should believe them:

- PostScript is an interpreted language, and is rather slow. C is much faster for most operations.

- PostScript is not very memory efficient. Large applications can use a lot of space.
- For complicated applications where an external client will do most of the calculation, if a lot of work is done in PostScript it will be necessary to maintain two copies of critical data structures. This will complicate the programming.

There is also an unofficial reason. The NeWS implementation has proven to be somewhat unstable. The C compiler is much more reliable — use it!

If we break these reasons down they say that PostScript is too slow and requires too much memory to do the job effectively. This is an empirical question, and we have found that it is neither too slow, nor too wasteful of memory. This is of course something that can only be found out by trial and error.

There are some very compelling reasons for sticking with PostScript. The main one is that once you start using C, it will become necessary for users to learn about your data structures and to modify your code. This would greatly reduce the appeal of the system. Furthermore, to create the same functionality, a little analysis shows that it would be necessary to recode a substantial part of the PostScript interpreter — not a pleasant undertaking.

The fact remains that using PostScript has limited our ambitions to some extent, particularly in the realm of text handling.

2.3.2 Data Structures in PostScript

PostScript has a reputation as a low-level language, akin to infix assembly code. Nothing could be further from the truth. PostScript is a modern language, which shares much more with Lisp than it does with assembler. We suspect that its reputation is based on the syntax of the language, rather than its semantics.

The most important feature of the language from our viewpoint is that it can read its own code. This is critical for our ability to save objects (persistence again), and to write simple PostScript interpreters for subsets of the language. It is, for example, possible to write very good error handlers, which are essential in an interactive programming environment.

The two main data structures are the list and the dictionary. Dictionaries are a form of hash table, and can be used to implement a classing and message passing mechanism very compactly. Inside HyperNeWS the main body of the message passing code requires about 350 lines of code. The whole of the core system contains around 12 thousand lines of PostScript.

2.3.3 The Scripting Language

HyperNeWS is a stand-alone system. While much of the code is written in PostScript, it could benefit from a "scripting language" in which to specify object behaviors and prototypes. We debated about how best to implement such a language. There is a considerable attraction in a simple language, such as HyperTalk, which can be learned by users with little effort. The disadvantages of this approach when working within NeWS seem to outweigh the advantages. The semantics, as opposed to the syntax, of PostScript are well suited to HyperNeWS, particularly with relation to graphics for which PostScript is designed. Any scripting language would be syntactic sugar for PostScript, and the overhead of translation between the two languages did not seem to be justifiable. Furthermore, at some stage more advanced users will need to learn PostScript to interface other programs to HyperNeWS; it seemed better to let one language suffice for all purposes.

The main issues that arise in using PostScript as a scripting language are:

- How to deal with errors. Users create and execute scripts. It is necessary that errors be caught, and explained to users.
- How can scripts be stored? The original version of HyperNeWS stored scripts as PostScript code, translated into text for editing by users. From Version 1.3, scripts are stored as text. This allows for comments to be included, and for long scripts to be stored in external files if desired.

Having said this, we have decided after further study that a scripting language would indeed be worthwhile. We have found that the translation time from scripting language to PostScript is acceptable. Also, the option of using a scripting language instead of PostScript should make scripting more attractive to most users. The *PdB* scripting language is nearing completion and should be available soon.

2.3.4 Canvases

A major difference between HyperNeWS and the various toolkits provided by Sun for X/NeWS is that objects do not in general have their own canvases. Each stack has a canvas, and the objects below the stack in the object hierarchy share this canvas. This approach has advantages and disadvantages. It is worth discussing these, as Sun took the opposite route in their designs.

The advantages in objects *not* having their own canvases are:

- Objects can be transparent without the problems caused by the lack of update events for transparent canvases.
- Non-transparent objects have a size and path which is not strictly determined by the dimensions of a canvas. This eliminates problems of stroking lines around objects.
- As objects don't need a canvas, there is no need for an event handler for each object. This makes objects simpler, and easier to modify and program.

- It is hard to make canvas-based objects persistent. For example, the shape of a canvas is difficult to translate into ASCII, and the event handlers associated with canvases would have to be saved somehow, since they must be started when the object is loaded.

The disadvantages are:

- Individual objects cannot receive events directly. The events have to be passed to the target object by an ancestor (the stack) which owns the canvas and receives the events.
- Existing programs may need canvases, and therefore cannot run inside an object without a canvas. This problem has been solved by providing the system with a canvas object.
- Objects with canvases can receive more events than those provided by HyperNeWS. For example, except for canvas objects, individual objects cannot receive *Enter* and *Exit* events.

Overall we feel very strongly that the approach we have taken is correct, although it is possible that changes to the architecture of NeWS could tip the balance back the other way.

2.3.5 Message Handling

Message handling in HyperNeWS has been designed to fulfill the requirements of a user interface design tool, and has been designed for freedom from error. We considered it essential that since the user will create messages, it should not be possible for the system to get into an unrecoverable situation from the passing of messages. Subsequently, the message system has not been designed to guarantee the delivery or execution of messages.

The syntax of message passing is as follows:

```
[args] /Message Object Send
```

Notice that the arguments to the message are enclosed in an array. If the object the message is sent to cannot handle the message (e.g. it is not defined in its dictionary), the message is passed on to the parent of that object. If the stack (the end of the line) cannot handle the message it is passed to the stack's client process, if one exists.

When a message is sent to another object and the message passing mechanism is implemented in PostScript, it is not simple to do this in a error-free manner. A PostScript process has a graphics state, an execution stack, a dictionary stack, and an operand stack. When a message is sent in HyperNeWS, all execution occurs within the same process. The operand and dictionary stacks are saved to ensure that if an error occurs, execution can continue. The execution stack is not saved, since this will not get clobbered by an error. The graphics state is not saved or altered in any way.

This approach has the consequence that it is not possible for the sender of a message to know whether the message has been delivered correctly, or to know what the results were. It is possible to get information from the graphics state. This is used in the system to concatenate paths. The drawback of not saving the graphics state is that objects cannot rely on any particular state when executing.

2.3.6 Input Handling and Synchronization

NeWS is a multi-processing system. This is a potential cause of confusion to users, who will generally not want to be involved with the complexities that ensue. To overcome this potential problem, HyperNeWS has been designed to look as if there is only one thread of control. To understand how this works, we shall have to explain a little of the input management in HyperNeWS. Each canvas (i.e. stack) has an event manager process which sits in a tight loop waiting for input events – usually mouse, keyboard, or damage events. When an event is received this process handles the event, which in HyperNeWS will usually involve one or more messages being sent. The process does not execute any `pause` statements, so it cannot lose control. This means that at any one time there is at most one "HyperNeWS process" running, and it is this which allows the user to assume operation under a single process.

2.3.7 Interfacing Clients to HyperNeWS

Simple applications can be run entirely within HyperNeWS, but more complicated ones should be programmed in a more appropriate language than PostScript. This can be done via the C interface to HyperNeWS. The interface provides a data structure in C for HyperNeWS messages and objects, and for both synchronous and asynchronous communication between stacks and their clients.

The C interface is also used to build higher-level interfaces to the AI languages Prolog and Lisp. In some ways interpreted languages like these are easier to use for developing interfaces, as ideas can be tested out interactively. At the Turing Institute we have built very large interfaces to Prolog-based AI programs. Applications such as the online manuals also use a C server, in the interests of minimizing data storage on the PostScript side.

2.3.8 Import and Export

An advantage of using PostScript, as NeWS does, is that it offers the opportunity to import and export complex graphics. HyperNeWS supports the import and export of PostScript in the following ways:

- PostScript graphics in the form of Encapsulated PostScript (EPSF) drawings can be imported into HyperNeWS to form the drawn component of objects such as buttons. This lets the user draw complex and precise objects using a specialized drawing tool, and then use them inside HyperNeWS. Import and export of EPSF is important if HyperNeWS is to be used as part of a wider environment, as many packages can now export in this format. We have found that although PostScript is a de-facto standard it can still be difficult to port PostScript files to NeWS. For example, we discovered that Aldus Corporation's version of EPSF uses the `setcolor` primitive that exists on the system running the PostScript in preference to their own definition – presumably because they assumed that no one else uses it. In cases like this the ability to define new procedures provided by PostScript is working against its portability.

- All HyperNeWS objects have a *Draw* method which is used to draw objects on the screen. HyperNeWS contains a specialized PostScript interpreter written in PostScript, which can take arbitrary code and extract an EPSF description of the drawing operations executed when the code is run. The specialized interpreter acts as a form of partial evaluator of the code modulo the drawing instructions. This is an extremely useful facility as it means that a device and resolution independent snapshot can be taken of any object or group of objects. It is of course necessary to use only the pure PostScript subset of the NeWS rendering operations to get a portable drawing of the object. This facility is of enormous benefit, as it provides a completely general facility for the extraction of high-quality graphics from the screen. The current implementation is slow since it is an interpreter running in an interpreter, but it is of sufficient benefit for us to suggest that it should be included as a core component of NeWS.

2.4 The Future of HyperNeWS

HyperNeWS is undergoing continued development at the Turing Institute. Items we've been thinking about but that didn't make it into this version include:

- Improved hypertext capabilities, including text items with multiple fonts, programmable display actions and the ability to link between documents.

- A grouping capability, in terms of object layout during stack editing, and perhaps also at run-time, allowing you to send a message to a group of objects.

- Closer integration with OpenWindows. This includes using the same selection mechanism, and perhaps taking advantage of the NeWS toolkit.

As always, we do our best to ensure that your existing stacks will work under a new HyperNeWS version. However, any stacks you develop under 1.4 will not work under older versions of HyperNeWS.

We hope that with this demonstration of the possibilities of NeWS more people will be encouraged to experiment with what we believe to be the best windowing software available under X/NeWS. With direct manipulation, there should be no need for users to be restricted any longer to windowing "toolkits". Our thanks go to those at Sun who had the vision and perseverance to bring NeWS to us.

3 Architecture

3.1 The Stack Model

The design of HyperNeWS was based on the idea of a *stack*, consisting of a group of *cards*. We aren't pioneers in this respect, as the folks at Apple and Xerox (among others) have also created hypermedia tools using this concept. This is, however, the first system we know of that offers this type of hypermedia model in the general purpose Unix environment.

A HyperNeWS stack represents a program. A stack consists of one or more cards, conceptually stacked on top of each other so that only one card is visible at a time. Behind a card resides the *background*, which may belong to only one card or which may be shared between a number of cards in a stack. If you are going to have a group of cards that share some graphics or a button, putting it on a shared background means you only have to create it once.

Each card consists of any number of *objects*. Currently, HyperNeWS supports six different card object types which can reside on a card, including buttons, menus, and text fields. Depending upon your needs, objects can be assigned to either a card, a background, or the stack itself. By attaching an object to the stack, it will always be visible, no matter what card or background is currently being displayed.

3.2 The Class Hierarchy

There are two different hierarchies in HyperNeWS that define relationships between stacks, backgrounds, cards, and objects: the class hierarchy and the message hierarchy. The class hierarchy defines the relationships between objects by showing what behavior is inherited from a parent class.

HyperNeWS objects are represented as PostScript dictionaries. The class hierarchy is also stored as a set of dictionaries, as shown in Figure 1. The classes shown in gray are those that will have actual instances in the system, while the classes shown in white are *abstract superclasses*, which define methods and instance variables for their subclasses, but which are never actually instantiated.

Each class contains a set of instance variables and a set of methods. In general, it is expected that instance variables will represent values in the form of numbers, arrays, strings etc., while methods will be procedures (executable arrays). This distinction is not however enforced. An object in a given class inherits the instance variables of that class, and of all the superclasses of the given class. A subclass method or instance variable overrides the same method or variable defined in a superclass. The difference between instance variables and class methods is that each object has its own copy of instance variables, while methods are shared between all objects. This means that the user should not overwrite methods for a particular object, as this will change the behavior of the entire HyperNeWS system. If a method is required to be overwritten for a particular object, it can be included as an instance variable.

The inheritance mechanism of HyperNeWS is tree-structured – multiple inheritance is not supported. We feel it is unnecessary because of the user's ability to create prototype objects, based on the classes described below:

RootObject

Holds the basic object structure and includes routines for creating and destroying an object, and for passing messages through the class hierarchy.

GraphObject

Holds an object's graphical information and script handling, which is common to all objects.

Background

Defines routines and variables for holding, cutting and pasting objects which are common to backgrounds, stacks and cards.

Stack

Defines all editing operations, window management, and overall control. All events from NeWS are handled at the stack class level.

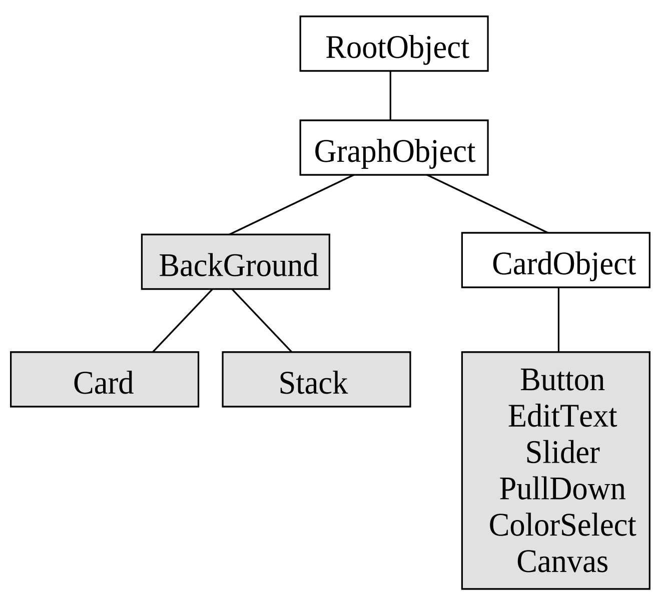

Figure 1: The HyperNeWS class hierarchy.

Card

Routines and variables specific to card objects.

CardObject

Routines and variables common to all six card object types.

Button, EditText, Slider, PullDown, ColorSelect, Canvas

Define operations specific to a particular card object type.

