Chapter 2. The Interactive Graphical Interface

Authors: *Joseph T. Buck*
Edwin E. Goei
Wei-Jen Huang
Alan Kamas
Edward A. Lee

Other Contributors: *Andrea Cassotto*
Wan-Teh Chang
Michael J. Chen
David Harrison
Holly Heine
Phil Lapsley
David G. Messerschmitt
Rick Spickelmier

2.1 Introduction

The Ptolemy interactive graphical interface (**pigi**) is a design editor for Ptolemy applications. It is based on tools from the Berkeley CAD framework. In **pigi**, Ptolemy applications are constructed graphically, by interconnecting icons. Hierarchy is used to manage complexity, to abstract subsystem designs, and to mix domains. This chapter explains how to use version 0.5.

2.1.1 Setup

Pigi requires the MIT X Window System. If you are not familiar with this system, see the appendix, "Introduction to the X Window System" on page B-1.

On most systems, when Ptolemy is installed, a fictitious user named '**ptolemy**' is created. If such a user exists on your system, then all you need to run Ptolemy is for the directory **~ptolemy/bin** to be in your path. For most users, you can ensure this with the following command in your **.cshrc** file in your home directory:

```
set path = (~ptolemy/bin $path)
```

After adding this line to your **.cshrc** file, you will need to source it:

```
source .cshrc
```

If Ptolemy has been installed without creating a '**ptolemy**' user, then you must set the environment variable **PTOLEMY** to point to the main directory of the Ptolemy installation. If Ptolemy is installed in **/usr/eesww/share/ptolemy0.5**, for example, your **.cshrc** file should contain the commands:

```
setenv PTOLEMY /usr/eesww/share/ptolemy0.5
set path = ($PTOLEMY/bin $path)
```

In the documentation, we will generally refer to the home directory of the Ptolemy installation as **$PTOLEMY**, but sometimes we forget and use **~ptolemy**.

If you are going to do extensive software development using Ptolemy, it may be helpful to also put the directory **$PTOLEMY/bin.$ARCH** in your path, where **$ARCH** specifies the type of machine you are running on (**sun4**, **mips**, or **hppa**).

Some window managers are configured to require that you click in a window before the "focus" moves to that window. This means that the window will not respond to input just because you have placed the mouse cursor inside it. You must first click a mouse button in the window. While it is possible to use **pigi** with this configuration, it is extremely unpleasant. In fact, it will be rather unpleasant to use *any* modern program that makes use of the window system. You will want to change the mode of the window manager so that the focus follows the mouse. The precise mechanism for doing this depends on the window manager. For the Motif window manager, **mwm**, the appropriate line in the .Xdefaults file is:

```
Mwm*keyboardFocusPolicy:   pointer
```

For the open-look window manager, **olwm**, the line is:

```
OpenWindows.SetInput:      followmouse
```

Alternatively, you can invoke **olwm** with the option **-follow**. Typically, the window manager is started in a file called **.xinitrc** in your home directory.

If you are running Sun's Open Windows, you may find that the Athena widgets have not been installed; **pigi** will not run without them. See the installation instructions in the appendix. For more information on using **pigi** with Open Windows, see "Introduction to the X Window System" on page B-1.

If your system has X11 installed in an unusual place, then you will need to set an environment variable for some of the the CGC (code generation in C) demos to compile correctly. That environment variable is **PTX11DIR**. It should specify a directory that contains as subdirectories **lib** and **include**, containing the X11 libraries and include files.

2.2 Running the Ptolemy demos

A good way to start is by running a few of the Ptolemy demos. Any user can do this, although average users are not permitted to change the demos. If you feel compelled to change a demo, you can copy it to your own directory by using "**cp -r**" (see the section below, "Copying objects" on page 2-40). You can modify the copied version.

2.2.1 Starting Ptolemy

In any terminal window, change to the master demo directory:

```
cd ~ptolemy/demo
```

or if there is no ~ptolemy,

```
cd $PTOLEMY/demo
```

Start the Ptolemy graphical interface:

```
pigi &
```

You should get three windows: a **vem** console window at the upper left of your screen, a palette with icons below that, and a message window identifying the version of Ptolemy, as shown in figure 2-1. The borders on your windows may look different, since they are deter-

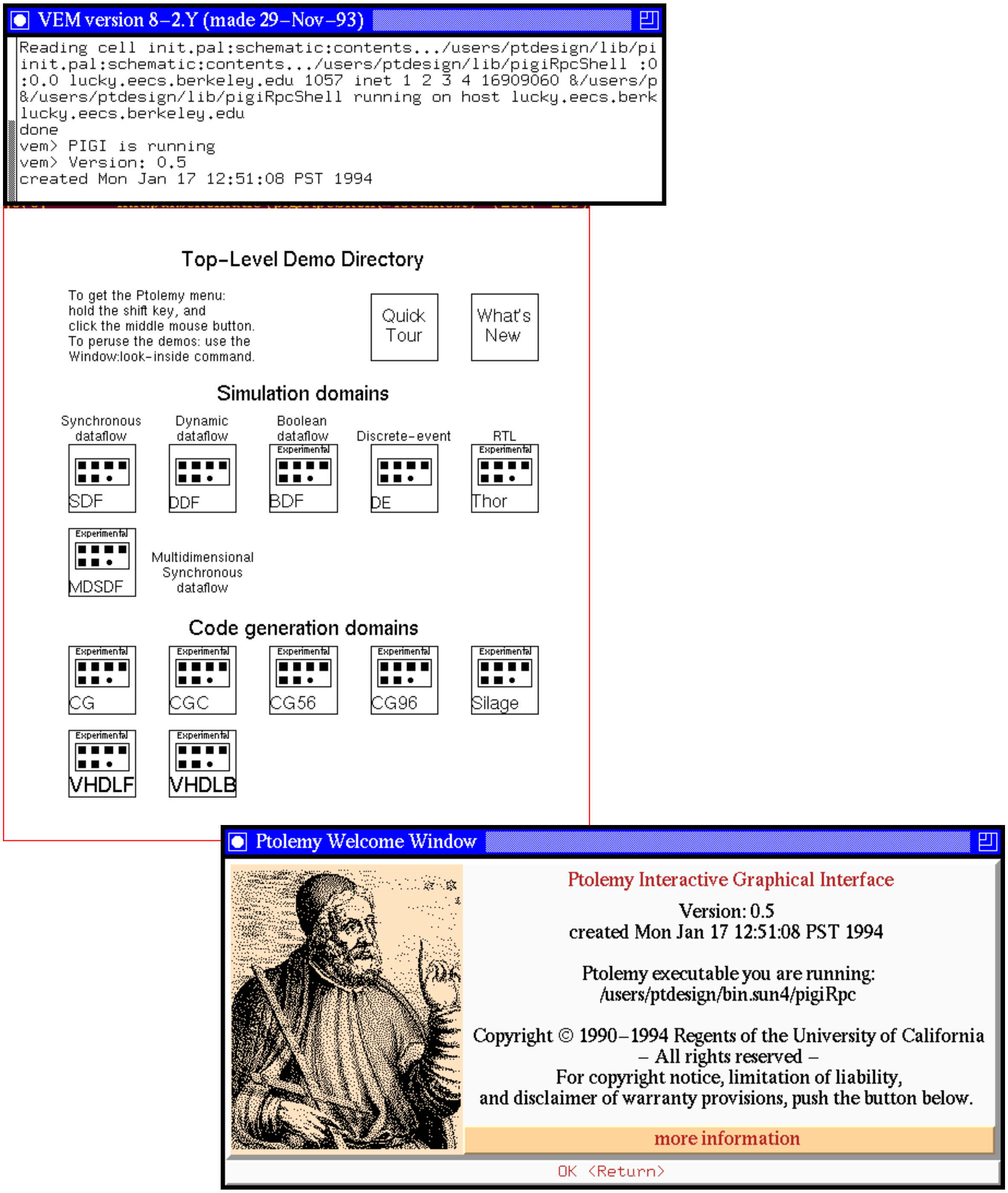


FIGURE 2-1: If you start pigi in the directory $PTOLEMY/demo, once the system is started you will see these three windows. The upper left window is the vem console window. Below that is a palette of icons representing demo directories. To the right is the Ptolemy welcome window.

mined by the window manager that you use. A complete list of options that you can specify on the command line is given in the section “Command-line options” on page 2-47.

Once you get all three windows, you have started two processes: the graphical editor, **`vem`**, and a process named **`pigiRpc`** that contains the **`pigi`** code and the Ptolemy kernel. You can largely ignore the **`vem`** console window. It is used primarily for capturing debugging messages.

Clicking any mouse button in the welcome window (the one with the picture of Prof. Ptolemy) will dismiss it. Clicking the left mouse button on the “more information” button will display copyright information. The remaining windows can be moved and resized using whatever mechanism your window manager supports. The windows can be closed by typing a control-d with the mouse cursor inside the window. Closing the **`vem`** console window will terminate the entire program.

For reference, a summary of the pertinent terms is given in table 2-1 on page 4. These will be discussed in more detail as we go.

The palette window contains icons. Five different types of icons are used in **`pigi`**, as shown in figure 2-2. The ones in the palette window are of the first type; they represent other palettes. If you have a color monitor, the outline on these icons is purple.

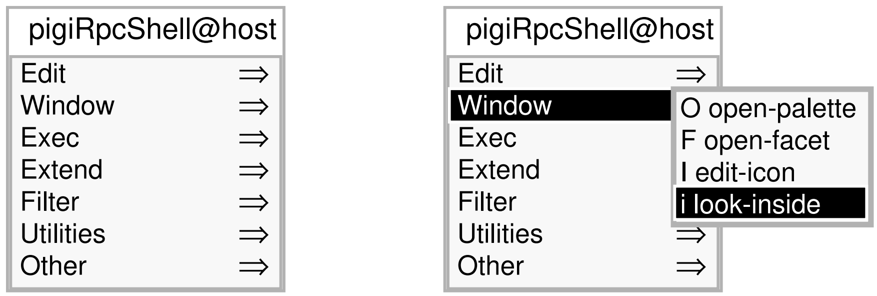
2.2.2 Exploring the menus

Place the mouse cursor on the icon labeled “SDF”. Get the **`pigi`** command menu by holding the shift key and clicking the middle mouse button. This menu is of a style called a “walking menu.” Make sure you hold the shift button. The resulting command menu is shown

Category	Term	Definition
Programs	**Ptolemy**	The entire design environment
	`pigi`	The Ptolemy graphical interface, including both **`pigiRpc`** and **`vem`**
	`vem`	A graphical editor for oct, upon which **`pigi`** is built
	`pigiRpc`	A process attached to **`vem`** by **`pigi`**
Design database	**`oct`**	The design manager and database
	facet	A design object (a schematic or a palette)
	schematic	A block diagram
	palette	A facet that contains a library of icons rather than a schematic
Ptolemy objects	**Star**	Lowest level block in Ptolemy, with functionality defined in C++
	Galaxy	A block made up of connected sub-blocks, with inputs and/or outputs
	Universe	An outermost block representing a complete system that the user can run
	Domain	An object defining the model of computation, which defines the behavior of a network of blocks. In code generation, a domain also corresponds to single target language.
	Wormhole	A galaxy that does not have the same domain on the outside as the inside.
Tcl/Tk	**Tcl**	An interpreted language built in to **`pigi`**
	Tk	An X window toolkit attached to Tcl

TABLE 2-1: Summary of terms defining software components.

below:



The names displayed in the left main menu are only headers. To see the individual commands under each header, you must move the mouse to the arrows at the right of the menu. The sub-menu that appears on the right contains commands. Clicking any mouse button with a command highlighted as shown on the right will execute that command. To remove the menu without executing any command, simply click a mouse button anywhere outside the menu.