3.3 The Message Hierarchy

The class hierarchy in HyperNeWS is static. However, the message hierarchy is orthogonal, and determines the default manner in which messages are passed between objects. The possible parent-child relationships that can exist are shown in Figure 2. Notice that a concrete message hierarchy is strict; each object can only have one parent at run-time. The message hierarchy has two effects:

- It determines the passing of messages between objects (see Section 8).
- It determines the order of display of objects. Objects are displayed in the following order – objects displayed later overlay those displayed earlier – objects with the stack as parent, objects with a background as parent, and finally objects with a card as parent. The display order of objects at any level can be varied, as each object stores its children on a list.

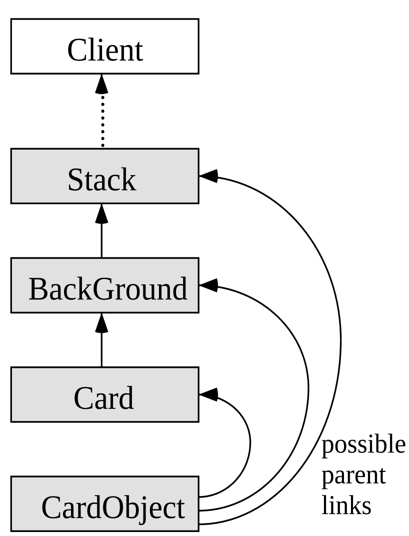

Figure 2: The run-time message hierarchy in HyperNeWS.

4 Getting Started

4.1 Starting HyperNeWS

Once your system administrator has installed HyperNeWS on your system, you can start it using the script called `hn1.4` located in `$HNHOME/share`. Even if you don't use this script, it is helpful to see what changes need to be made to your environment in order to run HyperNeWS. Once the OpenWindows and HyperNeWS banners have been displayed, you will see the HyperNeWS background and the Introduction stack, displayed in the middle of the screen. At this point you can either look through the Introduction stack, or click the left mouse button over the *Done* button. If you do the latter, the stack will go away, a group of *system buttons*, shown in Figure 3, will appear in the lower-left corner, and the Console stack (representing the Unix system console) will appear in the upper-right corner.

A piece of information that is just as important as learning how to *start* HyperNeWS is how to *exit* HyperNeWS. If you click on the *Exit HyperNeWS* system button, you will exit HyperNeWS but remain in OpenWindows. This is useful if you are developing stacks and you want to restart HyperNeWS without having to exit and restart OpenWindows. To exit HyperNeWS *and* OpenWindows, press the right mouse button over the background, which will show you the OpenWindows menu. Choosing the *Exit* menu item will exit the entire window system.

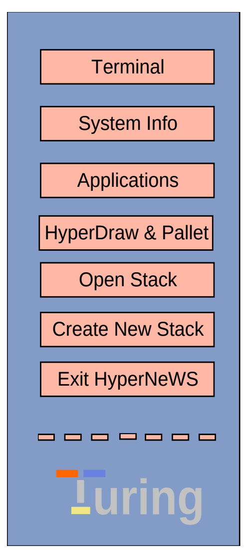

Figure 3: The HyperNeWS system buttons.

4.2 Browsing

4.2.1 Basics

HyperNeWS is split up into two distict modes: *browse* mode and *edit* mode. Browse mode is meant for using existing tools or applications. If you want to create your own stacks or change existing ones, you'll do it in edit mode. First let's look at browse mode.

The mouse has three buttons, and each one has a purpose and a name under HyperNeWS. The left button is called the *PointButton*, and it is used to point out things and select them. If you want to delete some text or resize an object, you must first select it so HyperNeWS knows what object you are talking about. The middle button is called the *AdjustButton*. It is used for moving or resizing objects once you have selected them. Finally, the right button is called the *MenuButton* because when you press it, you see a menu until you release the button.

The menus you will be seeing are *context dependent*, meaning that you may see different menus depending on what you're in the process of doing or what your cursor is over when you press the MenuButton. There are slightly different menus for browse and edit modes. You will see a different menu if you press the MenuButton while the cursor is over the background versus over an object. HyperNeWS only shows the menu items that are valid at a given point in time.

A number of the menu selections have a quick keyboard equivalent, or *shortcut*. These are shown along the right hand side of the menu. Shortcuts use two special keys which are described here:

Command Keys

> These keys are indicated as a *hash* (#) followed by a character. To enter a command key, press the given character while holding down the HyperNeWS *command key*. This key is labeled *Left* on a Sun-3 keyboard, and it is represented by a small diamond on a Sun-4 keyboard. In either case, it is found to the left of the spacebar.

Function Keys

> Displayed as 'F' followed by a numeral. These can be found above the numeral keys on the keyboard.

Figure 4 shows a menu with some keyboard equivalents.

If you press the MenuButton over a stack, you will see anywhere from two to six menu items such as #, *Edit*, and *Object*, all of which lead to different submenus. We won't refer to this top-level menu itself, but usually to the menus that you get to from it. Therefore, if we reference the # menu, we mean the menu shown in Figure 4, with such selections as *Next* and *Zap*.

The background of the screen, colored in two shades of blue and displaying the Turing and HyperNeWS logos, is called the *system stack*. It is similar to other stacks, but it has some unique behavior all its own. It is discussed further in Section 5.22.

4.2.2 The # Menu

The # menu (or *hash* menu, for lack of a better name) contains some of the more common stack commands. The first three menu items will move you around to different cards in a stack. *Home* will take you to the *home card*, or the first one in the stack, *Next* will move you to the next card in the stack, and *Previous* will take you in the other direction. The cards have a wrap-around effect, so the previous card to the home card will be the last card. *Find ...* will let you search for a string in a stack (the ellipse means you'll be asked for more information). *Top* and *Bottom* will bring the stack to the front of the rest, or send it to the back of all stacks, respectively. *Redisplay* will redraw a stack, and *Zap* will hide the stack, but it will still reside in memory.

4.2.3 The Edit Menu

The *Edit* menu and # menu are the only ones available when you are in browse mode. *Cut*, *Copy*, and *Paste* are commands used to cut and paste text in text fields. *Clear* can be used to delete text and not have it put in the clipboard. *Copy as Drawing* will copy either a HyperNeWS object or a complete stack into the clipboard as a HyperDraw drawing. From there, the drawing can be pasted into other stacks (once we learn how to edit them) or it can be pasted into the HyperDraw drawing tool.

The next group of five commands in the menu is for selecting all the text or moving the cursor around in a text field. Note that the shortcuts for the moving commands match the arrow keys on the keypad.

Finally, the last item on the menu is an alternative for switching between edit and browse modes. The shortcut is `#E` no matter which mode you are in.

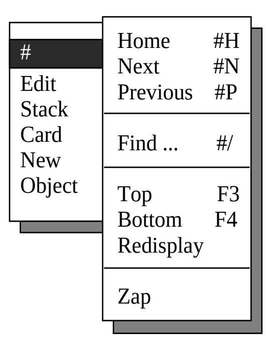

Figure 4: The # menu.

4.2.4 System Buttons

The buttons in the lower-left corner of the screen, shown in Figure 3, bring up commonly used stacks that you will probably want to access often. Here's what the buttons do:

Terminal

Pressing this button will bring up a HyperNeWS terminal emulator. Each time you press the button you will get a new terminal emulator stack with a unique name.

System Info

This function brings up the SystemInformation stack, which allows you to browse and edit HyperNeWS system parameters.

Applications

This stack provides easy access to a number of different applications running within HyperNeWS stacks.

HyperDraw & Pallet

This button makes for a quick way to bring up the drawing tool and the ColorPallet stack.

Open Stack

Use the OpenStack stack to load a HyperNeWS stack from a file.

Create New Stack

Creates a new untitled stack with a unique name, and with default *Previous*, *Next*, and *Iconify* buttons (see Figure 5). This stack is a copy of the Untitled stack, which is used as a template for newly created stacks.

Exit HyperNeWS

Exits from HyperNeWS back to the OpenWindows environment, closing all open stacks. Confirmation is first required, and a warning is given if there are any currently unsaved stacks. Either use the *Stack Info* button on the SystemInfo stack to find out which stacks are currently unsaved (see Section 4.2.4), or use the OpenStack stack. To exit HyperNeWS *and* OpenWindows, select the *Exit* menu item from the background (OpenWindows) menu.

4.2.5 Browsing Stacks

Just to get a feel for navigating through a stack, click on the *Open Stack* system button. Once it is displayed, place the cursor over the stack (you need to do this so HyperNeWS knows which stack should receive your keyboard input) and type in `Introduction`, followed by a carriage return. This is how you open up stacks. Though you'll soon learn other ways to start HyperDraw and other applications, you could also start them using the *Open Stack* button.

Once the Introduction stack has been opened, you have a few choices based upon the buttons in the lower-left corner of the Introduction stack. You can either look at this stack, look at the online manuals, or exit. For now, let's look at this stack. The arrow keys in the lower-right corner are for working your way through the cards of this stack. But all they are actually doing is sending a message equivalent to the *Next* and *Previous* selections on the # menu. So you might experiment with these menu items as well as the *Home* selection which will take you to the first card in the stack. Also try the much-more-convenient shortcuts (`#N`, `#P`, and `#H` respectively).

For another example, bring up the Hyperman stack, which is the online equivalent of this document. The Hyperman stack is identical to the hardcopy version of this manual but with an added feature: if you double-click the PointButton on any text in red, it will take you to that section or show you that figure. Figures aren't normally shown because they take time to display, and scrolling becomes very slow. But you can have any figure displayed by double-clicking on a reference or the figure tag above the caption. Perusing the online manual in this manner is a good way to get used to moving around in stacks.

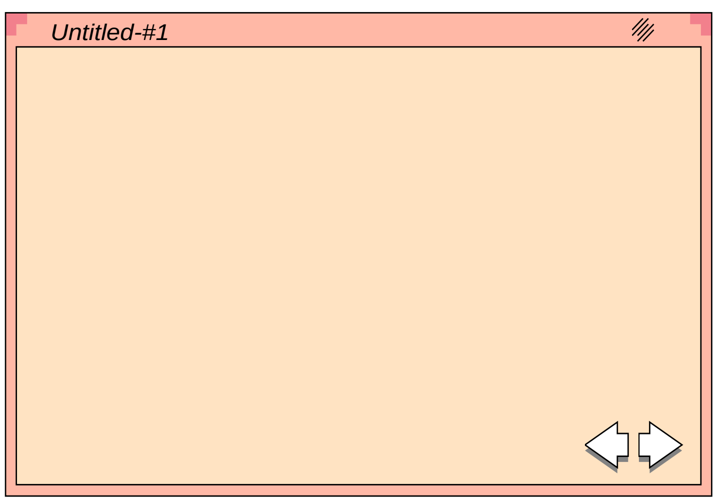

Figure 5: The Untitled stack.

4.3 Editing

4.3.1 Basics

As we said earlier, HyperNeWS has two distinct modes of operation: browsing and editing. Now that you're ready to explore the editing world, there are a few things you should know.

First and probably most important is how do you get into edit mode? You can either do it via the # menu, or you can use the shortcut of holding down the Control key and pressing the PointButton on the mouse. This will toggle back and forth between browse and edit modes. Be aware that each stack can be in a different mode. If you toggle to edit mode while the cursor is over one stack, the others will remain in browse mode unless you explicitly toggle to edit mode for each of them. You can tell when you are in edit mode because the cursor will change from the hand with the index finger pointing up, to a crosshair.

Mouse button operations take on some new meaning when you are in edit mode. When a stack is in edit mode, the PointButton is still used to select objects; you will see a *selection rectangle* around an object you have selected. The AdjustButton is used to move and resize objects. Once an object has been selected, the lower-right corner will be the *resize corner*. By pressing the AdjustButton with the cursor near the resize corner, you can change the size of the object. If the cursor is in any other part of the resize rectangle, then you'll be able to move the object. Finally, the MenuButton has added functionality when you are in edit mode. Creating new objects, modifying the stack, and adding a new card are all done from the menu while in edit mode.

4.3.2 The Expanded Menus

You'll notice that once you are in edit mode, the top-level menu (that we promised not to reference much) expands to offer much greater functionality. The *Stack*, *Card*, and *New* menus will always be displayed, and the *Object* menu will appear when a card object has been selected. We will explain what you can do with these menus as we go along.

4.3.3 How to Edit a Stack

Let's go ahead and try modifying a stack. Click the system button called *Create New Stack*. This will bring up a basic untitled stack (see Figure 5) that can be used as a framework for your own stack. Now go into edit mode on this stack by holding down the Control key and clicking the PointButton. Next, select one of the arrow keys in the lower-right corner of the stack. You can now move it around or resize it by dragging with the AdjustButton. Notice that when the selection rectangle gets bigger, this particular button doesn't change size with it. However, making the selection rectangle smaller than the button will obscure part of the button since the button drawing stays the same size. To find out how buttons react when they are resized, see Section 6.2.1.

With one of the arrow buttons selected, bring up the *Object* menu and select *Info* (`#I`). You'll see an *info stack*, one of which exists for every object type in HyperNeWS. Here you can see and modify some of the characteristics of the object. If you press the *Script* button from the ButtonInfo stack, you'll see some PostScript code that the button will execute when it is pressed. For the arrow buttons, the code instructs the stack to go either to the next or the previous card. The Untitled stack has only one card, so clicking either button will have no visible effect. Note that you have to be in browse mode to click the button; clicking it in edit mode will merely select it. To see what the info stacks for objects other than card objects look like, *Stack Info* (`#T`) can be found on the *Stack* menu, and both *Card Info* (`#R`) and *Background Info* (`#U`) can be found on the *Card* menu.

Once you've made changes to this stack, you might want to save it. You can do this in edit mode by selecting *Save* (`#W`) or *Save As* from the *Stack* menu. *Save* will only ask you for a name if it is a new stack, and *Save As* is handy if you want to save an existing stack under a different name. When HyperNeWS tries to save your stack, it will look for a directory called `~/Stacks`. If it doesn't find such a directory, it will save the stack in your home directory.

When you save a stack, HyperNeWS will almost always overwrite the stack if it already exists. An exception to this rule is HyperNeWS will not let you overwrite any system-related stacks. If at some point you decide to customize a system-related stack, HyperNeWS will save it in your stack directory, so you don't have to worry about crashing HyperNeWS because you've modified a system-related stack and (heaven forbid) you made a mistake.

If you make changes to a stack and then try to zap it without saving it, HyperNeWS will ask you if you want to throw away these changes. If you click the *No* button, the stack will remain. If you click the *Yes* button, the stack will go away. However, if you open the stack again, you will see that your changes were *not* thrown away. In effect, zapping a stack only removes it from the screen and resets the flag stating that the stack has been updated. It still exists in memory in the state that you left it.

Finally, if you make mistakes while editing a stack, using the *Revert* option from the *Edit* menu (only while in edit mode) allows the stack to be reverted to the state of the stack when it was last saved. This means any changes made to the stack since the last save will be lost.

4.3.4 More Stack Editing

Now we'll create another stack, but make this one a bit more interesting. Create a new stack like you did in Section 4.3.3. Now click on the *HyperDraw & Pallet* system button so we can create some artwork. Section 5.11 discusses HyperDraw in detail, but we'll stick to something simple here. On the ColorPallet stack, click one of the small boxes that has a color you like. This will set the current color. Now click on the little box with a square in it, above the circle on the left side of the HyperDraw stack. Using the PointButton, drag the mouse on the drawing area (the *canvas*) until you've made a rectangle. Select *Copy* from the *Edit* menu or use the `#C` shortcut (while the cursor is over the canvas so that it does a HyperDraw copy instead of a stack copy). This shape will soon become the shape of your stack.

Click on the system button *Create New Stack* and enter edit mode. From the *Stack* menu, select *Paste Shape*. What this command will do is set the stack shape based upon the contents of the clipboard. Similarly, *Copy Shape* will take the shape of the stack and place it in the clipboard. Once the new shape has been pasted, you'll notice that any objects that were on the old stack are on this one, though they may need to be moved around depending on the shape of the new stack.

Now that we have a uniquely shaped stack, let's make it slightly more interesting. From the *New* menu, create a new EditText object. This is one of the object types available to you. Once you have moved it and resized it to your liking, modify its contents on the EditTextInfo stack (by selecting *Info* on the *Object* menu). For instance, change a couple of the check boxes such as *Boxed* or *One line* (which are explained in Section 6.2.4). To close the stack, press the *Done* button.

Next, add a card by selecting *New Card* from the *Card* menu. This new, clean card will become the current visible card, though in terms of order it was placed right behind the previously visible card. You'll see that the EditText object you just created is no longer visible because it is on a different card, but the *Previous* and *Next* arrow buttons still show because they are on the stack itself, so they will be visible for every card and background.

Since we have a blank card, let's put a couple objects on it and show how one can communicate with the other. Using the *New* menu, create a new Slider object and a new PullDown object on the current card. Bring up the SliderInfo stack and call the object *TheSlider*, and click on the *Scrollbar* checkbox so there is no check in it. Make its minimum value 0 and its maximum value 100, following each entry with a carriage return. Click on the *Done* button to close the stack. Now bring up the PullDownInfo stack, and set its value to 4 numbers on separate lines, such as: 0, 40, 75, 110, and click the *SetMenu* button. (The *SetMenu* button or the similar *SetValue* button is often used in Info stacks with variables that are more than one line. On single-line variables, a carriage return tells the program, "That's the end of my input", but for multi-line variables, the carriage return is used to get to the next line.) Click the *Script* button so we can make the menu do something when it is clicked on. We want the PullDown object to send the user's selection to the slider so the slider can set itself. The following short PostScript routine will do the trick:

```
/Action { % string --
      cvi [exch] /SetValue /TheSlider Send
} def
```

Don't worry if this looks Greek to you. Basically, what this routine does is converts the string representation of the number you selected into integer, wraps it in an array, and sends it as an argument of the *SetValue* message that is sent to the slider object called *TheSlider*. But before you can test it out, click on the *Set* button to save the script of the PullDown object, and then the *Done* button to close the PullDownInfo stack. Once you have exited edit mode (and thus re-entered browse mode) you can now try selecting different menu items, and watch as the slider is set each time you make a menu selection. Notice that the maximum allowed value for the slider is 100, but one of the menu options is 110. If you try to set a slider's value to a number outside its range, it will set itself to the closest number within its legal range.

That concludes our examples that show how new stacks are made. At this point you may be ready to go off and try creating some stacks of your own. Or, you may want to read some more. Section 7 will cover other editing features in greater detail.

4.4 Startup Options

When you first start HyperNeWS, there are a couple of things that you can do to alter the environment. The first is the `HNPATH` environment variable, which lets you specify directories to search for stacks . If you specify one or more directories separated by colons before starting HyperNeWS, such as:

```
setenv HNPATH /users/joe/Stacks:/users/fred/Stacks
```

these directories will appear in the search path card of the SystemInformation stack, along with the directories that you specified within HyperNeWS.

Once HyperNeWS has loaded all of its PostScript files, it will execute a file in your `~/Stacks` directory called `HyperInit.ps`, if it exists. This is a convenient place to load other PostScript files. Possible uses might be, for example, to redefine the routine that determines where icons are placed, or to add a new class to the HyperNeWS class hierarchy. You might also experiment with loading and positioning stacks that you use on a regular basis.

Finally, if you are a more advanced user and want to experiment with some of the HyperNeWS PostScript files, you can do so by placing your modified copy in `~/HyperNeWS`. HyperNeWS will check this directory first before loading any PostScript files from the usual place. For example, if you wish to experiment with button objects, copy `button.ps` from the directory `$HNHOME/etc/HyperNeWS` into `~/HyperNeWS` and make your changes. The next time you start HyperNeWS, it will load your copy of the PostScript file.

5 Stack Descriptions

5.1 Basics

All the stacks within HyperNeWS are available for your perusal and modification. The stacks that come with the base system are organized into three subdirectories of `$HNHOME/Stacks`:

Info

This directory contains all the info stacks for HyperNeWS object types.

System

Stacks that are a part of the base system reside here. This includes things like the HyperDraw, Clipboard, and Applications stacks.

Misc

These are miscellaneous stacks that aren't a part of the base system. This includes the Hyperman and Perfmon stacks.

If you make any changes to one of these stacks and save it, the stack will be saved in your `~/Stacks` directory, if it exists. Otherwise it will be placed in your home directory. Therefore, you can experiment with stacks and save your own personalized copies of them.

The rest of this section will describe most of the stacks that you will be working with.

5.2 Applications

This stack demonstrates a number of different HyperNeWS stacks that you may find useful. Pressing any of the buttons causes the application to start. You can iconify this stack or move it around, since you may want to access it often. All of the stacks that are available from the Applications stack are described in this section.

5.3 BrowsingHelp

The BrowsingHelp stack explains some of the basics of browsing and mouse usage in HyperNeWS. Most of the drawings you see in this stack are actually Button objects...changing or moving drawings on a stack is sometimes easier with buttons than it is if you modify the actual stack drawing.

5.4 ButtonIdeas

The ButtonIdeas stack has samples of some customized buttons. It has buttons that you can use for resize corners or close buttons on your stacks, and it also shows how to make a button consisting of multiple HyperDraw drawings that will be cycled through when you click on them.

5.5 ClassBrowser

The ClassBrowser stack gives you some online information regarding HyperNeWS object types. The main card shows you the relationship between the objects, identical to that in Figure 1. By clicking on any of the classes, you will see a list of methods and instance variables for that object type. Clicking on one of those lists will show you the specifics of that method or instance variable. This information is stored in HyperNeWS, and is generated from the actual PostScript code. Notice that the shaded text on the home card represents the only objects that you will ever have an instance of. The others are abstract superclasses that define behavior and instance variables for all subclasses.

5.6 Clipboard

Whenever you do cut & paste operations, you are cutting to, and pasting from, the Clipboard stack. This stack has 16 cards, giving you the ability to store 16 different objects at any given time. Use the normal card movement commands (such as *Next* and *Previous*) to move between cards on the Clipboard stack. Cutting and pasting will always use the current card. For information on the programmer's interface to the clipboard, see the Reference Manual.

5.7 Clocks

This stack shows some examples of digital and analog clocks. They are actually buttons, and you can get an idea how buttons can be programmed by looking at the scripts in this stack.

5.8 ColorPallet

When you start HyperDraw, you automatically see the ColorPallet stack as well. The ColorPallet can also be started from the Applications stack if you won't be using HyperDraw but you may be changing colors in an info stack. As shown in Figure 6, the home card of the ColorPallet has two large squares on the left, and quite a few smaller squares. The top left square is the fill color, and the square below it is the stroke color. Choose a fill color by clicking the PointButton in a smaller square. Choose a stroke color by doing the same while holding down the Shift key.

To modify a color, double-click on one of the smaller squares. You will see the secondary ColorPallet card shown in Figure 7. Adjust the red, green, blue, and gray sliders until the large square is the color you want. Then click on the large square to take you back to the main card. Finally, click on the small square you just changed to set the current color.

5.9 Console

This is a HyperNeWS terminal emulator, acting as the Unix console. You can only have one Console stack open. If you try to open it again, it will just move to the front of the screen.

5.10 dial

The dial stack is an example of a new object that was created by using an existing object as its parent. Most of its functionality is the same as a Slider object, but the appearance is drastically different. Creating new object types is discussed in the HyperNeWS Reference Manual.

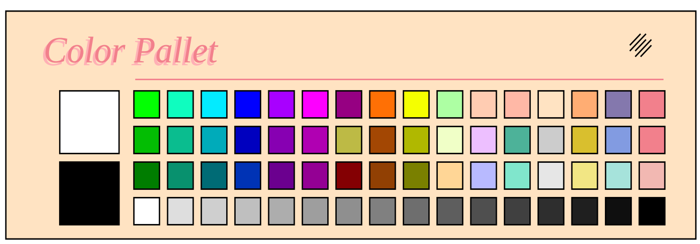

Figure 6: The ColorPallet home card.

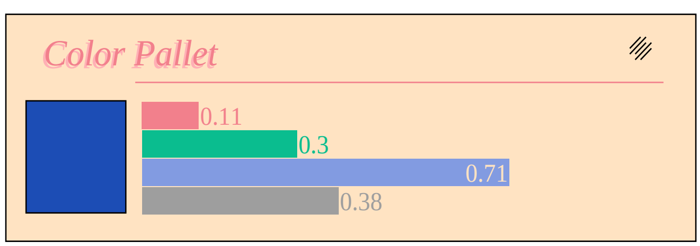

Figure 7: The ColorPallet secondary card.

5.11 HyperDraw

5.11.1 Basics

HyperDraw is split into three parts: the button panel, the canvas, and everything else. Using the mouse in HyperDraw is fairly consistent with the rest of HyperNeWS. In the canvas, the PointButton is used to select objects, the AdjustButton is used for moving and resizing objects, and the MenuButton brings up the HyperDraw menu (discussed in Section 5.11.3). The AdjustButton will move an object if it has been selected and you drag the mouse, except when the cursor is on a *handle*, which is one of the small squares that are used for resizing. The AdjustButton can also be used to rotate an object if the Shift key is held down before the AdjustButton is pressed on a handle. In the button panel, the PointButton activates everything. And since HyperDraw is just another stack, the rest of the areas respond to mouse presses the same way any other stack would.

5.11.2 The Button Panel

The panel along the left edge of HyperDraw has most of the controls that you will need to create things. The panel is shown in Figure 9, and the button descriptions are as follows:

File menu

A menu that offers file input and output capabilities for HyperDraw.

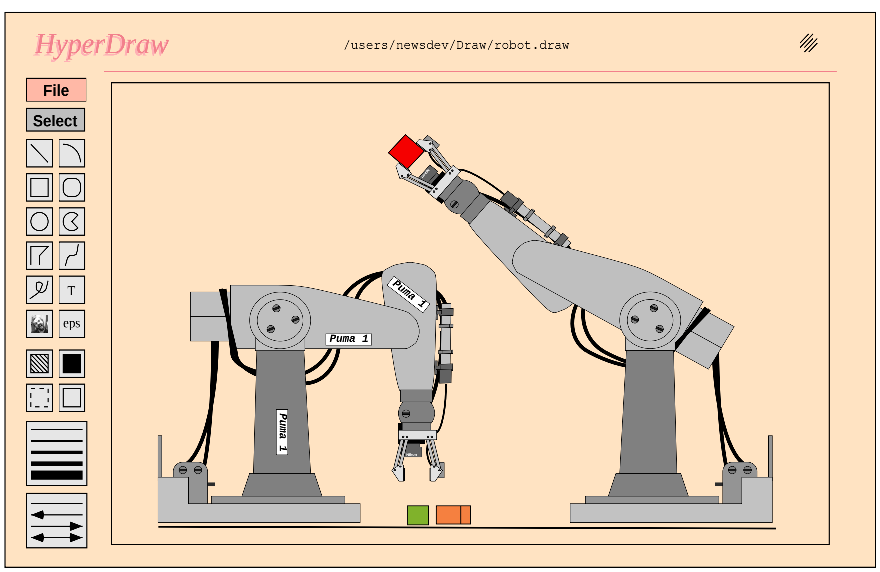

Figure 8: The HyperDraw drawing tool.

New

If a drawing exists, you will be asked if you really want to start a new drawing. A click on the *Yes* button will cause HyperDraw to delete any existing work, leaving you with a clean canvas.

Open

Since opening a drawing deletes anything currently in the canvas, HyperDraw will ask for confirmation if you've made any changes to the current drawing since last saving it. Then it will display the OpenFile stack, shown in Figure 10. You can navigate through the file system by clicking *Up* to go up one directory, or by double-clicking on a displayed file. If it is a directory, OpenFile will open that directory. If it is a HyperDraw file, it will open that file. In order to help you move around, the OpenFile stack saves a record of directories you've been in. By clicking the PointButton on the directory path at the top of the stack, you'll see a menu of recently-visited directories to choose from.