2.2.3 Traversing the hierarchy

Go to the "Window" sub-menu, and execute the *look-inside* command, as shown above on the right. A new palette will open, containing icons representing further palettes. Look inside the first of these, labeled "Basic". The icons inside contain application programs, called "universes" in Ptolemy. The two palettes you have just opened are shown in figure 2-3. They are both explained in further detail in "An overview of SDF demonstrations" on page 5-36.

Note in the Ptolemy menu that the *look-inside* directive has an "i" next to it. This is a "single key accelerator." Without using the walking menu, you can look inside any icon by simply placing the mouse cursor and hitting the "i" key on the keyboard. The single-key accelerators are extremely useful. In time, you will find that you use the menu only for commands that have no accelerator, or for which you cannot remember the accelerator. The Ptolemy commands obtained through the above menu are summarized in table 2-2. The few commands you will need immediately are shaded in table 2-2.

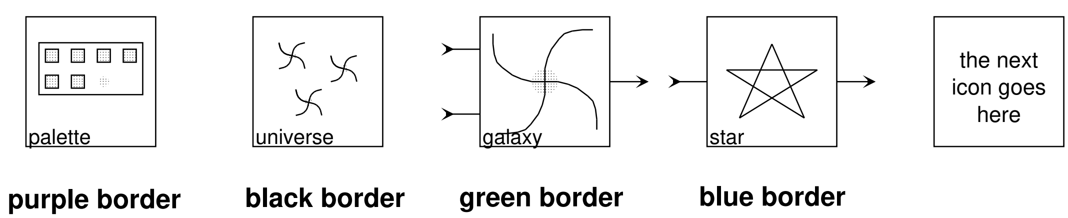


FIGURE 2-2: Five different types of icons are used in pigi. From left to right, the icons represent palettes (windows containing more icons), universes (windows containing Ptolemy applications), galaxies (functional blocks defined using other functional blocks), and stars (elementary or atomic functional blocks). The last icon on the right is the *cursor*, marking the position into which the next icon will be placed. On a color monitor, the borders of the icons have the indicated colors. The designs inside the icons and their shape are the default. They may be customized.

Look inside the first demo on the third row, labeled "**sinMod**". You will see the schematic shown in figure 2-4. Try looking inside any of the icons in this schematic. If you look inside the icon labeled "modulator", you will see the lower schematic in figure 2-4. If you look inside the icon labeled "XMgraph", this time, instead of graphics, you will see text that defines the functionality of the block. The syntax of this text is explained in the programmer's manual, volume 3 of the Almagest. You can change the editor used to display the text by setting an environment variable **PT_DISPLAY** (see "Environment variables" on page 2-46).

2.2.4 Running a Ptolemy application

To run the **sinMod** system using the walking menu, place the mouse cursor anywhere in the window containing the **sinMod** schematic. I.e., your cursor should be in the window

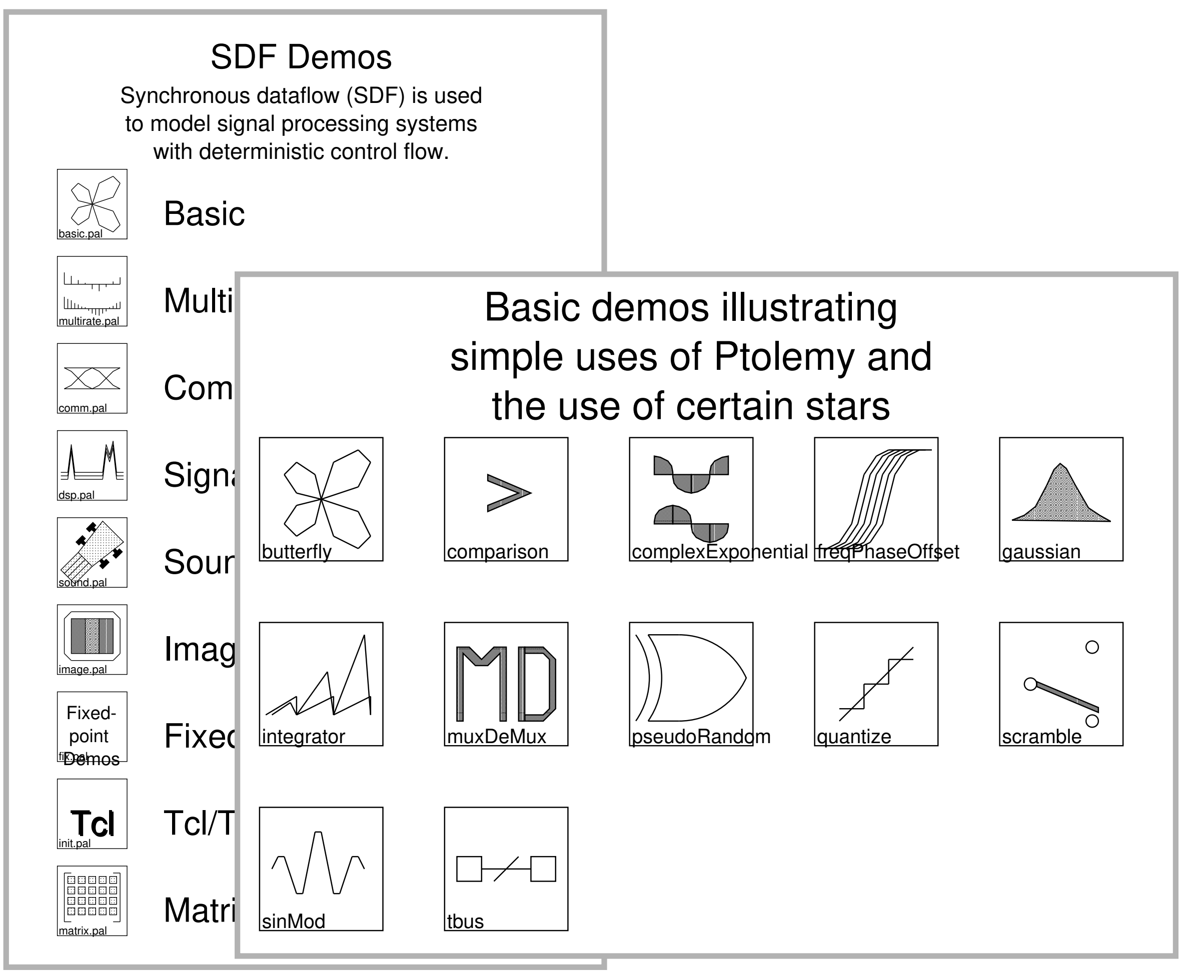
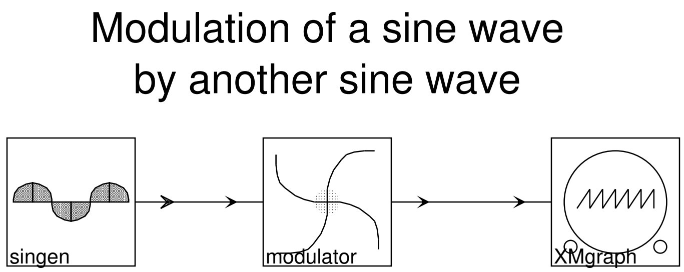


FIGURE 2-3: The "SDF" and "basic" palettes. The SDF palette contains icons representing other palettes containing a variety of demos in the synchronous dataflow domain. The "basic" palette is one such palette of demos. The icons here represent universes. These palettes are explained in more detail in "An overview of SDF demonstrations" on page 5-36

Menu	Heading	Command	Key	Description
pigi	Edit	edit-params	e	change parameters of a star, galaxy, or universe
		edit-domain	d	change the domain of a universe or galaxy
		edit-target	T	specify a target to manage the execution
		edit-comment	;	add comment to a universe or descriptor to a galaxy
		edit-seed	#	set the random number seed
		find-name		highlight a block with a specified name
		clear-marks		clear all icon highlighting
	Window	open-palette	O	open one of the standard palettes of blocks
		open-facet	F	open an arbitrary palette, universe, or galaxy
		edit-icon	I	modify the physical appearance of an icon
		look-inside	i	look inside an icon for its definition
	Exec	run	R	run a universe
		run-all-demos		testing command - run everything in a palette
		compile-facet		testing command - translate oct to Ptolemy
		display-schedule		show the most recent static schedule, if any
	Extend	make-schem-icon	@	make an icon to represent a facet
		make-star	*	dynamically link a new star and make an icon
		load-star	L	dynamically link a star that already has an icon
		load-star-perm	K	link a star so that derived stars can link dynamically
	Filter	equirripple FIR	<	invoke a provisional filter design utility
		window FIR	>	invoke another provisional filter design utility
	Other	facet number	H	testing command - display the Tcl facet handle
		man	M	open a manual page corresponding to a star
		profile	,	display a brief summary of the functionality of a star
		print-facet	cntr-P	print a facet or generate a PostScript file
		show-name	n	display the name of an icon and its master
		options		change various esoteric options
		version		display the version of Ptolemy that is running
		exit-pigi		quit Ptolemy without exiting vem

TABLE 2-2: A summary of the Ptolemy commands in the pigi menu, which is obtained by holding the shift button and clicking the middle mouse button. The single-key accelerators for commands that have them are shown. The commands that are most useful for exploring the Ptolemy demos are shaded.

that contains the following schematic:



Again holding the shift key, click the middle mouse button. Go to the “Exec” sub-menu, and select “run” by clicking any button. Notice that typing an “R” would have had the same effect.

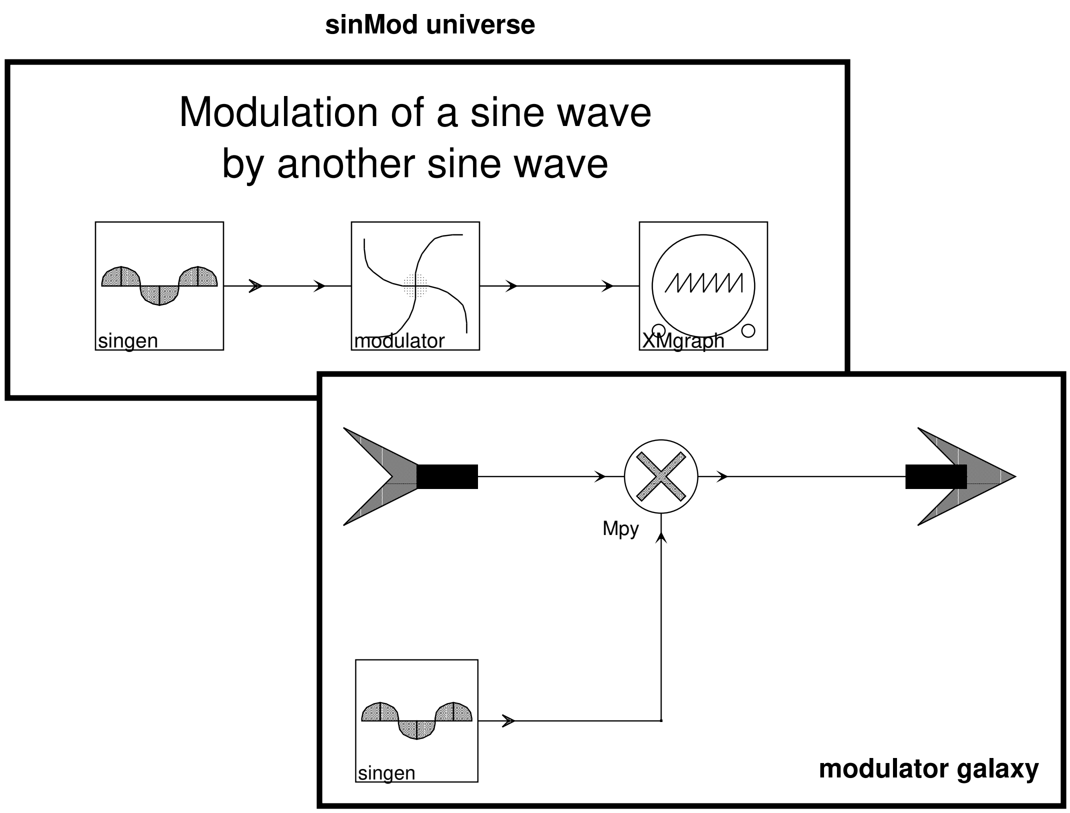
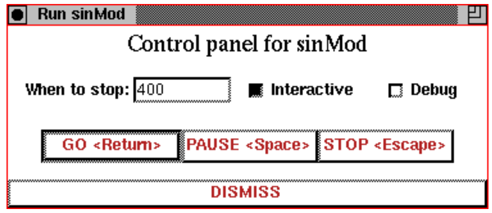


FIGURE 2-4: One of the synchronous dataflow demos. This Ptolemy application modulates a sine wave with another sine wave. The upper diagram is the top level. The lower is the contents of the “modulator” subsystem.

The following control panel pops up:



If you click the left mouse button on the “GO” button (or hit “return”), Ptolemy will run this application through 400 iterations. When the run is finished, a graph appears, as shown in figure 2-5. Try resizing and moving this display. Experiment in this “pxgraph” window by drawing boxes; to draw a box, just drag any mouse button. This causes a new window to open with a display of only the area that your box enclosed. Although the new window covers the old, if you move it out of the way, you can see both at once. Any of the now numerous open windows can be closed with a control-d.

2.2.5 Examining schematics more closely

Place the mouse cursor in any schematic or palette window, and click the middle mouse button without holding the shift key. The **`vem`** command menu, which is different from the **`pigi`** command menu, appears. This menu is the same style of “walking menu” as the

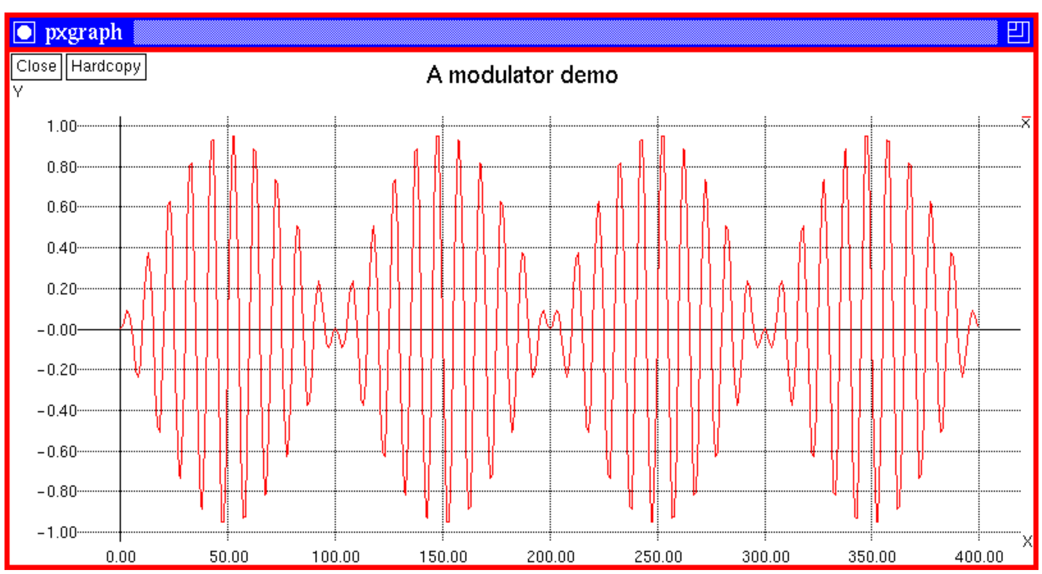
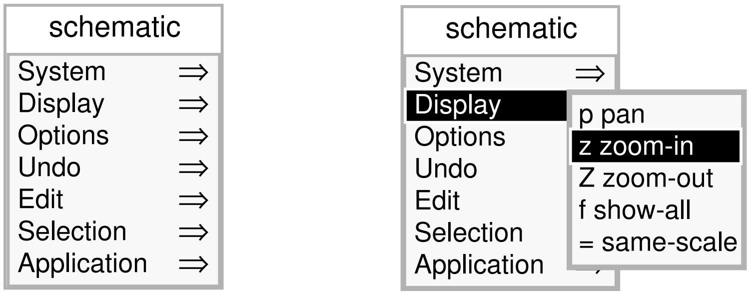


FIGURE 2-5: The graph generated by the “sinMod” application in figure 2-4. The graph is displayed by a program called “pxgraph,” based on xgraph by David Harrison.

pigi menu, and is shown below:



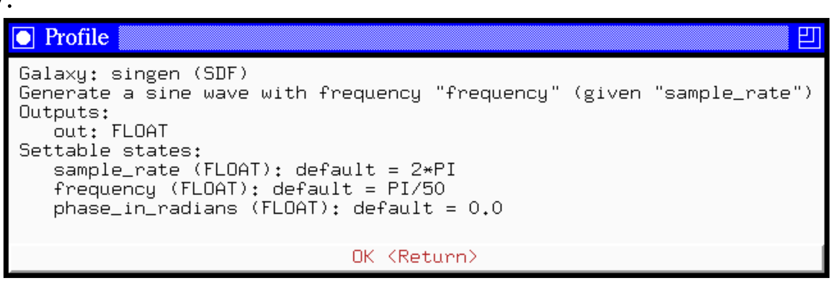
The **vem** menu is used for manipulating the graphical description of an application. The commands obtained through this menu are summarized in table 2-3, and explained in full detail in Chapter 20.

A few additional window manipulations will prove useful almost immediately. In any of the **vem** windows, you can closely examine any part of the window by drawing a box enclosing the area of interest and typing an “o”. Like in a pxgraph window, this causes a new-window to open, showing only the enclosed area. Unlike pxgraph windows, typing the “o” is necessary. In addition, you can enlarge a window using your window manager manipulation, and type an “f” to fill the window with the schematic. You can also zoom-in (or magnify) by typing a “z”, and zoom-out by typing a “Z” (see table 2-3 on page 2-11). These and other **vem** commands are referenced again later, and documented completely in chapter 20.

2.2.6 Invoking on-line documentation for stars

You may wish to understand exactly how this **sinMod** example works. There are several clues to the functionality of the stars. After a while, the icons themselves will be all you will need. At this point, you can get several levels of detail about them. First, you will want to know the name of each star. If you have closed the **sinMod** window, open it again. Notice the names that appear on each of the icons. In more complicated schematics, when the icons are much smaller, the names will not show. You can zoom-in on a region of the window to see the names. Alternatively, you can place the mouse on any icon and issue the “show-name” command (in the “Other” menu), or type “n”.

Find the **singen** block at the left of the **sinMod** schematic. To understand its function, place the mouse cursor on it, and execute the Other:profile command. Here “Other” refers to the command category and “profile” to the command in the submenu (you may also type “,”). This command invokes a window that summarizes the behavior of the block, as shown below:



For some blocks, further information can be obtained with the Other:man (“M”) command, which displays a formatted manual page. This facility, however, currently has many limitations. Try it on the **XMgraph** block at the right of the schematic. The man command requires that the program **xditview** be installed and in your path. It also requires a program **ditroff**[1]. If it fails to find these programs, the manual page will be formatted with nroff and displayed on the standard output, which is considerably less useful. If it fails to work, you are

Menu	Heading	Command	Key	Description
vem	none	no command name	cntr-h	remove the last argument (point, box, etc.)
			del	remove the last argument (point, box, etc.)
			cntr-u	remove all arguments from the argument list
			cntr-l	(control lower case L) redraw the window
	System	open-window	o	open a new view into a facet
		close-window	cntr-d	close a window
		where	?	find the position of the cursor in oct units
		palette	P	open the color palette for editing icons
		save-window	S	save a facet
		bindings	b	display key bindings (single key accelerators)
		re-read		restore a facet to the last saved version
	Display	pan	p	move the view to be centered at a given spot
		zoom-in	z	zoom in for a closer view of a facet
		zoom-out	Z	zoom out
		show-all	f	rescale the schematic to fit the window
		same-scale	=	used to get two windows to use the same scale
	Options	window-options		adjust snap, grid spacing, etc.
		layer-display		selectively display colors
		toggle-grid	g	turn on or off the grid display
	Undo	undo	U	undo any number of previous changes
	Edit	create	c	create a line, icon, name, etc.
		delete-objects	D	remove selected objects from an icon drawing
		edit-label	E	modify a label in a schematic
	Selection	select-objects	s	add an object to the argument list for a command
		select-net	cntr-n	select a wire (net) connecting blocks
		unselect-objects	u	remove an object from the argument list
		transform	t	rotate or reflect an object
		move-objects	m	move an object in a schematic
		copy-objects	x	copy one or more objects in a schematic
		delete-objects	D	delete objects from a schematic
	Application	rpc-any	r	start a vem application (pigiRpc is one)

TABLE 2-3: A summary of the Ptolemy commands in the vem menu, which is obtained by clicking the middle mouse button without holding the shift button. The single-key accelerators for commands that have them are shown. The commands that are most useful for exploring the Ptolemy demos are shaded. More complete documentation can be found in chapter 20, “Vem — The Graphical Editor for Oct” on page 20-1.

1. It is possible to replace ditroff with a freely available program called groff. See “Configuring your system to use the Gnu groff program” on page 19-11.

stuck with the old fashioned documentation on paper. Complete documentation of all Ptolemy blocks and demos is available in the “Star Atlas” volume of *The Almagest*. The ultimate documentation for any block is, of course, its source code. For the **`singen`** block, the source is another schematic. Use the “look-inside” command (using the accelerator key “i”) to see it. Recall that you can also look at the source code of the lowest level blocks (called *stars*) by looking inside them.

2.2.7 More extensive exploration of the demos

You can safely explore other demos in the palette by the same mechanism. The **`butterfly`** demo at the upper left of the “basic” palette in figure 2-3 is particularly worthwhile. The demos in this and other palettes are briefly summarized in “An overview of SDF demonstrations” on page 5-36, and more completely explained in the “Star Atlas” volume of *The Almagest*.

The **`init.pal`** palette in figure 2-1 contains icons leading to a top-level demo directory for each domain distributed with Ptolemy. Some of these are labeled “experimental”. These domains largely reflect research in progress, and should be viewed as concept demonstrations only. The mature domains have no such label, although even these domains contain some experimental work. A quick tour of the basic capabilities can be had by looking inside the icon labeled “quick tour” in the start-up palette shown in figure 2-1. Each time you encounter a universe, run it.