Save

If you want to save a new drawing (i.e. it is currently called *Untitled*), selecting this option will display the SaveFile stack, which is quite similar to the OpenFile stack. Once you have gotten to the directory you want to save the file in, click the mouse in the lower text field and type in the name you want to save the drawing as. Be sure to end the drawing name with a `.draw` suffix, and then press the Return key. If the drawing you are working on is *not* new, then HyperDraw will save the displayed copy of the drawing over the old copy without prompting you.

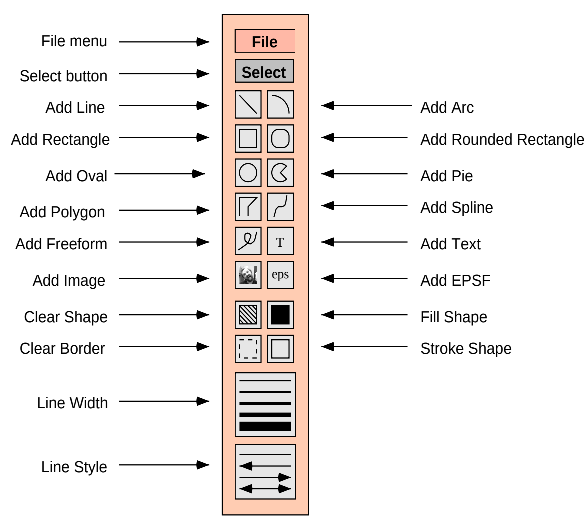

Figure 9: The HyperDraw button panel.

Save As

If you have modified an existing drawing but you don't want to clobber the original, use this option. Selecting this option is similar to selecting *Save* when working on a new drawing.

Revert

In case you've messed up your drawing beyond recovery, selecting *Revert* will read in the last saved version of the drawing. Therefore, this operation only makes sense when you have at some point saved this drawing.

Select button

When over the canvas, the cursor is in *select* mode by default. This is represented by an arrow pointing to the upper-left. If one of the next ten buttons are pressed, they can be undone by clicking on the Select button.

Add Line

Create a straight line segment.

Add Arc

Create an arc that can be modified by moving its handle.

Add Rectangle

Create a rectangle by dragging by the corner. Next to the rubber band will be the dimensions of the new rectangle.

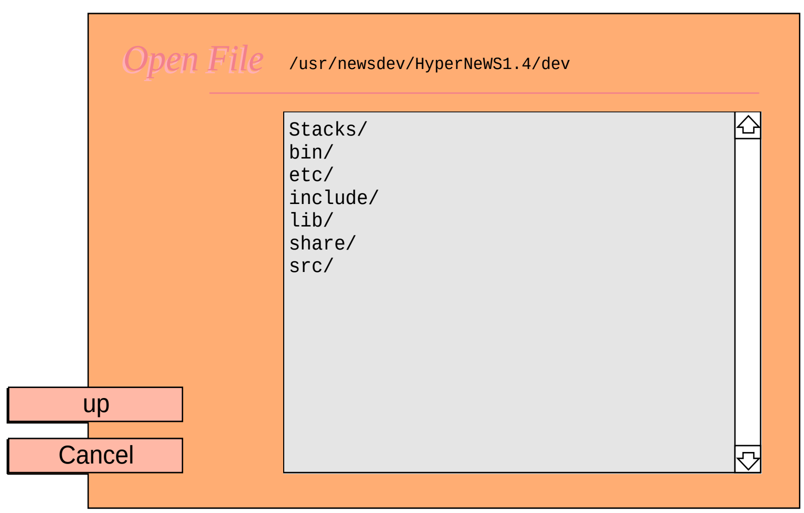

Figure 10: The OpenFile stack.

Add Rounded Rectangle

This is similar to a plain rectangle, except there is a handle inside a corner that lets you modify how rounded the corners are.

Add Oval

Create an oval. Again, the dimensions will be interactively displayed.

Add Pie

Similar to *Add Oval*, except the oval will not be closed.

Add Polygon

What first acts like adding a line will leave a handle in the middle of the first line. Drag the handle where you want it, and a new handle will appear. Each time you drag a handle, a new one will be created.

Add Spline

Similar to *Add Polygon*, except that the sharp corners of a polygon will be replaced by the rounded forms of a spline.

Add Freeform

As long as you hold down the PointButton and drag the cursor around the canvas, you will create a line with a number of handles to define it.

Add Text

Once you click on a spot on the canvas, a dialog box will appear for you to type in the text and set style attributes. Note that text can be treated just like any other object: it can be moved, resized, and even rotated.

Add Image

This button will ask you to specify an image file. It may be in HIPS format or Sun raster format. Once it has been loaded, this too can be manipulated like any other HyperNeWS object. Selecting either this option or *Add ESPF* will display the OpenFile stack for you to specify the image or ESPF drawing to read in. These two functions are similar in that they will load a drawing in with the existing drawing. They will not erase the current contents the way *Open* and *New* will.

Add EPSF

Specify an Encapsulated PostScript (EPSF) drawing to read into HyperDraw.

Clear Shape

Do not fill new objects or currently selected objects with the fill color.

Fill Shape

Fill new objects and currently selected objects with the fill color from the ColorPallet.

Clear Border

Do not stroke the border of new objects or currently selected objects.

Stroke Shape

Stroke the border of all new objects and currently selected objects with the stroke color from the ColorPallet.

Line Width

Choose one of five line widths for lines and the borders of other objects.

Line Style

For lines and arcs, choose a line style that has from zero to two arrowheads.

5.11.3 The Canvas Menu

When you click the MenuButton over the HyperDraw canvas, you'll see a HyperDraw menu with five sub-menus. The sub-menus are listed below:

Edit

This menu offers the same type of commands as the *Edit* menu for all stacks. However, common commands like *Copy* and *Paste* have some subtle differences depending upon whether the cursor is over the canvas (i.e. the canvas menu), or the stack (thus the usual *Edit* menu). When using HyperDraw, make sure you have the cursor over the HyperDraw canvas when you perform operations from the *Edit* menu.

Arrange

By using *To Front* or *To Back*, you can make selected objects appear in front of, or behind other objects. If you have a number of objects that you want to group together as a single entity, use *Group*. *Ungroup* will remove the grouping attribute from the objects.

View

The *Color* command lets you specify whether drawings are in color or shades of gray. Using *Zoom In*, *Zoom Out*, and *Zoom Normal*, you can look at your drawing in more detail, or look at the entire drawing. The move operations at the bottom of the menu can also be performed with the arrow keys on the keypad.

Grid

Adjust the *snap grid* that all objects will cling to. You can turn the grid off or on, and you can adjust how big the grid is.

Special

The *Flip Poly and Spline* option will toggle back and forth between straight lines connecting the handles, and splines that go through the handles. *Create Solid Group* will let you make a solid out of a non-solid, such as a line made with *Add Freeform*.

5.12 Hyperman

This is the online documentation stack for HyperNeWS. It consists of the User Guide and the Reference Manual, just like the hardcopy documentation. Any references to figures or other sections will be displayed in red. If you double-click on those references, Hyperman will take you to that figure or section. The Hyperman stack is not in such a format that it can be printed out. If you want hardcopy versions of the manuals, contact the Turing Institute at the address on page 2 for details.

5.13 Information

The Information stack is the first thing you'll see when you start HyperNeWS. It says a few words about HyperNeWS conditions of use, and it has buttons that allow you to start the online manuals, start the TuringInfo stack, or exit the stack. If you press the *Done* button, you will see the console and the system buttons appear, if they aren't already showing.

5.14 Movie

This stack shows a simple example of animation in HyperNeWS, with Dr. Peter Mowforth of the Turing Institute in the title role. Like the Clocks stack, it is all done with a button object. The button was made by pasting 16 digitized images into it, and a little loop continuously simulates mouse presses over the button causing it to cycle through the images.

5.15 NewObjects

The NewObjects stack contains examples of the card object types available in HyperNeWS. These are objects that you can easily cut & paste into your own applications.

5.16 OpenStack

This stack is used to select a HyperNeWS stack for opening (see Figure 11). Directories are searched in the order specified in the search path of the SystemInfo stack. Enter the name of the required stack (without the `.stack` suffix). If you have opened and closed a stack, it will still be resident in HyperNeWS. If you click the PointButton on the word *Name:* in the stack, you will see a menu consisting of all previously loaded stacks. If the stack you want to show is in the list, simply select it from the menu instead of typing it in again. Any stack name on this menu with an asterisk after it has been modified.

5.17 Peek

The Peek stack lets you "peek" at PostScript data structures. When you open the stack, you'll see the keys and values of the HyperNeWS dictionary. If you click on a dictionary, array, or packed array, it will then descend into that structure and peek at its contents. Pressing the *Up* button will return up one level, and the *Top* button will take you back to the HyperNeWS dictionary. This can be quite helpful for debugging or for exploring the innards of HyperNeWS.

5.18 Perfmon

This performance monitor stack displays some system statistics. Along the right-hand side, it lists the users logged on to the machine, and if logged in remotely, the remote host is shown in parentheses. The blue, orange, and green portions

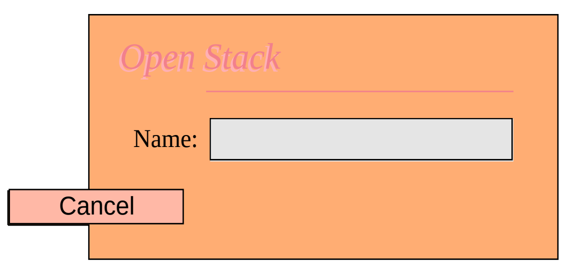

Figure 11: The OpenStack stack.

represent the system load, user load, and nice values, respectively. The black line displays the load average.

5.19 postit

This is a simple little stack not unlike a post-it pad. You can keep a few notes on each "page" and move between them. Pressing *Save* will save the stack and your additions in your stacks directory.

5.20 PostScriptError

This stack replaces the normal PostScript error handler, so you can see the PostScript error in a HyperNeWS stack when a problem has occurred in your script. You'll see the contents of the operand stack shown in an EditText object, and the stack also shows the error that occurred, the message that caused the error, and the sender and receiver of the message. Don't bother trying to press the *Trace* button, because tracing has been disabled...it tends to bring the X/NeWS server to its knees.

5.21 SaveScripts

The SaveScripts stack gives you a way to save all the scripts from a stack into an ASCII file, suitable for printing. It will prompt you for the name of the stack, and it will save a script file in the directory `~/Stacks`. For instance, if you have a stack called MyStack, it will save the scripts from the stack in `~/Stacks/MyStack.scripts`.

5.22 system

The system stack is a stack that covers the whole screen; it is the background of the screen, colored in two shades of blue and displaying the Turing and HyperNeWS logos. It is similar to other stacks, but it has some behavior all its own:

- Some of the normal stack operations that don't make sense with the system stack are ignored. They include: showing, hiding, moving, resizing, bringing to front or sending to back, and closing.
- When in browse mode, it does not have a HyperNeWS menu. You will see the OpenWindows menu instead. But when you switch into edit mode, you will see the same menu as you would when editing any other stack.
- It does not respond to menu shortcuts. This becomes evident particularly in edit mode. You must actually select the menu items for the tasks you want to perform on the system stack.

We make this distinction between system stack and background because what is typically called the background in a windowing environment is in fact a stack in HyperNeWS. Also, calling it a background may lead to confusion later when we talk about the relationship between cards and their backgrounds.

5.23 SystemInfo

The SystemInfo stack allows you to browse and edit HyperNeWS system parameters that take on default values when you first start HyperNeWS. Once the stack is open, the following options can be selected, after which *Index* returns you to the home card, and *Done* will exit the stack:

Client Info

This card displays the status of all clients which are connected to stacks.

Color

This option allws the background color to be changed using the ColorPallet stack. A facility is also provided to view the HyperNeWS environment in black & white. This is useful when developing applications on a color workstation which are to be used on a grayscale or monochrome screen.

Logging

Message logging can be enabled or disabled from this card. Initially logging is turned off. Logging output may be directed to a file or terminal by entering the name of the device in the box provided. For example, entering `/tmp/mylog` causes logging messages to be directed to that file. If the field is empty, logging goes to the console. The *Log Updates* checkbox will show *all* messages that are sent in HyperNeWS, whereas without this checkbox set, you will see only the more interesting messages.

Search Path

This card lists the directories which HyperNeWS searches (starting from the top) when opening stacks. Once you have created your own `~/Stacks` directory, it will be included in front of the system directories (see Figure 12). Additional paths may be added by editing the EditText object holding the existing paths, then clicking the *Change* button to update it. HyperNeWS will always place the system directories last; you can't put a directory after it.

Stack Info

This displays the status of all currently loaded stacks. A stack remains loaded in memory for the duration of the HyperNeWS session, even after it has been closed or zapped. If it is re-opened during the session, the memory copy will be loaded rather than the copy saved on disk.

5.24 Terminal

When you press on the *Terminal* system button, you get a terminal emulator with a unique name consisting of "Terminal-#" followed by a number.

If you want to modify how all your terminals will look, edit the Terminal stack, which is the template from which all unique terminals will be created. You may want to change the fill color of the Canvas object, for example. When you save the stack, it will be saved as `~/Stacks/Terminal.stack` (assuming you have created the directory), and every subsequent terminal will use that stack as a template.

5.25 TuringInfo

This stack gives some background on the Turing Institute and some of the services it provides.