2.2.8 What’s new in version 0.5

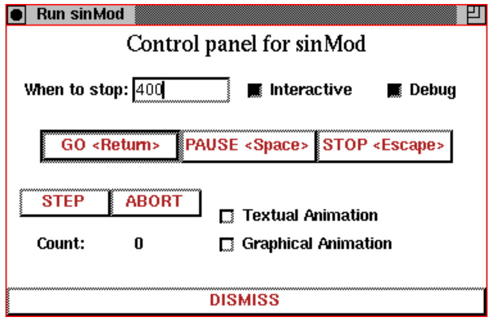
For readers familiar with previous versions of Ptolemy, you may wish to take a tour of the new features only. The second icon in the **`init.pal`** palette in figure 2-1 leads to such a tour. Look inside it, and explore the icons therein. Each time you encounter a universe, run it.

2.3 Dialog boxes

As you explore the demos, you will frequently encounter dialog boxes and control panels. For example, the run command opens a control panel like the one shown above that, among other things, allows you to specify how long the simulation should run. Most of the control panels that you will encounter have been designed using an X window toolkit called Tk, and every effort has been made to follow the Motif design style. Hopefully, this will look familiar to most people.

2.3.1 Tk control panels

Most of the items in a control panel are self-explanatory. Consider the run control panel shown on page 2-9. The button with the double relief (the GO button) is the default button. Hitting the return key has the same effect at clicking the mouse on this button. A different type of button is the “check button”, labeled “Debug”. Clicking on this button expands the control panel, as shown below, giving the user options that are sometimes useful in debugging a complex application.



The “Animation” buttons show (textually or graphically) which blocks are running at any given time. Graphical animation will dramatically slow down a simulation, so it is not advised except for occasional use. It is often useful in combination with the “STEP” button, which will fire stars one a time. The “ABORT” button differs from the “STOP” button in that the “wrapup” methods of the stars are not invoked. Thus, for example, signal plots that normally appear at the end of a simulation will not appear. Clicking on the “Debug” button a second time will reduce the control panel to its previous form.

Many control panels have text widgets. In the control panel above, for example, the box labeled “When to stop” is a text widget. To change the number, you must use Emacs-like editing control characters. These are summarized in table 2-4. In addition, using the mouse, you can position the cursor anywhere in the text to begin editing by clicking the left button. For example, to enter a new number for “when to stop”, position the cursor in the number box, type control-k followed by the new number. You can then push the GO button (or type return) to run the application the specified number of iterations.

Many control panels have more than one text widget. The current field is the one with the cursor, and anything you type will go into it. To change the current field to a different one, move the mouse or use the “Tab” key to move to the next one.

The button “Interactive” on the Control panel toggles interactive mode on and off. When interactive mode is on, pigi pauses periodically during the run to check if the user has moved the mouse or pressed a button like “pause” or “stop.” The run is interactive because pigi is always checking for user requests. Unfortunately, there’s no free lunch. The same

Key	Description
Delete, control-h	Delete previous character.
control-a	Move to beginning of line
control-b	Move backward one character
control-d	Delete next character
control-e	Move to end of line
control-f	Move forward one character
control-k	Kill (delete) to end of line

TABLE 2-4: Summary of key bindings for Emacs-style text editing.

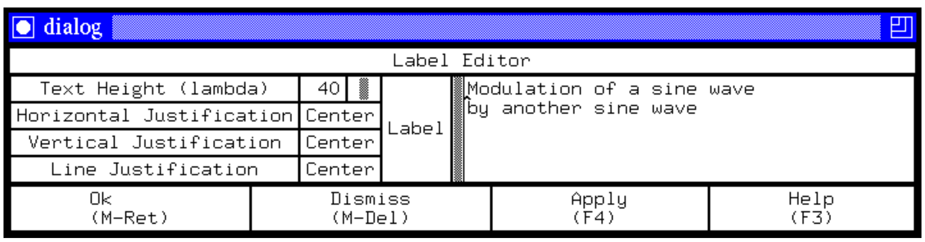
checking that enables pigi to interact during a run also slows the run down.

For faster runs, click on the “Interactive” button to toggle interactive mode off. When interactive mode is off, the “pause” and “stop” buttons are deactivated since the pigi will no longer check to see if a user may have pressed one of these buttons during a run. Thus, there is no clean way to abort the run once it starts. Many users find it best to run a system first in interactive mode and then, after they are sure that the system functions correctly, they get faster runs by turning interactive off.

Note that some of the Ptolemy stars that use Tk for dynamic displays or inputs depend on pigi being in interactive mode to work correctly. If windows appear during a run, but nothing is plotted in them, or if slider bars or other inputs seem to have no effect, try re-running the system with Interactive mode back on.

2.3.2 Athena widget dialog boxes

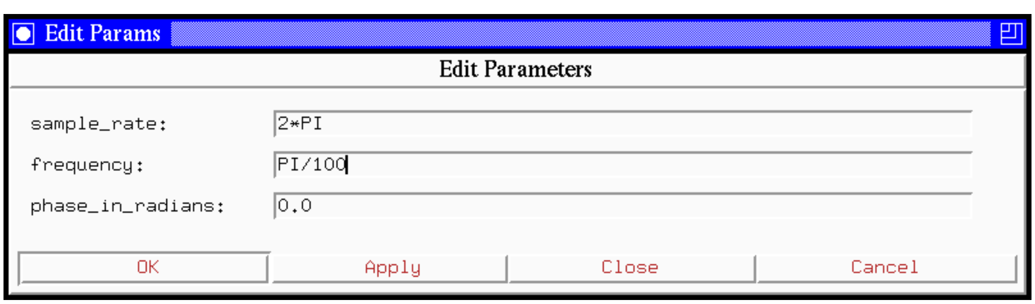
Although we have been working hard to eliminate them, a few old-style dialog boxes based on the Athena widgets from MIT still survive in the system. You will recognize these immediately, because they are much uglier and more difficult to work with than the Tk-based widgets. Here is an example:



The text widgets in these dialog boxes also use Emacs-style commands. However, do not type return; this adds a second line to the dialog entry, which for most commands is confusing at best. If you accidentally type return, you can backspace sufficiently to get back to one line. Meta-return is the standard way to invoke the “OK” button in these widgets.

2.4 Parameters and states

To see the parameter values of a star or galaxy, execute the Edit:edit-params command, which has the accelerator key “e”. The **singen** star in the **sinMod** application has the following parameter screen:



Notice that the *frequency* parameter is given as an expression, “PI/100” (PI represents the constant π). This section describes the expression language for specifying parameter values.

The parameter screen can be kept open while you experiment with different values of the parameters. Try changing the value “PI/100” to “PI/200”. Click “Apply” in the parameter window, and then “GO” in the run control panel. How does this change the display? Clicking “Cancel” in the parameter window will restore the parameter values to the last saved values and dismiss the parameter window. Clicking “Close” will dismiss the parameter window without restoring the parameter values.

2.4.1 A note on terminology

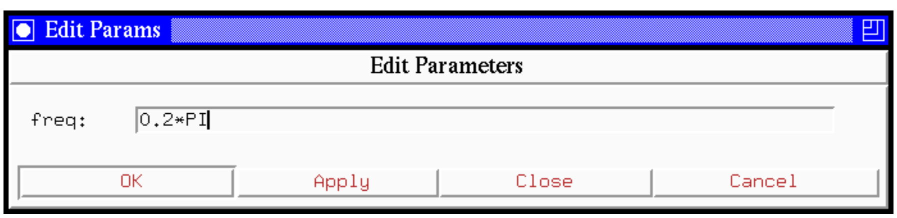
A **`State`** is a data-structure associated with a star, used to remember data values from one invocation to the next. For example, the gain of an automatic gain control is a state. A state need not be dynamic; i.e, its value may not change during the course of a simulation. Technically, a *parameter* is the initial value of a state. **`Pigi`** is responsible for defining parameter values and storing them in the design database.

2.4.2 Changing or setting parameters

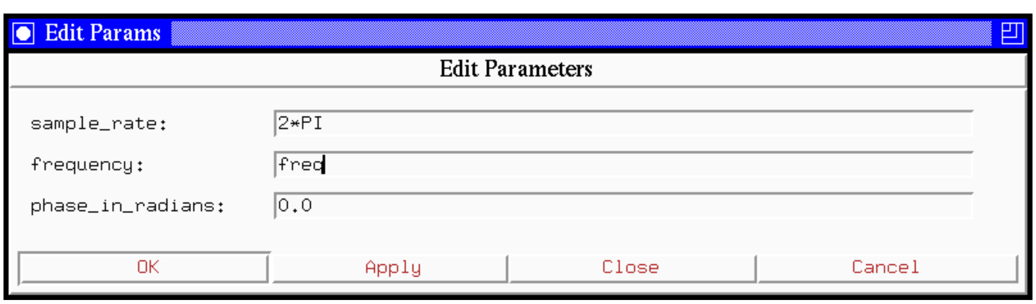
The *edit-params* command in **`pigi`** permits the user to set the initial value of a settable state of any star (lowest level block), and to define and set parameters for a galaxy (composite block) or universe (complete application).

Passing parameters through the hierarchy

Star parameters may be linked to the parameters of the galaxy or universe that contains the star. The syntax for linking the values of the star parameters to values of galaxy or universe parameters is simple. Consider again the **`sinMod`** application shown in figure 2-4. The parameter screen for the **`modulator`** block is shown below:



This block, however, is a galaxy, not a star. If you look inside (as has been done in figure 2-4), and edit the parameters of the **`singen`** block inside **`modulator`**, you will see



Notice now that the value of the *frequency* parameter is a symbolic expression, “freq”. This refers to the galaxy parameter “freq”. Thus, parameter values can be passed down through the hierarchy. These symbolic references can appear in expressions, which we discuss next.

Parameter expressions

Parameter values set through **`pigi`** can be arithmetic expressions. This is particularly useful for propagating values down from a universe parameter to star parameters somewhere down in the hierarchy. An example of a valid parameter expression is:

```
PI/(2*order)
```

where **`order`** is a parameter defined in the galaxy or universe. The basic arithmetic operators are addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (^). These operators work on integers and floating-point numbers. Currently all intermediate expressions are converted to the type of the parameter being computed. Hence, it is necessary to be very careful when, for example, using floating-point values to compute an integer parameter. All intermediate expressions will be converted to integers.

Complex-valued parameters

When defining complex values, the basic syntax is

```
(real, imag)
```

where **`real`** and **`imag`** evaluate to integers or floats.

Fixed-point parameters

Fixed-point parameters may be assigned a precision directly. To do this, the parameter is given in the syntax "(*value*, *precision*)", where *value* is ordinary number and *precision* is given by either of two syntaxes:

- **Syntax 1**: As a string like "3.2", or more generally "*m.n*", where *m* is the number of integer bits (to the left of the binary point) and *n* is the number of fractional bits (to the right of the binary point). Thus length is *m*+*n*.
- **Syntax 2**: A string like "24/32" which means 24 fraction bits from a total length of 32. This format is often more convenient because the word length often remains constant while the number of fraction bits changes with the normalization being used.

In both cases, the sign bit counts as one of the integer bits, so this number must be at least one.

Thus, for example, a fixed-point parameter might be defined as "(0.8, 2/4)." This means that a 4-bit word will be used with two fraction bits. Since the value "0.8" cannot be represented precisely in this precision, the actual value of the parameter will be rounded to "0.75".

A fixed-point parameter can also be given a value without a precision. In this case, the default precision is used. This has a total word length of 24 bits with the number of integer bits set as required to store the value. For example, the number 1.0 creates a fixed-point object with precision 2.22, and a value like 0.5 would create one with precision 1.23.

The precision of internal computations in a star is typically given by a parameter of type **`precision`**. A precision parameter has a value specified using either of the two syntaxes above.

2.4.3 Files and comments

The values of most parameter types can be read from a file. This syntax for this is to use the symbol “<” as in the following example:

```
< filename
```

The contents of the file **filename** are then read in and used as the parameter value. File inputs can be very useful for array type parameters which may require a large amount of data.

Comments are also supported for parameters. A comment is specified with the “#” symbol. Everything after the “#” until the end of the line is discarded when the parameter is evaluated. Comments are especially useful in combination with files as they can help remind the user of which galaxy or star parameter the file was written for.

For example, a comment could be added to the *frequency* parameter above:

```
freq # This is set to the Galaxy parameter
```

Note that as of this writing, files and comments are not supported for the String parameter type, but they are supported for the String Array parameters.

2.4.4 Array parameters

When defining arrays of integers, floats, complex numbers, fixed-point numbers, or strings, the basic syntax is a simple list separated by spaces. For example,

```
1 2 3 4 5
```

defines an integer array with five elements. The elements can be expressions if they are surrounded by parentheses:

```
1 2 PI (2*PI)
```

Repetition can be indicated using the following syntax:

```
value[n]
```

where **n** evaluates to an integer. An array or portion of an array can be input from a file, using the symbol “<” as in the following example:

```
1 2 < filename 3 4
```

Here, the first two elements of the array will be 1 and 2, the next elements will be read from file **filename,** and the last two elements will be 3 and 4. This latter capability can be used in combination with the **WaveForm** star to read a signal from a file.

2.4.5 String Parameters

There is a bit of complication when one wishes to set a string parameter or string array parameter equal to the value of a galaxy or universe parameter, because a distinction must be made between a sequence of characters that give the name of a symbol and a sequence of characters to be interpreted literally. The syntax to use is explained in the example:

```
This string has the word {word} taken from another parameter
```

Here, **{word}** represents the value of a string universe or galaxy parameter. This capability is especially useful for constructing labels for output plots.

String Arrays have a few more special restrictions. Each word (separated by white space) is a separate entry in the array. To include white space in an element of the array, use quotation marks. Thus, the following string array

```
first "the second element" third
```

has three elements in it. The string array

```
repeat[10]
```

has ten separate copies of the string “repeat” in 10 separate entries in the array. Curly braces are used to substitute in values from galaxy parameters. Thus, in

```
{ paramname }
```

“**paramname**” must be the name of either a string array or a scalar-valued parameter (an integer, float or complex array, for example, is not permitted). If it is a string array, then each element of **paramname** becomes an element of the parameter. If it is some other kind of parameter, the value becomes a single element of the string array.

To use one of [,], {, or } literally, quote them with double quotes. To turn off the special meaning of a double quote, precede it with a backslash: \”. Similarly, use \\ to get a single backslash

String array values may also be read from files using the “<” symbol. For details on how to use file reference, see section 2.4.3 above. Note that for string arrays, the filename must be a literal string such as

```
< /users/ptolemy/data/filename
```

and cannot make indirect reference to the file using other parameters. For example:

```
< /users/ptolemy/{data_dir}/data_file
```

would cause an error since the parameter *data_dir* would not be substituted for.

2.5 Particle types

The packets of data that pass from one star to another in Ptolemy are called *particles*. So far, all particles have simply been floating-point numbers representing samples of signals. However, several other data types are supported. Each star icon has a stem for each porthole. In **pigi**, if you are using a color monitor, the color of the stem indicates the type of data that the porthole consumes or produces, as summarized in table 2-5. A blue stem on an input or output of a star icon indicates type “float”, a purple stem indicates type “fix” for fix-point particles, a white stem indicates type “complex”, an orange stem indicates type “int” for integer particles, a green stem indicates “message”, and a red stem indicates “anytype”. The “message” type is a user-defined data type (see the programmer’s manual). A star that operates on “anytype” particles is said to be *polymorphic*. It processes all input data types exactly the same way. In addition, stars which input or output Matrix type particles have stems which are extra thick with colors corresponding to the four main types, float, int, complex, and fix.

Ptolemy usually makes conversions between these types automatically. The float to complex conversion does the obvious thing, putting the float value into the real part of the complex number, and setting the imaginary part to zero. The complex to float conversion computes the magnitude of the complex number.

The **Xscope** star, and some other stars that generate output, accept “anytype” of input. If the input is complex, the magnitude will be plotted, because it will be cast to a float internally. If you want to plot both the real and imaginary parts, you should use the **ComplexReal** conversion star first.

For technical reasons too involved to explain here, automatic type conversions cannot always be made. In particular, Ptolemy can only do one type conversion for any given output of a star. So if a single output is connected to multiple inputs of different types, Ptolemy may complain. For example, suppose you connect a "float" output to both a "complex" input and a "float" input. Ptolemy will complain. A simple solution is to insert an explicit type conversion. The type conversion stars can be found in the "conversion" palette of the appropriate domain. It will be explained below how to find this. Note that the matrix particle types do not support automatic type conversion at all. Conversion stars need to be explicitly inserted between two stars that work on different Matrix types.

Ptolemy's automatic type conversions have one potentially serious pitfall. Suppose you connect a "float" output to a "complex" and "anytype" input, as shown in figure 2-6. This is legal, because Ptolemy can satisfy the constraints with just one type conversion, float to complex. However, suppose the "anytype" input belongs to an **Xscope** star. As explained above, the **Xscope** star plots the magnitude of a complex number, unless you have an explicit **ComplexReal** conversion star. Hence, instead of plotting the "float" output, **Xscope** will plot the magnitude, converting all negative numbers to positive. A way to correct the problem is to explicitly convert the float to complex in the upper path only, as shown in figure 2-7.

2.6 The oct design database and its editor, vem

With the experience gained so far, it may be helpful to explain more clearly the software architecture of the system. **Pigi** is built on top of existing CAD tools that are part of the Berkeley CAD framework. An important component of this framework is **oct**, which serves as the design database. **Oct** keeps track of block connections, parameter values, hierarchy, and file structure, and hence moderates all accesses to designs stored on disk. The organization is shown in figure 2-8. **Vem** is an interactive graphical editor for **oct**. **Vem** provides one of many ways to examine and edit designs stored by **oct**. This chapter gives just enough information about **vem** to use it with Ptolemy in simple ways. More complete documentation is contained

Type name	Color	Description
ANYTYPE	red	any data type is accepted
FLOAT	blue	floating-point scalars
FLOAT_MATRIX_ENV	blue (thick)	floating-point matrices
COMPLEX	white	complex scalars
COMPLEX_MATRIX_ENV	white (thick)	complex matrix
INT	orange	integer scalar
INT_MATRIX_ENV	orange (thick)	integer matrix
FIX	violet	fixed-point scalar
FIX_MATRIX_ENV	violet (thick)	fixed-point matrices
MESSAGE	green	user-defined data type

TABLE 2-5: Data types supported by the Ptolemy kernel.

in chapter 20, “Vem — The Graphical Editor for Oct” on page 20-1.

In **pigi**, the Ptolemy kernel runs in a separate Unix process, called **pigiRpc**, attached to **vem**. Users edit designs using **vem**, store their designs using **oct**, and execute their

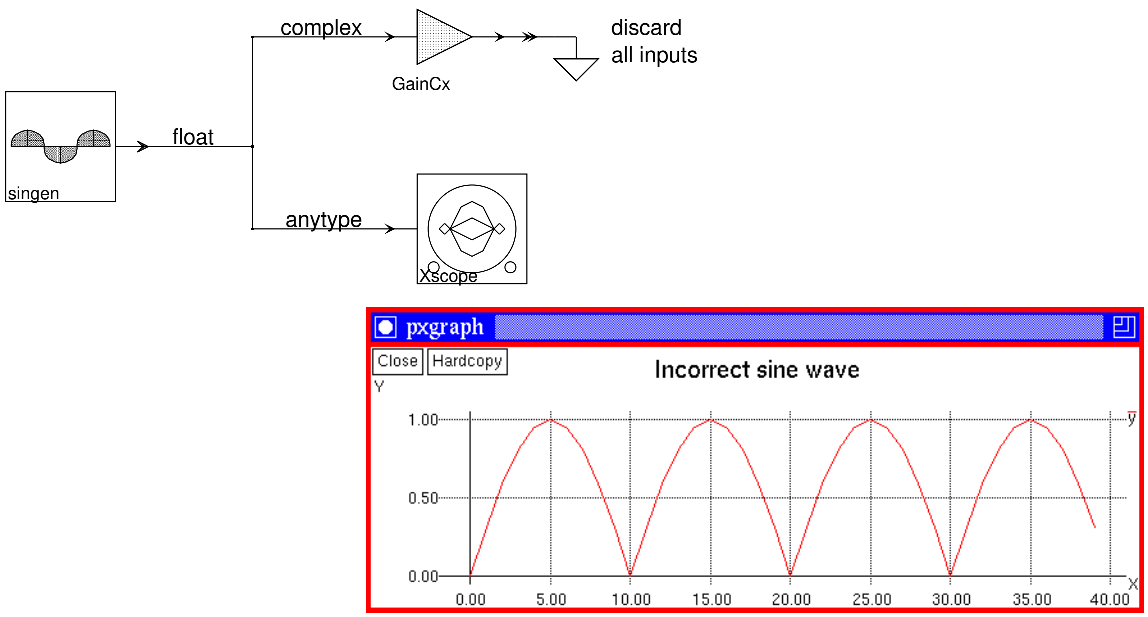


FIGURE 2-6: Automatic type conversion has one serious pitfall. Any given output can be converted only once, so the above float signal will be converted to complex. But the Xscope star, which accepts “anytype”, then converts this back to float by computing the magnitude. As shown in the plot, the result is unexpected. Explicit type conversions will avoid this problem

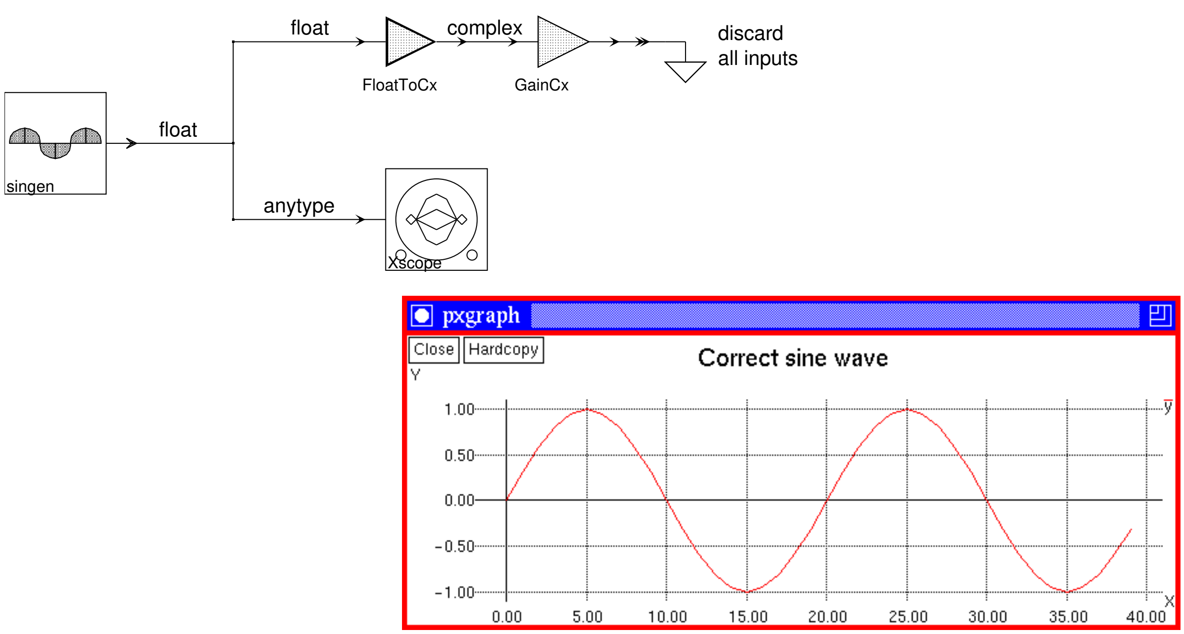


FIGURE 2-7: The correction to the problem illustrated in figure 2-6 is to use explicit type conversion.

application through the link to the Ptolemy kernel. The two Unix processes are shown in the shaded boxes in figure 2-8. The user interacts with both processes, and only the user interface of the **pigiRpc** process has been upgraded to use Tcl/Tk, as explained above. With this software architecture in mind, we can now define terms that we have been using informally.