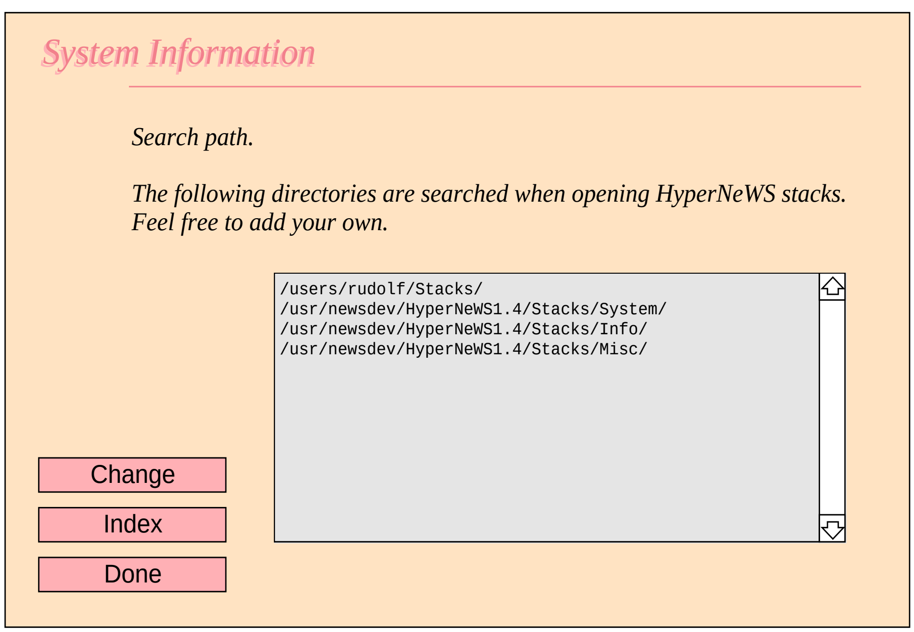

Figure 12: The card for viewing or changing your search path.

6 System Objects

6.1 Basics

All HyperNeWS objects share certain common features. The way they are selected, moved and the way they exist in the hierarchy of cards, backgrounds and stacks is identical regardless of object type. Objects are addressed using a two part name:

```
Stackname:Objectname
```

This means that each object must have a unique name within a stack. Two different objects may not have the same name even if they exist on different cards or backgrounds, or if they are different object types.

All objects may contain PostScript methods which define their behavior. An object's methods are defined collectively as a *script*. For details on writing scripts see Section 9. To edit an object's script, select the *Script* option from the *Object* menu (see Figure 26), or alternatively select the *Script* button from the object's info stack.

Each object type has a different set of attributes which are relevant to that object type. However, they all require a font name, font size, and a unique name identifier. These fields are common across all object types and can be edited from the fields shown below:

Name

> Define a new name for the object. Click the PointButton in the name field then type the name followed by the Return key. Names must be unique and may not contain spaces or any of the following characters: ".\t\n()#". If the name entered is not unique, a system name will be supplied. (Stack names cannot be changed this way. See Section 7.5.3 for details.)

Font

> A new font for the object is specified by selecting the font name and type size from two menus. One menu is labeled with the font style, the other with the font size. Together they define a font for the text. Click the PointButton over the box label, drag down and release over the desired selections.

Each object type has its own info stack which is used to change an object's attributes. An info stack is named by concatenating the object type name with *Info*. For example, the info stack for Button objects is named ButtonInfo. Once a system object has been selected, its info stack can be displayed by choosing *Info* from the *Object* menu. Each system object, its behavior, and info stack are described below.

6.2 System Object Types

6.2.1 Button

Button objects are used to create a button type input selection mechanism. Clicking on them with the PointButton activates them and causes their value to be returned. Buttons can take one of four different types:

Push

Pushbuttons have a single state (always 0), and hence can only return a unary message indicating selection.

Check

Checkboxes have a binary state (1 or 0), indicated by the presence or absence of an "X" in their box.

Transparent

Transparent buttons are similar to pushbuttons in that they have a single state (always 0). However they have no visible form on the stack. They can be used to detect user input from a given stack area without the need for a visible form. Note that a transparent button is not the same as an invisible (hidden) button object from which user input is disabled.

Drawing

Drawing buttons are created using HyperDraw and pasted onto the stack (see Section 7.6.1).

An example of each of the button types is shown in Figure 13. The left and right arrows are drawing type buttons. They exist as the template objects *NextButton* and *PreviousButton* (see Section 7.2.1). Their behavior has been set up to move to the next or previous card of the stack when they are selected. A transparent button is also shown, although you can't see it!

Once a default button object has been created from the *New* menu, it can be modified by entering its details on the ButtonInfo stack (see Figure 14).

The following specific attributes of the object may be set from this stack:

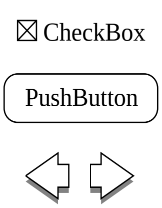

Figure 13: Examples of button types.

Label

Set the label for the button, i.e. the text that appears on the screen. This field only applies to Check and Pushbutton types.

Type

A menu to change the type of the button. The default is Push.

Rounding

Specifies how rounded the corners of the button are to be. Setting to zero gives a rectangular box, while setting to the maximum rounding yields a circular button. This applies only to Pushbuttons.

Checkbox

Used to set the size of the checkbox.

Stroke & Fill

Defines the text and background colors for the button label using a ColorSelect object. This menu allows a choice of colors including the current color in the ColorPallet.

When you are editing buttons, you'll notice that they don't all resize the same way. Only the Pushbutton's size is controlled by resizing the selection rectangle. Drawing buttons and Checkbox buttons will remain the same size regardless of the selection rectangle's size, though parts may be obscured if the selection rectangle becomes smaller than the button.

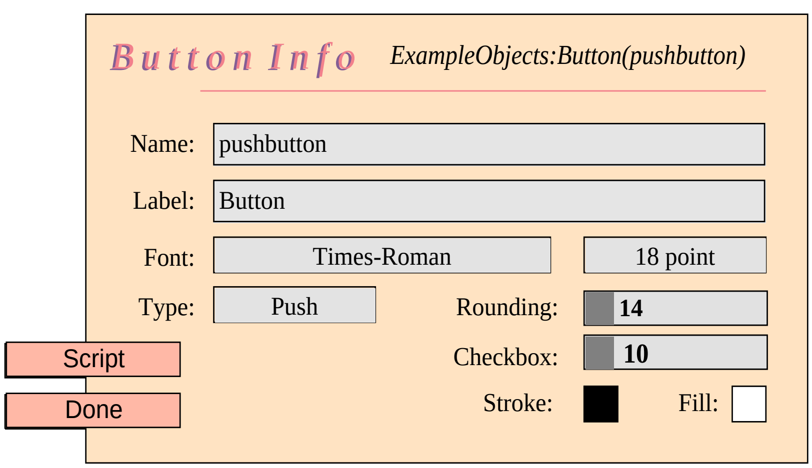

Figure 14: The ButtonInfo stack.

6.2.2 Canvas

Canvases are used to define an area of the stack which can be used by application programs for their own use. An example use of a canvas is the HyperNeWS GraphEditor, shown in Figure 15 (but not a part of the base HyperNeWS system).

Canvases send messages when the cursor enters or leaves the canvas area. Once a default canvas object has been created using the *New* menu, it can be modified by entering its details on the CanvasInfo stack, shown in Figure 16.

The following specific attributes of the object may be set from this stack:

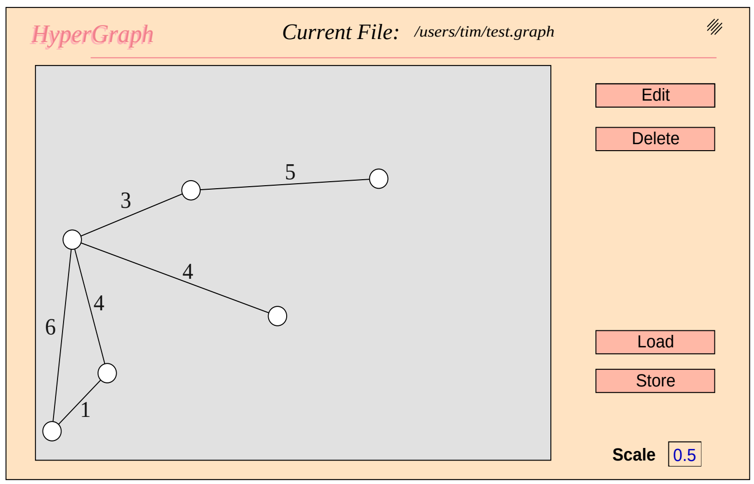

Figure 15: The GraphEditor tool using a canvas object.

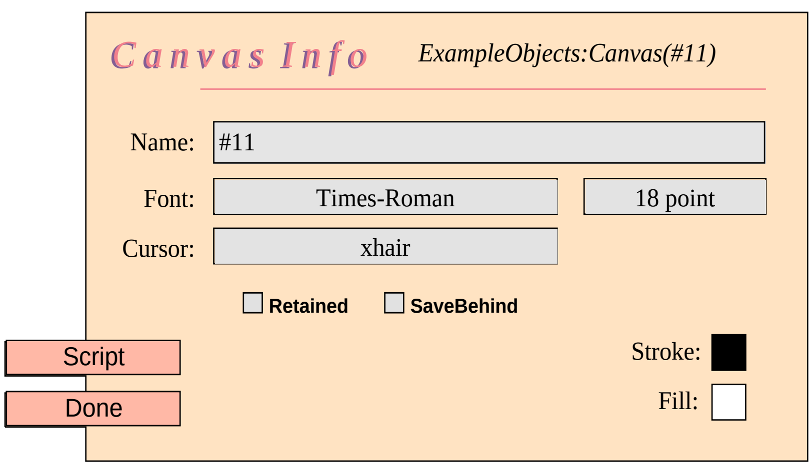

Figure 16: The CanvasInfo stack.

Cursor

A menu used to define the shape of the cursor while it is over the canvas. The default is *ptr*.

Retained

When selected, the contents of the canvas are retained so that updates occur faster when the canvas is obscured and then uncovered.

SaveBehind

When selected, the screen area behind the object is saved in memory for fast updating. This is useful if the canvas will only be displayed for a short time before the screen area behind it is exposed again.

Stroke & Fill

Defines the text and background colors for the canvas using an object of type ColorSelect. This menu allows a choice of colors including the current ColorPallet color.

6.2.3 ColorSelect

ColorSelect objects are used to specify a color. Its current color setting is displayed in a box along with a text label. Pressing the PointButton over the color box will display a menu. The default menu allows a choice of grayscale colors from white through black, or selection of the current color from the ColorPallet stack. Examples of ColorSelect objects are shown in Figure 17. Once a default ColorSelect object has been created from the *New* menu, it can be modified by entering its details on the ColorSelectInfo stack (see Figure 18).

The following attributes of the object may be set from this stack:

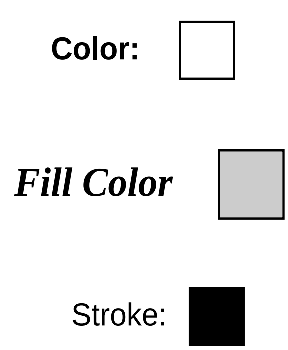

Figure 17: Examples of ColorSelect objects.

Label

Defines the visible text displayed as a label for the color box.

Stroke

Uses a ColorSelect object to define the text color for the label.

6.2.4 EditText

EditText objects, such as those in Figure 19, allow text to be presented in a number of different formats: it may be boxed, editable, scrollable, etc. On a stack, only one EditText object is active (in focus) at any one time. ASCII input, with the cursor anywhere on the stack, is directed to the EditText object currently in focus. To change the focus from one EditText object to another, simply click the left mouse button in the desired object. A thin vertical line in reverse video shows which EditText object is in focus.

If the text is scrollable, a scrollbar allows scrolling one line at a time, by clicking the left mouse button on the up-arrow or down-arrow. More than one line can be scrolled by either dragging the scroll position box using the PointButton, or

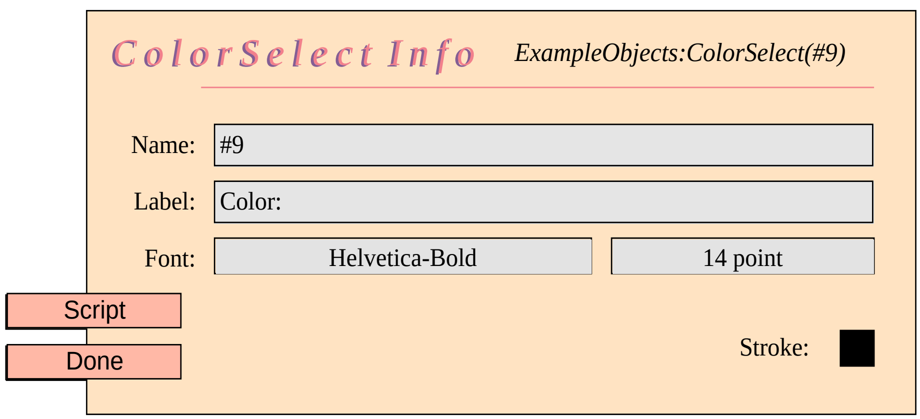

Figure 18: The ColorSelectInfo stack.

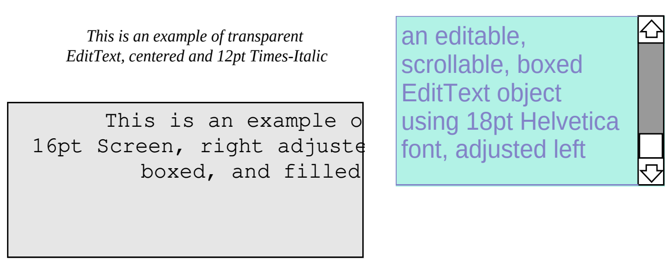

Figure 19: Examples of EditText objects.

clicking the mouse in the scrollbar area. If the EditText object is not scrollable, some of the text may not appear if the box is not sufficiently large to display all the text. If this occurs, then resize the object.

Once a default EditText object has been created from the *New* menu, it can be modified by changing its details on the EditTextInfo stack (see Figure 20).

The following specific attributes of the object may be set from this stack:

Justification

A menu to select text justification. Initially set to *Left*.

Value

The contents of the text field. Edit this field to show the required text, then press the *Set Value* button to change its value. Formatting of the text in the object is based upon its attributes and the size of the object, not by the way it is displayed in the EditTextInfo stack.

Stroke, Fill, ScrollFill, and ScrollCol

Sets text and background colors, as well as scrollbar colors for theEditText object using ColorSelect objects.

Editable

Selects the text to be user-editable, otherwise it is read-only.

Scrollbar

When checked, makes the text scrollable by adding a scrollbar on the right-hand side of the object.

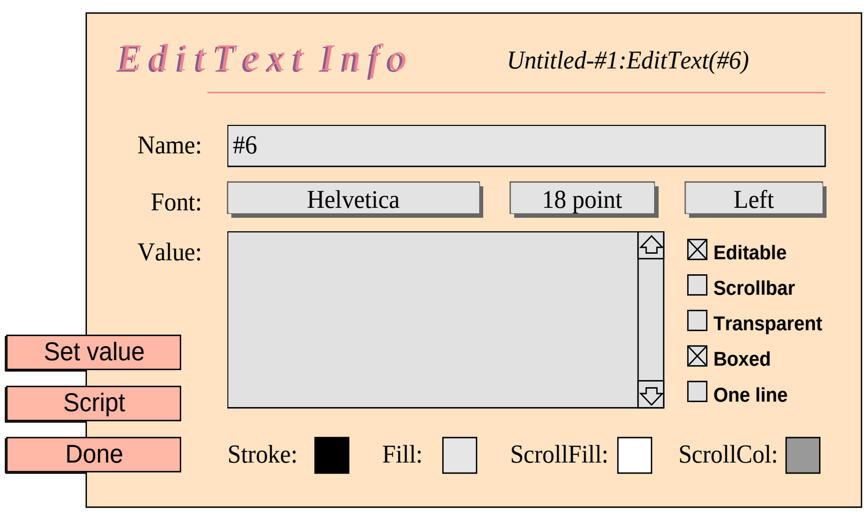

Figure 20: The EditTextInfo stack.

Transparent

Results in the text appearing with a transparent background. Setting this attribute resets the *Editable*, *Boxed* and *Scrollbar* checkboxes.

Boxed

Causes an enclosing box to be drawn around the text object.

One line

If set, only a single line of the text box is returned as the message argument when selected by the user. Otherwise the whole text field is returned.

Details on exactly what is returned by an EditText object under all circumstances can be found in Table 2 in Section 8.3.2.

6.2.5 PullDown

PullDown objects are used to define your own menus. They consist of a boxed label which will display the menu selections enclosed in a shadowed box. Some examples of PullDown objects are shown in Figure 21. Selection of a menu item is done by dragging the mouse down over the required selection using the PointButton. Once a default PullDown object has been created from the *New* menu, it can be modified by entering its details on the PullDownInfo stack (see Figure 22).

The following attributes of the object may be set from this stack:

Label

Defines the visible text displayed as a label for the menu. The text can either be fixed, or it can be set to the new selection each time a selection is made.

Menu

This EditText object allows the menu items to be defined. Use a separate line for each selection. Horizontal menu lines can be generated by entering a '-' on a new line. To update the menu, click the *Set Menu* button. Any empty lines you leave at the end of the EditText object will be removed.

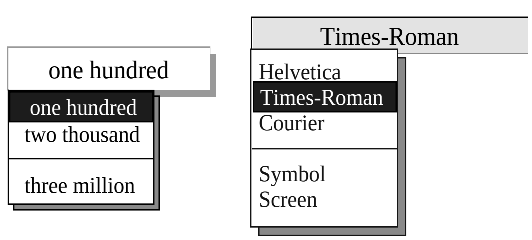

Figure 21: Examples of PullDown objects.

Fill, Stroke & Shadow

Used to set the background, text, and shadow menu colors using ColorSelect objects.

Shadow size

Used to select a shadow size for the label box. Setting to zero causes no shadow, and using 10 causes the largest shadow.

Set label

If set, the menu label is updated to reflect the most recent selection.

6.2.6 Slider

Slider objects are used to select an integer value from a sliding scale. The number must lie between a minimum and maximum value. They can appear in two different basic forms:

- A long rectangle showing the value.
- In a scrollbar form, with a movable ball which can be dragged along the length of the object.

Examples of Slider objects are shown in Figure 23. Once a default slider has been created from the *New* menu, it can be modified by entering its details on the SliderInfo stack (see Figure 24).

The following specific attributes of the object may be set from this stack:

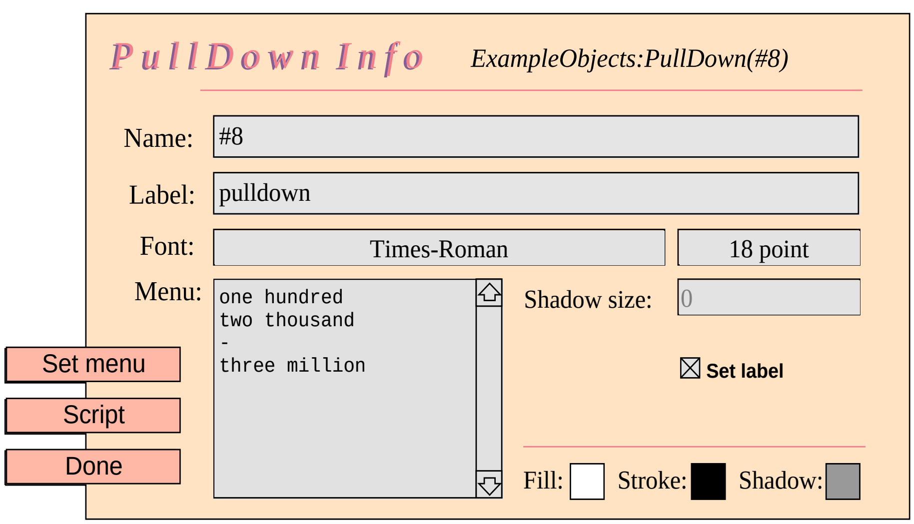

Figure 22: The PullDownInfo stack.

Stroke, Fill, Value & Inactive

Defines the text, background, active, and inactive colors for the slider using ColorSelect objects. The menus allow a choice of colors including the current color from the ColorPallet.

Min

The minimum allowable integer value.

Value

The current slider value. This must be an integer and lie between the minimum and maximum.

Max

The maximum allowable integer value.

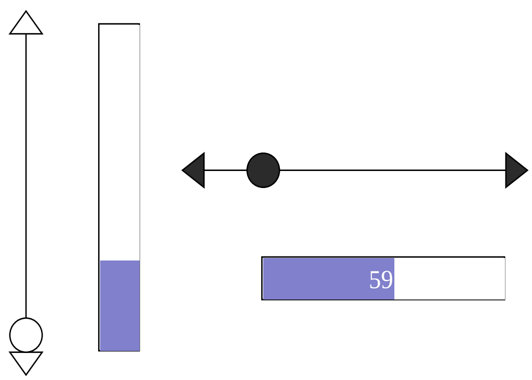

Figure 23: Examples of Slider objects.

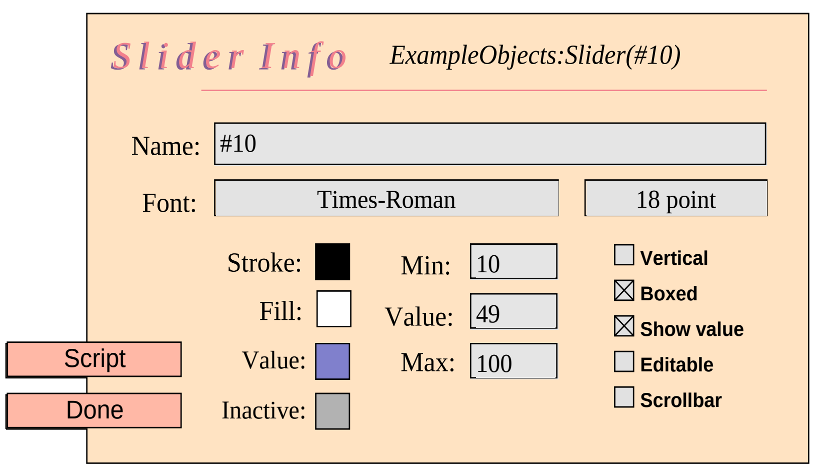

Figure 24: The SliderInfo stack.

Vertical

If selected, the slider is oriented vertically rather than horizontally.

Boxed

If set, an enclosing box is drawn around the slider.

Show value

Displays the current value of the slider only if the *Scrollbar* checkbox is not set.

Editable

Defines whether the slider can be changed by user input on the screen. You can change the value of the slider either by clicking on the slider, or by dragging the PointButton so you can see the values while you are dragging.

Scrollbar

When set, the slider takes on the representation of a window scrollbar.

7 Advanced Editing

7.1 Creating New Objects

As you probably discovered while going through the examples starting in Section 4.3.3, the *New*menu (see Figure 25) is broken up into two parts: one which lists available template objects (described in Section 7.2) and one which shows card object types. When you create a new card object, it will appear at some default location on your stack. Six card object types exist: *Button*, *Canvas*, *ColorSelect*, *EditText*, *PullDown*, and *Slider*. Each of these object types has its own particular set of attributes which are described in Section 6.

Each new object that is created is automatically given a name by the system. Object names must be unique within a stack, and are named using a *hash* (#) followed by a number. To rename a card object or change any of its other attributes, select the object and select *Info* from the *Object* menu (see Figure 26).

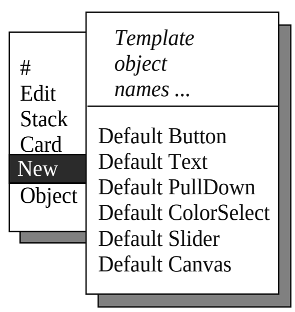

Figure 25: The *New* menu.

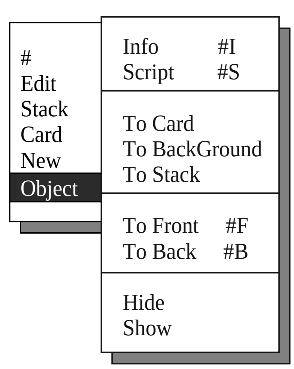

Figure 26: The *Object* menu.

7.2 Using Templates

7.2.1 Card Objects

HyperNeWS uses the NewObjects stack, shown in Figure 27, as a template for new objects. The stack contains all the objects which appear in the *New* menu above the separator line. You can add your own object templates to this stack and assign a relevant name to them. Since the menu is constructed at initialization time, objects you have added will appear on the *New* menu the next time HyperNeWS is started.

7.2.2 Stacks

The Untitled stack serves as a template for newly created stacks. A copy of this stack (with a unique name) is created when you press the *Create New Stack* system button. You can edit and tailor this stack into your own format, and then save it so that each newly created stack will be in your own preferred style.

7.3 Moving Objects

7.3.1 Card Objects

When a card object is first created, it is placed by default on the current card. Once selected, card objects may be assigned to a card, background, or stack. To

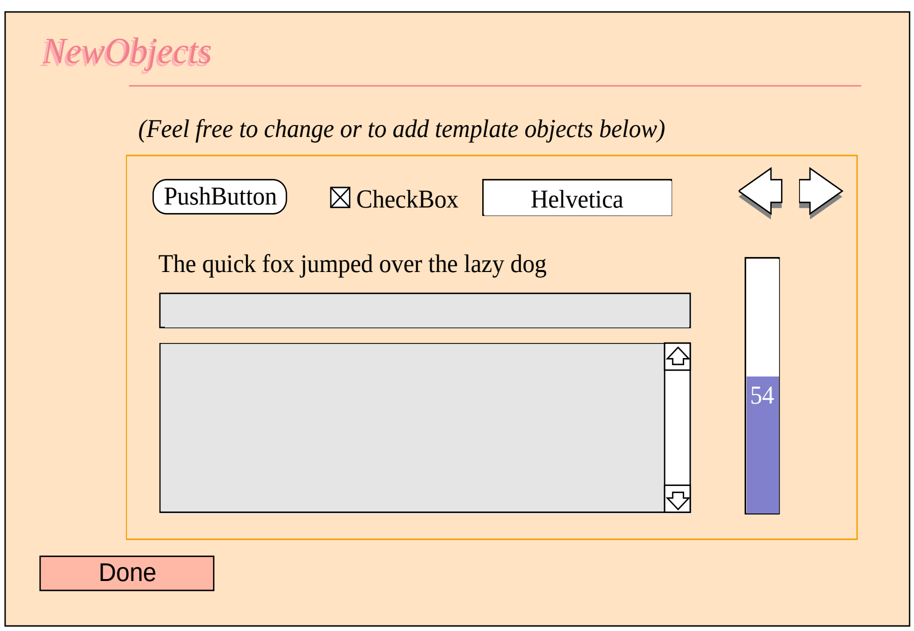

Figure 27: The NewObjects stack.

move an object to another card, you would simply select the object, cut it from the current card, make the desired card the current card, and then paste the object onto that card. Other types of moves can be done using one of the following options from the *Object* menu:

To Background

The object is assigned to this card's background, and appears on all cards that share the background. If the stack has only one background, the object will be present on all cards.

To Card

The object is specific to this card, and will not appear on any other. This is useful if you have previously moved an object to the stack or a background and now you want to move it back to a card.

To Stack

The object is assigned to the stack and will appear on all cards regardless of background.

7.3.2 Cards

When a new card is created using the *New Card* option from the *Card* menu (see Figure 28), it is added to the stack after the currently visible card, and then made the current card. This means that no new card can ever be added before the home card of the stack.

In order to move a card within a stack, that is change the sequence of the cards, you'll need to use *Copy Card* from the *Card* menu and *Paste* from the *Edit* menu. (While you may question why you need to use two different menus, there is a valid reason. When you copy, HyperNeWS needs to know if you want to copy a card, card object, or stack to the clipboard. So you specify *Copy Card*. But when you paste something, HyperNeWS knows you are pasting from the clipboard, and it can tell what kind of object it is pasting.)

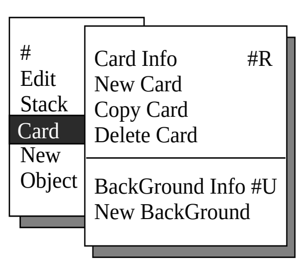

Figure 28: The *Card* menu.

7.3.3 Backgrounds

Although a number of cards may share the same background and a card may be given a new background, there is no explicit command to change a card's background to that of another. Instead, the card can be copied to the clipboard, then pasted back onto the stack while a card with a different background is the current card. The original card can then be deleted.