Oct objects (which are stored on disk) are called facets. A *facet* is the fundamental unit that a user edits with **vem**. As an analogy, we can think of a facet as a text file in a computer system and **vem** as a text editor, such as *vi* or *emacs*. However, instead of calling system routines to access the data stored in a text file like *vi* does, **vem** calls **oct** routines to access the data stored in a facet. Thus, **oct** manages all data accesses to facets. Facets may define a universe or a galaxy, for example. Thus, for example, figure 2-4 on page 2-8 shows a facet that defines a universe and a second one that defines a galaxy.

Facets may also define the physical appearance and formal terminals of icons that represent stars, galaxies, universes, and wormholes. Thus, for example, the physical appearance of each icon in figure 2-4 is defined in another facet called the *interface facet*. A schematic that uses icons, by contrast, is called a *contents facet.* The "edit-icon" command ("I") will open the facet defining an icon. Instructions for modifying the appearance of an icon are given in "Editing Icons" on page 2-33.

A facet may also contain a *palette*, which is simply a collection of disconnected icons. Palettes are directories of stars, galaxies, and universes in a library. Thus, for example, figure 2-3 on page 2-6 shows two palettes, both of which contain sets of icons. Note that facet names, like file names in Unix, should not contain spaces.

2.7 Creating universes

If you are following this chapter sequentially, then you still have Ptolemy running from previous sections. To see how Ptolemy will behave when started in your own directory, exit **pigi**. Do this by typing a control-d character in the **vem** console window. A dialog box may appear with a menu of facets that **vem** thinks have been changed. Since all of these

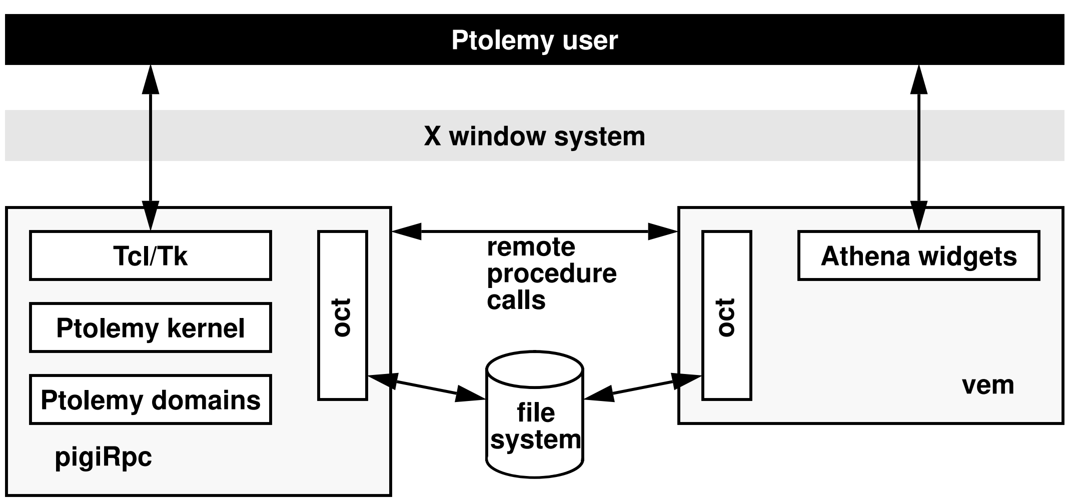


FIGURE 2-8: The software architecture of the Ptolemy design environment running under pigi, the graphical interface. The user interacts with two Unix processes, pigiRpc and vem.

belong to the user "ptolemy", you do not want to save them. If it appears, do not select any of them. Just click "OK". A a warning window may then appear telling you that closing the console window will terminate the program. Just click "Yes".

In this section, we will show how to create your own universes with a simple example that is very similar to the **`sinMod`** demo explored above. First, be sure you are in a directory where you have write permission, like your home directory.

- Create a new work area:

```
mkdir example
cd example
```

- Start **`pigi`**:

```
pigi
```

You will see the message:

```
creating initial facet "init.pal"
```

Wait until the welcome window with the picture of Ptolemy appears. We are now ready to learn about the basics of using **`vem`**.

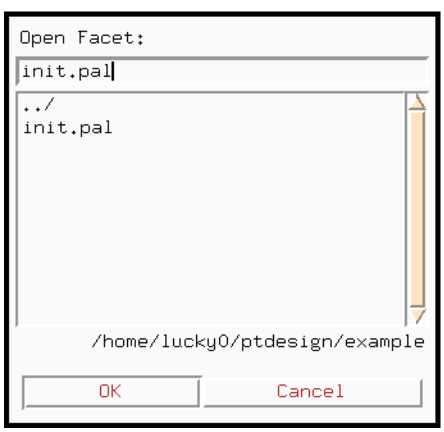
2.7.1 Opening working windows

Now we are ready to create a simple universe. Let's create a simulation that generates a sine wave and displays it.

- Open a new facet: The facet that is already open, called "**`init.pal`**", is special because **`pigi`** always opens a facet by this name in the directory in which it starts. Convention in Ptolemy dictates that "**`init.pal`**" should be used to store icons representing complete applications, so instead of using this facet, we will create a new one.

    - Place the cursor in window labeled "init.pal:schematic".

    - Select the *open-facet* command from the "Window" **`pigi`** menu (shift-middle-button). Alternatively, type an "F". You will get a directory browser the looks like this:



    - Replace the name "init.pal" in the text widget with "wave" and click the "OK" button (or hit the return key). A quick way to delete the "init.pal" is using control-u. A new blank window will appear.

- Open a palette:
  - Place the cursor in either blank window.
  - Select the *open-palette* command from the "Window" **pigi** menu. Alternatively, type an "O".
  - **Pigi** will present a palette menu. Select the "sdf" palette by clicking the left button in the box next to "$PTOLEMY/src/domains/sdf/icons/main.pal" (the first entry), and then click on "OK".
  - The palette that opens is shown in figure 2-9. This palette shows the basic categories of synchronous dataflow stars that are available. There are too many stars to put in just one palette. You can use the Window:look-inside ("i") command to open any of the palettes. At this point you should look inside the "Signal Sources", "Nonlinear Functions", and "Signal Sinks". Arrange these palettes on the screen so that you can see the blank window labeled "wave". The stars and palettes are summarized in "An overview of SDF stars" on page 5-4.

2.7.2 Some basic vem commands

At this time, it is worth exploring some basic **vem** commands for manipulating window displays. **Vem** uses post-fix commands. This means that the user enters the arguments to a command before the command name itself. Arguments appear in the **vem** console window as the user enters them. Note that although the text of what the user enters is displayed in the console window, the cursor should be in one of the facet windows.

There are several types of arguments. Each argument type is entered in a different way. All graphics arguments are created with the left mouse button. The five types of arguments are listed below:

Point: Position the cursor, click the left mouse button.

Box: Position the cursor, drag[1] the left mouse button.

Line: Make a point, position the cursor on the point, and drag the left mouse button.

Object: Use *select-objects* and *unselect-objects* commands (explained later).

Text: Enclose text in double quotes.

Arguments can be removed from the command line by typing the delete key, backspace key, or "control-u", which deletes all the current arguments. There are three ways to enter com-

1. "Drag" means to press down on a mouse button, move the mouse while holding it down, and then release the button.

mands:

Menus: Click the middle-button for **`vem`** commands, shift-middle-button for **`pigi`** commands. Menus are of the “walking” variety, as explained before.

Key bindings: Commands can be bound to single keys and activated by just pressing the key. Key bindings are also called “single-key

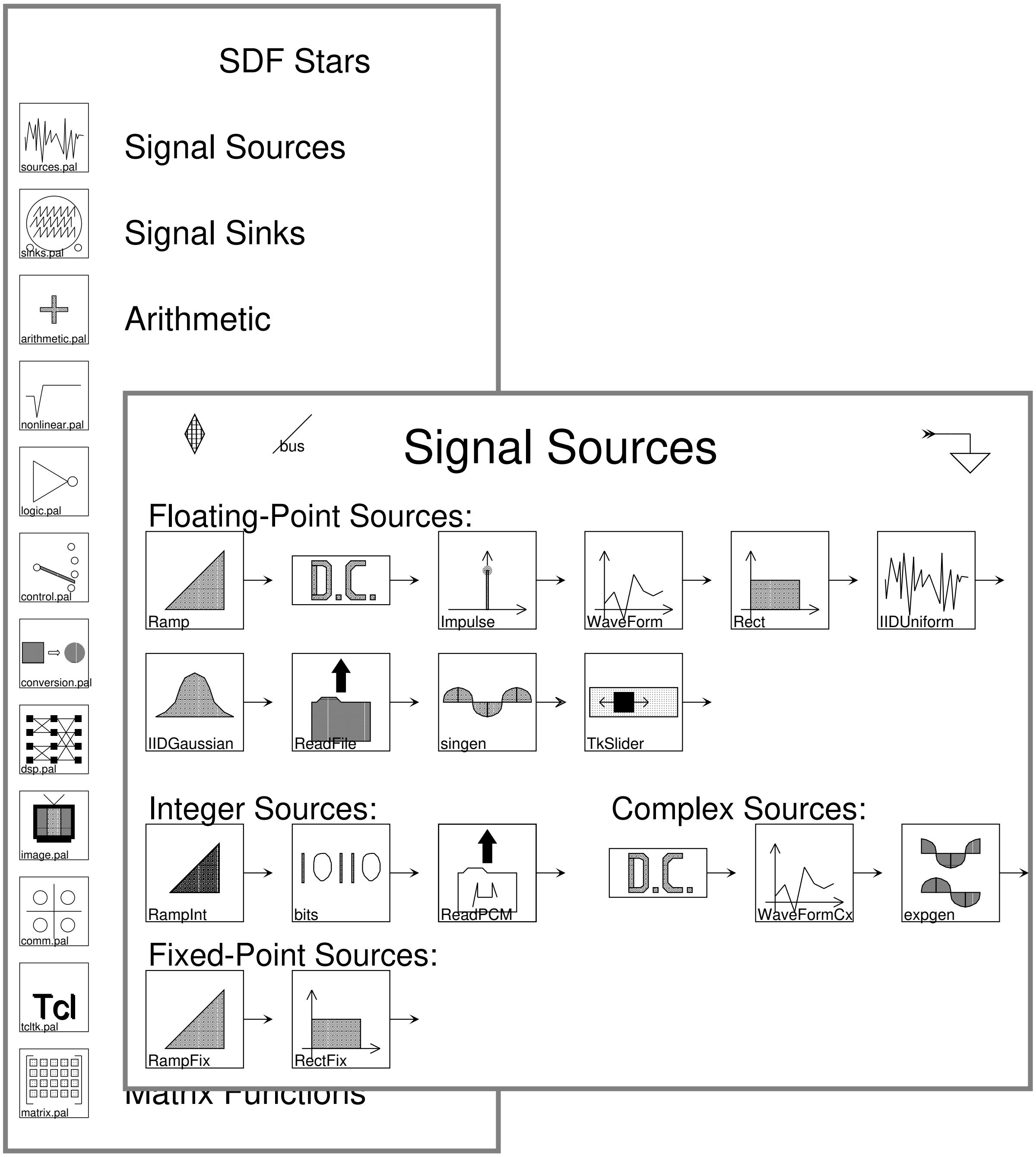


FIGURE 2-9: The master palette for the stars in the SDF domain (left) and one of the sub-palettes (right). The subpalette shows “sources” (signal generators). The palettes are explained in more detail in “An overview of SDF stars” on page 5-4.

accelerators", and are case sensitive. The key bindings are summarized in table 2-2 on page 7 and table 2-3 on page 11.

Type-in: Type a colon followed by the command name. This is rarely used by Ptolemy users, but **`vem`** experts use it occasionally.

Let's try a few examples, some of which should be familiar by now. Place the cursor in one of the palette windows (containing library stars) and:

- Type "shift-Z" (capital Z) for *zoom-out*. This makes everything smaller.
- Type "z" (lower-case z) for *zoom-in*. This makes everything bigger. If you zoom in sufficiently, labels will appear below each icon giving the name of the star. **`Vem`** does not display these labels if they would be too small.
- Try "p" for *pan*. Pan moves the spot under the cursor to the center of the window.
- The **`vem`** *pan* command can also take as an argument a point which will indicate the new center of the window. Recall that the argument must be entered first. Place a point somewhere in the palette window by clicking the left button, and type "p". The location of your point became the center of the window.
- The **`vem`** *open-window* command can take a box as an argument. Draw a box in the palette window by dragging the left mouse button and then type "o", or find the *open-window* command in the **`vem`** menu.
- Try placing points in the new window. Notice that they also appear in the original palette window. Also notice that you are only permitted to place points at certain locations. **`Vem`** has an implicit *grid* to which points *snap*. The default snap resolution is suitable for making Ptolemy universes.
- You can get rid of your point (or any argument list) by typing "control-u". You can delete arguments one-at-a-time by typing "control-h". Try placing several points and then deleting them one by one.
- You can close the new window (or any **`vem`** window) with "control-d".
- A particularly useful command at this time is *show-all*, or "f". This rescales and recenters the display so that everything in the facet is visible. Try this command in the palette window that you have been working with.
- You can also resize a window, using whatever X Window bindings you have installed, and then type "f" to rescale the display to fill the window.

2.7.3 Building an example

- Create an instance of the star called "**`Ramp`**". This star is at the upper right of the sources palette. Its icon has an orange triangle. To do this:
  - Put the cursor in the window "wave:schematic".
  - Create a point anywhere in the window by clicking the left button.

- Move the cursor over the "**`Ramp`**" icon in the palette and press the "c" key. This is a key binding that executes the **`vem`** "create" command.

- You have just created an **instance** of the "**`Ramp`**" icon. The actual data that describes how the "**`Ramp`**" icon should be drawn is stored in another facet (an "interface facet"). An **instance** of the "**`Ramp`**" icon points to this facet.

• Delete and select instances: Sometimes in the process of editing your schematic, you may need to delete objects. As an example, let's create another **`Ramp`** instance and then delete it.

- Create another **`Ramp`** instance next to the first one: place a point near the original **`Ramp`**, place the cursor over the **`Ramp`** icon in the palette and press "c". Actually, you don't have to use the icon in the palette — you could also put the cursor over the already existing **`Ramp`** icon to achieve the same effect.

- Place the cursor over the new **`Ramp`** icon and execute *select-objects* by typing "s". This creates an object argument on the **`vem`** command line. This is necessary because the **`vem`** *delete-objects* command takes arguments of type "object". The *select-objects* command takes point, box, and/or line arguments and turns the items underneath them into object arguments. The *unselect-objects* command ("u") does the reverse of *select-objects*.

- Execute *delete-objects* by typing "D" (upper-case!). This deletes the objects we selected previously.

- You could also have deleted the newly created Ramp with the *undo* command ("U"). This is an infinite undo, so you can backtrack through all changes you have made since starting the **`vem`** session by repeatedly executing the undo command.

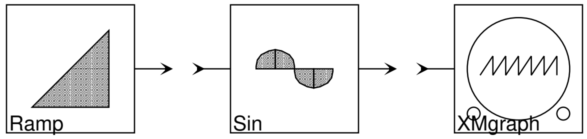
- Occasionally when you use the select and unselect commands, the objects are not redrawn correctly. When this happens, use the **`vem`** *redraw-window* command, "control-l" (lower case L), to redraw.

• Create the remaining instances in our example:

- Create an instance of the "**`Sin`**" icon to the right of the Ramp. "**`Sin`**" is in the "nonlinear" palette, where icons are arranged alphabetically by name. Make sure it does not overlap with the Ramp icon. If it overlaps, you can delete it and create a new one.

- Create an "**`XMgraph`**" instance to the right of the **`Sin`** icon. "**`XMgraph`**" is the first icon in the first row of the "sinks" palette.

We now have three icons: a **`Ramp`**, a **`Sin`**, and an **`XMgraph`**. Your facet should look something like this:



Next, we will connect them together.

- Connect the **`Ramp`** output to the **`Sin`** input using the following steps:

  - With the mouse cursor in the "wave" window, type "f" to show all. This will rescale your system, and make it easier to make connections.

  - Draw a line between the output of the **`Ramp`** and the input of the **`Sin`**: put the cursor over the **`Ramp`** output, double-click on the left mouse button, drag the cursor to the **`Sin`** input, and then let up on the mouse button. If the two terminals are not on a horizontal line, you can bend the line by momentarily releasing the mouse button while dragging it.

  - Type "c" (for *create*) to create a wire. Notice that the *create* command creates wires or instances depending on the type of arguments it is called with.

  - If you need to delete a wire, you can draw a box around it (click and drag with the mouse), select it (press "s"), and then delete it ("D").

- Connect the **`Sin`** output to the **`XMgraph`** input in a similar way.
- Run the universe: We now have a complete universe that we can simulate.

  - Execute the *run* command from the **`pigi`** "Exec" menu, or type an "R".

  - Enter "100" for "When to stop". Do this by typing "control-u" to remove the default entry in the text widget and typing 100. This specifies that the system should be run through 100 "iterations". What constitutes an iteration is explained in chapter 5, "SDF Domain" on page 5-1. For this simple system, it is just the number of samples processed.

  - Clicking on the GO button or typing a return character will run the system.

A new window with a graph of a rough sine wave should appear. The system generates the sine wave by taking the sine of a sequence of increasing numbers generated by the **`Ramp`** star. The execution of the **`XMgraph`** star created this new window to show the output of our simulation. To remove this window, click on the "Close" button or press "control-d" in it.

- Save the facet by typing “S” (upper-case) with the cursor in the “wave” window. This executes the **`vem`** *save-window* command. It is wise to periodically save your work in case the editor or computer system fails unexpectedly.
- Change parameters: If we look at the output, the sine wave appears jagged. This is because the **`Ramp`** star has a set of default parameters which cause it to generate output values with an increment that is too large. We can change the parameters of as follows:

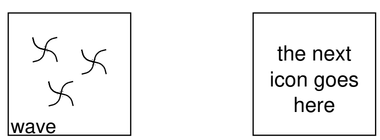
    - Place the cursor over the **`Ramp`** icon and execute *edit-params* in the **`pigi`** menu (or type “e”). A dialog box will appear that shows the current parameters.

    - Replace the value of *step* with “PI/50”. (You can use “control-u” to erase the old value.) Finally, click the “OK” button to store the new parameters. This is an example of Ptolemy’s parameter expression syntax, explained above.

- Run the simulation again using 100 iterations. This time the output should look like one cycle of a reasonably smooth sine wave.
- Use *save-window* again to save the new parameters.

To be able to conveniently access this example again, you should create an icon for it. We will do this with the **`pigi`** command “Extend:make-schem-icon”, or “@”.

- Place the mouse cursor in the “wave” facet window, and hit the “@” key. A dialog box appears asking for the name of the palette in which you would like to put the icon. By convention, we put universe icons in palettes called “init.pal”. So replace the default entry (which should be “./user.pal”) with “init.pal”. When the icon is made, find the “init.pal” window that first opened when you started the system, and type “f” to show all. It should look like this:



  Looking inside this icon (“i”) will get you your “wave” facet. The second item in the palette is a marker indicating where the next icon that you create will go. Henceforth, anytime you start **`pigi`** in this same directory, the first window you will see will be this “init.pal” window.

- Our example is now complete. To exit:

    - Close all pxgraph windows with “control-d”.

    - Type “control-d” in the **`vem`** console window. If nothing needed to be saved, the program exits immediately. Otherwise, a dialog box appears asking you to choose buffers to be saved. Unfortunately, as of this writing, some of the buffers listed may have already been saved and do not need to be saved again. The program is overly cautious. To indicate which of the listed buffers you wish to save, click on the box to the left of each name. Then click on the “OK” button.

- A final warning appears telling you that closing the console window will terminate the program. Click on “Yes”.

2.8 Using galaxies

In this section we will explain how to create galaxies. Galaxies allow you to use hierarchy to partition your design into more manageable pieces and to re-use designs as components in other designs.

2.8.1 Creating a galaxy

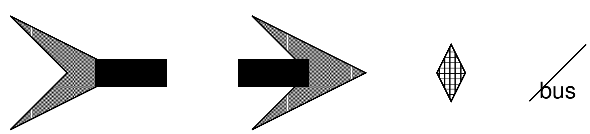
Use the schematic we created in the last example to make a sine wave generator galaxy.

- Instead of modifying our previous example, we will make a copy of it. In your “example” directory, type:

```
cp -r wave singen
```

The recursive copy, “cp -r”, is necessary because **oct** stores data using a hierarchical directory structure. Of course, if the facet **singen** exists already, you must remove it with “**rm -r**” first before copying.