7.4 Visibility

Objects can be visible or invisible. Invisible items do not respond to any user action, however an object can always be selected when the stack is in edit mode even if it is invisible. An item can be made invisible by selecting *Hide* from the *Object* menu (see Figure 26). It can be made visible again by selecting *Show* from the same menu.

7.5 Editing Other Objects

7.5.1 Cards

So far we've discussed how you can create and edit different card objects on a stack, but we haven't talked about modifying cards, backgrounds, or stacks.

If you want to add another card to your stack, you can do so with the *New Card* option from the *Card* menu (see Figure 28). The new card will be added just after the current card and then displayed as the current card. Note that this makes it impossible to add a card before the home card.

The new card inherits its background from the previously visible card. There is no explicit option which allows you to change the order of the cards once they are created. This means that the order of the cards in the stack is fixed, unless you copy and paste a card to a new position in the stack.

The current card can be deleted by selecting *Delete Card* from the *Card* menu. However, you can never delete the home card of a stack. This means that the stack will always be at least one card long. Similarly, you can copy the current card to the clipboard using *Copy Card* from the same menu.

Selecting *Card Info* (`#R`) from the *Card* menu will show details of the card on the CardInfo stack (see Figure 29). This stack allows the card's name to be changed and scripts to be defined.

7.5.2 Backgrounds

Recall that a background can be shared between cards, so that you can create a common button (for example) only once and place in on the background instead of on each card that will use it. However, in your more elaborate stack designs you may want to have more than one background. To create a new background select the *New Background* option from the *Card* menu (see Figure 28). The current card will get a new background, and any card created after this card will share the same background.

The *Background Info* (#U) option from the *Card* menu will show details of the background on the BackGroundInfo stack (see Figure 30). This stack allows a background's name to be changed and scripts to be defined for the background.

7.5.3 Stacks

You may have noticed that when you edit a stack, on some occasions your commands will seem to do nothing and you won't be able to select card objects. The first thing you should do is check the StackInfo stack, shown in Figure 31, by selecting *Stack Info* (#T) from the *Stack* menu, in Figure 32. It has little in common with the other info stacks. On the info stack you will find information about the number of objects on the stack, background, and current card. You will also find the creator, the filename of the stack, and the name and Unix process ID of the client if one exists.

The StackInfo stack also has the following checkboxes:

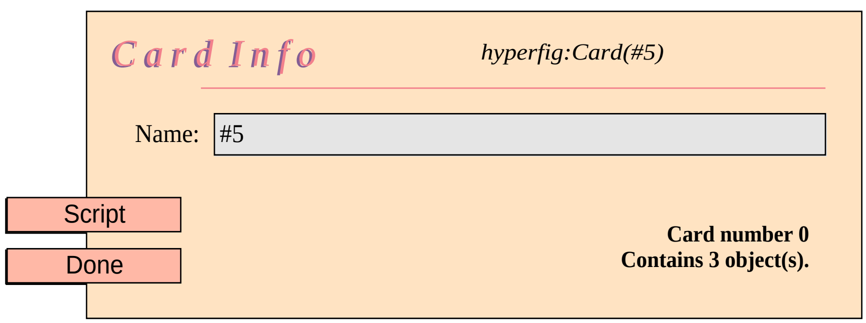

Figure 29: The CardInfo stack.

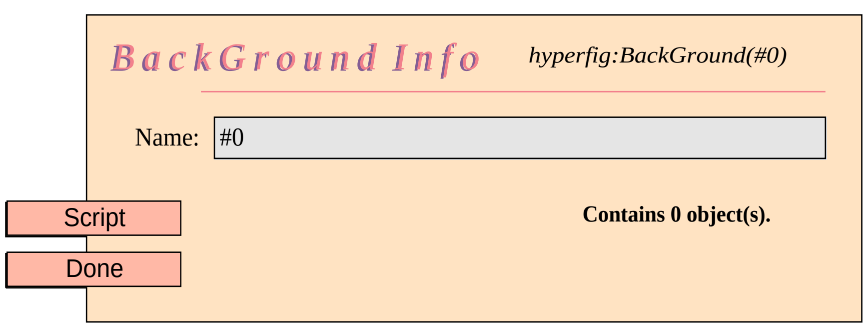

Figure 30: The BackGroundInfo stack.

Editable

If set, this flag will let you edit the stack. If unset, you will not be able to perform editing commands (or even select card objects) on the stack. Most system stacks do not have the *Editable* flag set, so you will have to set it before you can edit the stack. If this checkbox is selected, the following four checkboxes are also displayed:

Fixed position

If set, the stack cannot be moved on the screen using the AdjustButton of the mouse.

Retained

If this is selected, the stack drawing is saved in memory to provide faster updating when the stack is obscured and then uncovered.

SaveBehind

When set, the screen behind the stack is saved in memory for fast updating.

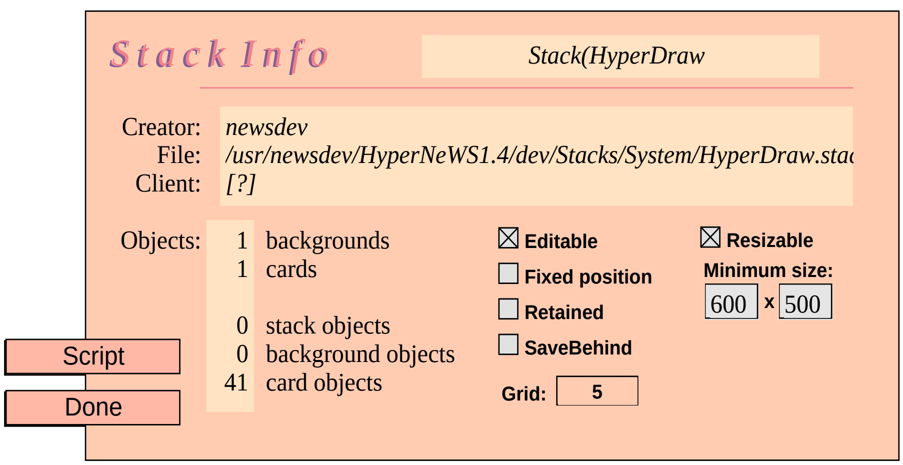

Figure 31: The StackInfo stack.

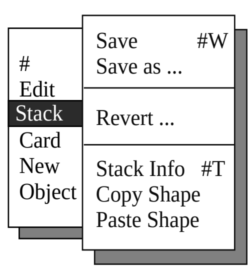

Figure 32: The *Stack* menu.

This is typically used when the stack will only be displayed for a short time.

Grid

If selected, card objects on the stack will cling to an invisible *snap grid* that makes object alignment easier.

Resizable

If this is set, it means the stack will respond to the *OnSizeObj* message, used for resizing stacks. The values below the checkbox are the minimum allowed width and height for the stack, so if you attempt to make a stack smaller than allowed, the rubber band will only get as small as the dimensions on this info stack.

Note that the checkboxes following *Editable* are only displayed if *Editable* is set. In order to modify the script of the stack, click on the *Script* button, which will be active whether or not *Editable* is set.

Unlike cards and backgrounds, it is not possible to change the name of a stack using the StackInfo stack. New stacks are created using the *Create New Stack* system button. Stack names are linked to the filename, and are generated automatically. The only way to change the name of a stack is to use the *Save* or *Save As* option from the *Stack* menu. You will be asked to specify a stack name if the stack you are saving is new (i.e. Untitled) or when you select the *Save As* option.

Some stacks can be iconified by clicking on the button in the upper-right corner of the stack. To hide a stack you can select *Zap* from the # menu, or press the *Done* button if it has one. If any changes have been made to the stack, HyperNeWS will ask:

Do you want to throw away changes to this stack?

If you do not wish to throw away the changes, then save the stack. Note that throwing away changes will not cause the memory copy to be changed; the stack will no longer be marked as updated and will not be flagged when exiting HyperNeWS (see Section 4.2.4). The next time the stack is opened it will appear exactly as it was before it was hidden.

Don't be afraid to change and save any of the HyperNeWS system stacks. The stacks will be saved in your `~/Stacks` directory (if it exists) after you have changed them, and they won't overwrite the existing HyperNeWS stacks. The originals will always be available in the HyperNeWS directories.

7.6 Designing Objects

7.6.1 Buttons

By now you've read about the four different button types: Pushbutton, Checkbox, Transparent, and Drawing. Using HyperDraw, you can create buttons of type Drawing with a unique look for your stack. The steps to create such a button are:

1. Create a drawing using HyperDraw. (Details on HyperDraw are in Section 5.11.)
2. Select the whole drawing.
3. Copy the drawing to the clipboard.
4. On your stack, select the button which you want to paste the drawing into.
5. Paste the drawing into your stack using the *Paste* option from the *Edit* menu. The stack of course must be in edit mode.

You can also specify a sequence of drawings for the button to cycle through each time the button is pressed. If you make a group out of objects of each drawing you want to display in a button, and then copy & paste all the groups into a button, you will be able to cycle through them in the order in which they were grouped. See Figure 33 for examples, and look at the ButtonIdeas stack for those and other examples that you can copy to HyperDraw or a button on your stack.

7.6.2 Stacks

Stack shapes can also be made easily using HyperDraw. You need to make a stack shape drawing which will form the basic window for the stack. The drawing can include any of the drawing tool features including images and EPSF drawings.

Once you have created the stack shape drawing, copy it to the clipboard and use *Paste Shape* from the *Stack* menu to set the stack window shape. You can

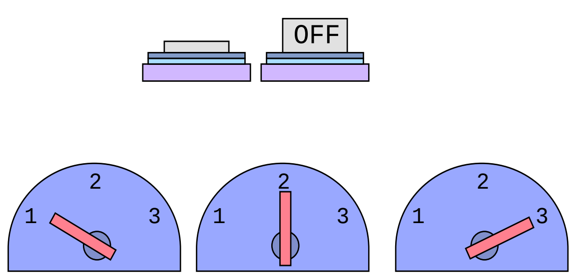

Figure 33: A sequence of button images.

also steal the shape from an existing stack by using *Copy Shape* from the *Stack* menu of the desired stack to copy its shape to the clipboard. Then use *Paste Shape* to change your own stack window, or paste it to HyperDraw and make some modifications before doing a *Copy* into the clipboard and a *Paste Shape* onto your stack.

8 Message Passing

8.1 Basics

HyperNeWS uses the object-oriented model of objects, classes and message passing, so that messages may be passed between any HyperNeWS objects. If the receiving object cannot handle the message, it is sent on to its parent, as defined in the message hierarchy. The parent then either handles it or passes it up the hierarchy. If at the stack level the message cannot be handled, it is sent on to a client program (if one exists) connected to the stack. Otherwise no action occurs. The HyperNeWS Reference Manual discusses how to connect HyperNeWS with client programs. If you want to see what messages are being passed in the system, turn message logging on (see Section 4.2.4). Figure 34 shows how events are passed through the HyperNeWS message hierarchy.

8.2 The PostScript Environment

NeWS is a multi-processing system, which makes it both very powerful and potentially difficult to use. The core HyperNeWS system tries to make life as simple as possible for the programmer by ensuring that only one thread of control is active at any one time. This means that the system does not use the NeWS `pause` primitive, and that whenever a process is forked for animated interaction, the calling process waits for it.

When the system is inactive and waiting for input, each canvas has an event

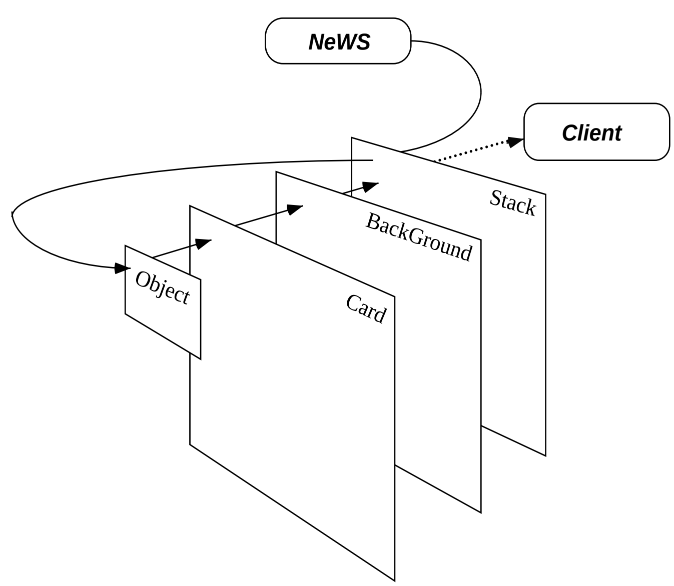

Figure 34: How events are passed through the message hierarchy.

manager process waiting for input. If clients are attached, each client has a PostScript process listening on its input stream. When an event arrives, the relevant canvas or client forks a process to handle it. This process runs without pauses. It can take action on any stack.

Passing a message from one object to another and the subsequent return of control has the following effect on the PostScript environment of the active process:

- The graphics state is unchanged.
- The execution stack is preserved.
- The dictionary stack maintains the system and HyperNeWS dictionaries. The current object's dictionary is popped and the dictionary of the object receiving the message is pushed in its place.
- The operand stack is cleared and the message arguments are stacked for the receiving object. This means that any data on the operand stack before a message is sent is not accessible to the receiving object.

Figure 35 illustrates the changes that take place in the PostScript environment when a message is sent. On return, the sender's object dictionary is reinstated and the operand stack is returned to its previous state, prior to the call. Note therefore, that sending a message to an object *cannot* be used as function call to return values. Message passing is not reliable in that it is not possible to find out whether or not a message was handled.

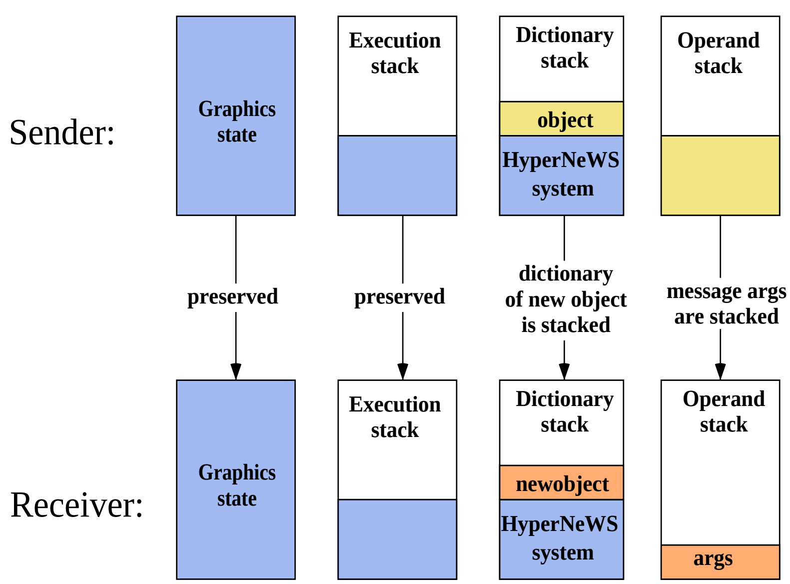

Figure 35: The PostScript state during message passing.

8.3 Events from HyperNeWS

8.3.1 Basics

Events from HyperNeWS are all handled at the HyperNeWS stack level. On receiving mouse or keyboard input the stack decides what action to take. Table 1 illustrates stack-level decisions and actions performed.

From Table 1 it is worth noting the following points:

- Clicking the left mouse button over a card object (in browse mode) sends that object an *OnMouse* message. Section 8.3.2 describes what action is taken by each basic object type on receipt of the *OnMouse* message.
- Only one EditText object is in focus at any given time. ASCII input (via the *OnAscii* message) is directed to the currently active EditText object (if one exists and is editable), otherwise the input is sent to the card.
- Canvas objects are an exception in that they do not receive an *OnMouse* message. In browse mode, moving the cursor into and out of the canvas generates the *OnEnter* and *OnExit* messages, respectively. When inside the canvas, clicking one of the mouse buttons sends a *OnLeft*, *OnMiddle*, or *OnRight* message to the Canvas object.

NeWS event	Location	Situation	Action
Left click	–	Control key held down	Toggle edit mode
Left click	Card object	Edit mode	Select object
Left click	Card object	Browse mode	/OnMouse to object
Left click	Stack	Browse mode	/OnMouse to card
Middle click	–	Browse mode	–
Middle click	Card object	Edit mode	Select object
Middle drag	–	Browse mode	Stack moved
Middle drag	Card object	Edit mode	Object moved
Right click	–	Browse mode	Short menu
Right click	–	Edit mode	Long menu
Right click	–	Edit + Obj selected	Object long menu
Ascii char	–	Browse mode	/OnAscii(char) to focus
Ascii char	–	Edit mode	–
Enter	Card	Browse mode	/OnMouseEnter to card
Exit	Card	Browse mode	/OnMouseExit to card

Table 1: Stack-level handling of NeWS events.

8.3.2 Messages Sent to a Card Object

This section describes what messages and arguments are sent to the card object by itself, when it receives an *OnMouse* message.

Push, Transparent buttons

Sends itself the *Action* message with integer argument 0.

Checkbox button

Sends itself the *Action* message with an integer argument 0 or 1, based upon the checkbox being reset or set, respectively.

Drawing button

The button's value rotates in sequence from 0 through N-1, where N is the number of drawings representing the button. A button with four positions would return 0, 1, 2, 3, 0, 1,... on respective clicks. The value is sent as the argument of the *Action* message.

Slider

If the *Editable* attribute is set, then the new value of the slider is returned as the argument of the *Action* message.

PullDown

Sends itself the *Action* message with the menu selection passed as a string argument.

ColorSelect

Colors are represented as an array of four values (red, green, blue and a grayscale), each ranging from 0 to 1 inclusive. A color is returned as a four-element array argument of the *Action* message.

Canvas

Canvas objects do not receive the *OnMouse* message. However, they generate a number of different messages, each without arguments. Moving the cursor into a canvas causes the *OnEnter* message, moving out sends the *OnExit* message. While inside a canvas, clicking left, middle, or right mouse buttons sends the messages *OnLeft*, *OnMiddle*, or *OnRight*, respectively. To find out where any of the mouse clicks occurred, you can access the global variables `LastX` and `LastY`.

EditText

EditText objects can be set up to return a single word (in the *WordAction* message) or a whole line of text (in the *Action* message). What is returned is a function of whether the text is *Editable*, *One line*, and whether a single- or double-click is performed. Table 2 illustrates exactly what is returned under different conditions.

8.4 Changing Default Object Behavior

The default behavior of any HyperNeWS object may be redefined by the user. This can be performed by writing your own methods to handle messages that are sent to an object. Making use of the message hierarchy, you can redefine an object's default behavior by defining your own response to a message. This is done by defining a PostScript executable procedure (or method) for a particular message. See Section 9 for further details on writing methods within an object's script.

Common messages sent to objects which may be redefined include:

Action

Sent to all card objects except canvases when the object receives an *On-Mouse* message as the result of a PointButton click. It returns the object's current value.

WordAction

Sent to EditText objects as the result of a double-click. It returns the actual word selected from the text, making it useful for hypertext functions.

OnMouse

Sent to card objects (not canvases) as the result of a PointButton click. Redefining this method will override the default behavior described in Section 8.3.2.

Editable	One line	Click	Result
yes	–	single	–
yes	–	double	WordAction(word)
no	yes	single	–
no	yes	double	Action(line)
no	no	single	Action(line)
no	no	double	Action(line) + WordAction(word)

Table 2: EditText messages.

OnAscii

Sent to either the currently focused EditText object, or to the current card when keyboard input is typed. One message is sent for each character, with the character code as its integer argument.

OnEnter, OnExit, OnLeft, OnMiddle, OnRight

Specific to canvas objects. None of the messages have any arguments. You can, however, determine where a mouse button was clicked on a canvas by checking the global HyperNeWS variables, `LastX` and `LastY`.

OnInit

Sent to a stack when it is first loaded into memory.

OnOpen

Sent to a stack when it is opened, then sent on to the visible card, its background and all the objects on the card. When moving from one card to another, the new card and all objects on it receive the *OnOpen* message. It is sent to an object whenever it becomes visible.

OnClose

The complement of the *OnOpen* message, it is sent to an object before it disappears. When moving from one card to another, all objects on the old card and then the card itself receive the message. When closing a stack the card objects, card, background and finally the stack itself receive the *OnClose* message.

9 Writing HyperNeWS Methods

9.1 Basics

HyperNeWS methods are currently written in PostScript. For those not familiar with the language it is suggested you take a look at two books by Adobe Systems Incorporated, called "PostScript Tutorial and Cookbook" and "PostScript Language Reference Manual" (also known as the "Blue Book" and "Red Book" respectively) before continuing with this section.

A number of methods are defined as part of the HyperNeWS system. These include functions for addressing objects, sending messages, creating new objects, and setting object instance variables. See the HyperNeWS Reference Manual for a description of the applicable methods and relevant instance variables. Both the system methods and the instance variables are organized by class type, with parent methods and variables inherited by the class below.

9.2 Scripts

Methods are defined collectively for an object using a *script*. Methods are defined for card objects, cards, backgrounds and stacks alike, all of which use the same ScriptInfo stack (shown in Figure 36). The figure shows a script containing three definitions: *Action*, *NextCard*, and *PrevCard* for the Button object on the ExampleObjects stack. When the button is pressed, the resulting *Action* message is handled by the object, causing the text message "hello world" to be printed on the console before setting the Slider object to the value 69.

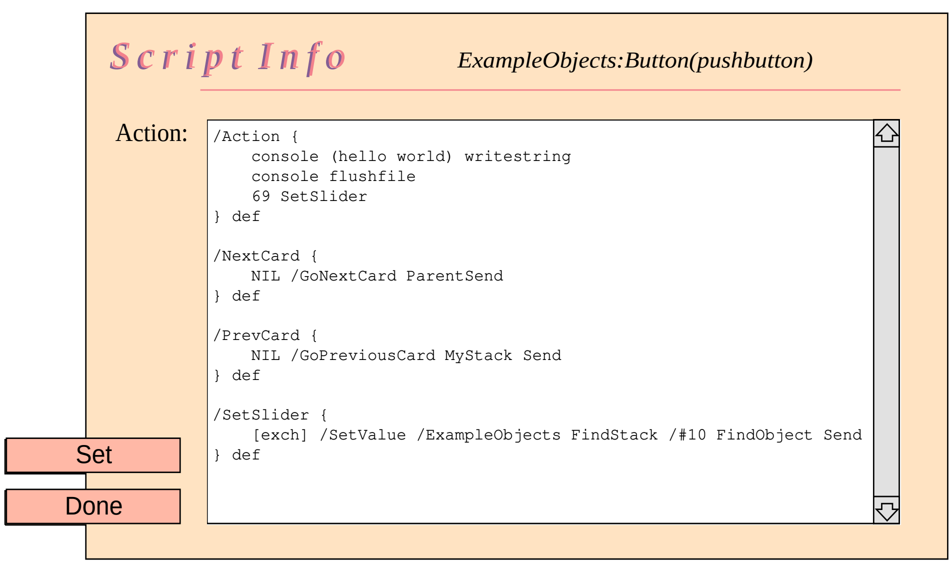

Figure 36: The ScriptInfo stack.

The ScriptInfo stack for all objects is accessed the same way. Bring up the info stack for the chosen object, and press the *Script* button on the info stack. In addition, scripts for card objects can be viewed by choosing *Script* (`#S`) from the *Object* menu.

9.3 Syntax

As we have mentioned before, HyperNeWS methods are represented in terms of PostScript procedure definitions. As with all PostScript routines, each procedure must have a unique key, a procedure enclosed by curly brackets, and defined using the `def` primitive:

```
/Keyname  {
      ... postscript code ...
} def
```

The PostScript code is executed when the object receives a message whose name matches the procedure's name. Scripts are a mechanism for defining dictionary key/value pairs for an object. All PostScript constructs must be defined using the `def` operator.

Each method must have a unique name within the object. Objects are implemented using PostScript dictionaries, holding scripts as key/value entries. When an object receives a message, the object's dictionary will be held as the current dictionary on the dictionary stack. This means that one method can reference another definition by its key name as illustrated in Figure 36. Here the *Action* method calls *SetValue* by referencing it by name. You cannot define a method with the same name as an existing HyperNeWS system method. If you attempt to do this, you will be informed which methods are illegally named and be asked to change them.

After looking at some script examples, you'll probably recognize the `FindStack` and `FindObject` routines used for finding objects in HyperNeWS. Each command has two forms. If you want to access a stack once, you can send it a message like `Show` using the syntax:

```
NIL /Show /MyStack Send
```

while if you plan to reference the stack often or if you want to do some things to the stack before showing it, it is slightly more efficient to do:

```
/s /MyStack FindStack def
% do some things
NIL /Show s Send
```

Similarly, if you want to send a message to an object on the same stack, this syntax will suffice:


```
[arg] /Doit /Thingie Send
```

while if `Thingie` is on another stack, say `Stack2`, you'll need to lookup the object using this syntax:

```
/Stack2 FindStack /Thingie FindObject
% object is now on operand stack
[arg] /Doit 3 -1 roll Send
```

Of course, if you had found the stack `Stack2` earlier, you could have done the same as the example with `MyStack`, and use the syntax:

```
[arg] /Doit s /Thingie FindObject Send
```

9.4 Script Examples

In many cases it is not necessary to write your own scripts. If you are using a HyperNeWS stack as a panel with mouse-sensitive objects on it, then the default *Action* message received by the object provides enough information for the client to act on. Consider a slider named *slide* that has its value set by the user to 35. The default message which the slider sends itself is *35 Action*. Since no object has a built-in *Action* method, this message will trickle up to the parent, which will be told that the message was sent by the slider. If we wish to give this message another name (such as *ValueChanged*), this can be done by the following script in the Slider object:

```
/Action { % integer --
    [exch] /ValueChanged Self Send
} def
```

Notice that we sent the message back to the slider. This is the best thing to do in general, as it allows us to take advantage of the message hierarchy. Every object above the slider, including the slider itself, will now have an opportunity to intercept this message, and either act on it or change it before it reaches the client.

This example was somewhat contrived. There are occasions when actions can be handled more conveniently inside HyperNeWS than from the client. Let us assume that we have a stack with several cards, each of which contains a set of objects for providing the client with parameters. There is no need for the client to know that there is more than one card. A button on the first card, with a label saying "remote" for instance can move us to a card where all the remote parameters are organized. Let us assume that we have named this card *remote_card*. The button script could then be:

```
/Action { % number --
    [/remote_card] /GotoCard ParentSend
} def
```

Notice that in this case we don't send the message to the button itself, since *GotoCard* is a system message that you shouldn't override. In fact we could (if we know the internals of HyperNeWS well) send the message directly to the stack, which handles this message:

```
/Action { % number --
    [/remote_card] /GotoCard MyStack Send
} def
```

You'll notice from the ScriptInfo stack that we used two different ways of sending a message. The first uses `ParentSend` and the second uses `MyStack Send`. Both will work equally well. The former may take a little longer, because it has to check the Button, Card, and Background classes for the methods. In that respect, the latter technique is better. Also, the latter example offers a bit of online documentation by showing which class has the first chance to accept the message.

Finally, a word of warning about writing scripts when clients exist. Let's say that there is a counter that the client sets, but that on some occasions (like this one) the stack has more accurate information and you want the stack to set the counter to 25 after the client has set it. So you tell the client to set the counter (as part of the `SetUpRemote` message), and then you set the counter yourself in the stack with the `SetValue` message. The script might look something like:

```
/Action { % integer --
    NIL /SetUpRemote ClientSend
    [25] /SetValue /Counter Send
} def
```

This may cause problems. The problem is that the client is a separate process, and you have no guarantee that the desired actions will be performed sequentially. With two processes going on simultaneously HyperNeWS can end up in a peculiar state. In this case the client should send a message saying basically "I'm finished" to the stack so the stack can then set the counter without worrying about the client setting it to something else moments later.

10 Index