- Start **pigi**.
- Use *open-palette* (or “O”) to open the “$PTOLEMY/lib/colors/ptolemy/system” palette (the last one in the list of palettes). The system palette contains input and output ports which can be instantiated into schematics just like stars. The contents of the palette are shown below:



- Use *open-facet* (or “F”) to open the “singen” you created using “**cp -r**”. You can use the file browser shown on page 2-22; just double click on the name “singen” in the lower window of the browser.
- In the “singen” window, delete the **XMgraph** star and the wire attached to it. The easiest way to do this is draw a box (click-drag) around the star and its input wire, press “s” to select these objects, and then press “D” to delete them. (You may want to enlarge your window to make it easier to work.)
- Place an output port where the **XMgraph** star used to be and connect it to the output of the **Sin** star. The output port is the icon in the system palette with an arrowhead (an input port, by contrast, has a fish tail), as shown above.
- Name the output port “out”:

- Position the cursor over the black box on the new output port.

- Type “out” as a text argument, including quotation marks.

- Type “c” for *create*. Again, note that the *create* command has a different action than before. It names input or output terminals when given a text argument.

We now have a galaxy. The fact that a schematic has input or output ports distinguishes it as a galaxy. This galaxy that you just created is similar to the “singen” galaxy in the “Signal Sources” palette. Find it, and look inside, to make the comparison.

2.8.2 Using a galaxy

We have just created a galaxy that we would like to use in another design. In order to do this, we need to create an icon for this galaxy that we will then instantiate in our other design.

- Create an icon:

  - Place cursor in “singen:schematic” window.

  - Execute *make-schem-icon* in the **`pigi`** “Extend” menu (“@”).

  - The dialog box should contain:
    `Palette: ./user.pal`
    This specifies the name of the palette that will contain your icon. By convention, we usually put galaxy icons in the palette called “user.pal” in the current directory. Hence, this is the default name.

  - Since you had already created an icon for the “wave” universe, and that icon was copied by your “cp -r”, **`vem`** asks whether it is OK to overwrite the icon. Click “OK”. Wait until *make-schem-icon* is done. **`Vem`** informs you that it is done with a message in the **`vem`** console window, which may be buried by now.

- Open the palette called “**`user.pal`**” using *open-palette* (“O”). The newly created galaxy icon should appear in this palette along with the same a special icon we saw before, called a **cursor**. A **cursor** distinguishes a palette from other types of facets and it determines where the next icon will be placed.
- At this point, we have an icon for our sine wave generator galaxy. It is this icon facet that is instantiated in the “**`user.pal`**” palette. We can now use our sine generator galaxy simply by instantiating our icon into another schematic.
- Use *open-facet* to create a new facet with the name “**`modulation`**”.
- In the “modulation” window, create a universe that takes two “singen” galaxies, multiplies their outputs together, and then plots the result using “**`XMgraph`**”. The “**`XMgraph`**” star can be found again in the “Signal Sinks” subpalette of the SDF palette. The multiplier star, called **`Mpy`**, is in the “Arithmetic” subpalette.

Hint: If you place the icons so that their terminals fall on top of one another, then a connection gets made without having to draw a wire.

- Now run the universe with “when to stop” set to 100. The output should appear as a squared sine wave, which is just a sine wave of twice the frequency shifted up by 1/2.

So far, we have created a galaxy and used it in another universe. But, we could also have used our galaxy within another galaxy. In this way, large systems can be broken up into smaller more manageable pieces.

2.8.3 Galaxy and universe parameters

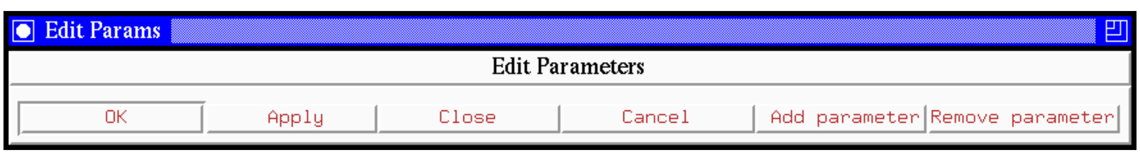
One of the problems with the “singen” galaxy that we just created is that it generates sine waves with a fixed frequency. We would like to make the frequency of the generator parameterizable. That way, we could set the two “singen” galaxies in our “modulation” universe to two different frequencies.

To make a galaxy parameterizable, we create **formal parameters** in the galaxy and then link the formal parameters to the **actual parameters** of the instances contained in the galaxy. The terms, “formal” and “actual” parameters, are analogous to formal and actual parameters in any procedural programming language. An example will make this clear.

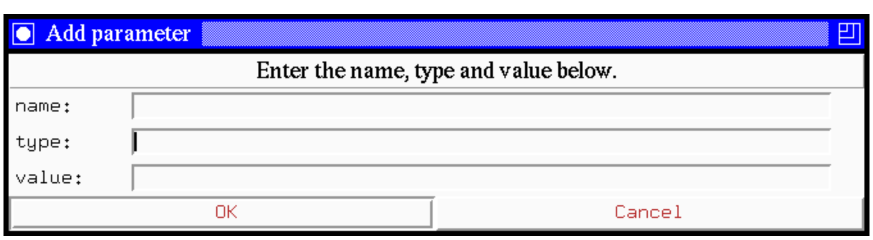
- Create formal parameters:

  - Place the cursor in the “singen” window but away from any instance, i.e., in the grey background of the facet.

  - Execute *edit-params* (“e”). An empty parameter window will appear, looking like this:



To add parameters to the galaxy, click on the “Add parameter” button. A window appears looking like this:



Fill in the dialog as follows:

name: freq
type: float
value: PI/50

The value will be the *default* value. Then click on "OK". Recall that you can use "tab" to move from one field to the next of the dialog box and "Return" instead of "OK". Hence, the dialog can be managed from the keyboard, without requiring the mouse.

We just created a new formal parameter called "freq" with a default value of "PI/50". Additional parameters may be added or old ones changed. The default value of a formal parameter can always be changed by executing *edit-params* in the background of the galaxy. Executing edit-params on the icon representing the galaxy changes the parameter values only for the instance represented by the icon. It overrides the default value specified in the background of the galaxy definition. The possible types for parameters are listed in table 2-6. The syntax for specifying values for parameters is described above in "Changing or setting parameters" on page 2-15. Exactly the same procedure can be used to attach formal parameters to a universe. This allows you to parameterize a complete Ptolemy application.

- Link formal parameters to actual parameters:
  - Place the cursor over the **Ramp** icon in the **singen** window.
  - Execute *edit-params* and fill in the dialog as follows:
    step: freq
    value: 0.0
    This allows the *freq* parameter of a singen instance to control the increment of the internal ramp star, thus controlling the output frequency.
- Change the frequency of one of the **singen** instances to "PI/5" by using *edit-params*. This **singen** will be ten times the frequency of the other.
- Run the universe with an iteration of 100. The output should show the product of two sine waves with different frequencies. Don't forget to save the facets we just created. It

Type name	Description	Example
float	floating-point number	0.2/PI
int	integer	10
complex	pair specified as (real-part, imag-part)	(1.0, 2.0)
string	string	this is a string
floatarray	array of floating-point numbers	0.0 [10] 1.0 0.0 [10]
intarray	array of integers	1 2 3 4 5 6 7
complexarray	array of complex numbers	(0.1, 0.2) (0.3, 0.4) (0.5, 0.6)
stringarray	array of strings	this string array "has five" elements

TABLE 2-6: Parameter types supported in Ptolemy

would also be a good idea to create an icon for **modulation** and put it in **init.pal**.

2.9 Editing Icons

Pigi automatically generates icons for stars and galaxies, respectively, when you invoke the *make-star* or *make-schem-icon* command from the Extend menu. **Pigi** puts the new icon in a user-specified palette, which by default is user.pal in the directory in which you started **pigi**. More or less any **vem** manipulations can be performed on this icon, but some guidelines should be followed. These icons have a generic symbol, shown in figure 2-2 on page 5. To change it, place the cursor over the icon and execute the *edit-icon* ("I") command in the **pigi** menu.[1] A new window containing the icon facet will appear.

Recall from section 2.6 on page 19 that icons are stored in *interface facets* and that the icons that appear in *contents facets* are really instances of icons. These instances merely refer to the actual icon facet. The *edit-icon* command opens a window into the actual icon. Any changes made in this window will affect the appearance of all instances referring to the icon.

Recall also that icon facets store a different kind of data from other facets. Icon facets contain information that tells **vem** how to draw objects. Hence, a different set of commands must be used to edit icons. Whenever you edit an icon, **vem** switches to a different mode called "physical editing style." In this mode, we create objects such as lines, boxes, and polygons. This is in contrast to "schematic editing style" which we used before to create instances and connect them together with wires. Physical editing style shares many commands with schematic editing style. For example, *select-objects* is active in both modes. A list of useful physical editing style commands and their key bindings is given in table 2-7.

The commands that create geometry expect a layer argument. The layer of an object determines its color and its fill pattern. To specify a layer, place the cursor over an object attached to the desired layer before executing a command. You can open a palette of layers with the *palette* ("P") command. The palette is shown in figure 2-10.

The layer palette contains several columns of solid and outline colors, with the name of the color at the top of the column. Colors at the top of each icon will be layered on top of colors below them in the columns. A set of special layers are arranged at the bottom of the palette. The layers for icon stems are explained below. The layers for icon bodies define the icon background and optional icon shadow.

A few simple notes will greatly help. First note that when the icon window is opened, the snap is automatically set to 5 "**oct** units". This is because the default snap for schematic windows, normally 25 units, is far too coarse for most icon editing functions. A reasonable compromise is 5 units, unless you are going to try to create a very elaborate icon, in which case 1 unit is probably what you want. The **vem** Options:window-options command allows you to change the snap.

When editing an icon, the **vem** menu is slightly different than when you are editing a schematic. In **vem** terminology, this is because you are working with the *physical view* of a facet. The commands are shown in table 2-7. Most icons can be created by experimenting with the following operations:

1. You must have write permission on the facet to change the icon.

- Select the default symbol within the icon that Ptolemy created when it created the icon (a star, galaxy, cluster of galaxies, or palette symbol, as shown in figure 2-2 on page 5). You can do this by drawing a box (drag the left mouse button) and typing “s” (or using

Menu	Heading	Command	Key	Description
vem	none	no command name	cntr-h	remove the last argument (point, box, etc.)
			del	remove the last argument (point, box, etc.)
			cntr-u	remove all arguments from the argument list
			cntr-l	(control lower case L) redraw the window
	System	open-window	o	open a new view into a facet
		close-window	cntr-d	close a window
		where	?	find the position of the cursor in oct units
		palette	P	open the color palette for editing icons
		save-window	S	save a facet
		bindings	b	display key bindings (single key accelerators)
		re-read		restore a facet to the last saved version
	Display	pan	p	move the view to be centered at a given spot
		zoom-in	z	zoom in for a closer view of a facet
		zoom-out	Z	zoom out
		show-all	f	rescale the schematic to fit the window
		same-scale	=	used to get two windows to use the same scale
	Options	window-options		adjust snap, grid spacing, etc.
		layer-display		selectively display colors
		toggle-grid	g	turn on or off the grid display
	Undo	undo	U	undo any number of previous changes
	Edit	create-geometry	c	create a line, box, circle, etc.
		alter-geometry	a	replace an object with one on the argument list
		change-layer	l	change the color of an object
		set-path-width	w	change the width of lines
		create-circle	C	draw a filled or empty circle
		edit-label	E	specify or modify a label
		delete-physical	D	remove the specified object
	Selection	select-objects	s	add an object to the argument list for a command
		unselect-objects	u	remove an object from the argument list
		move-physical	m	move an object
		copy-physical	x	copy one or more objects in a schematic
		transform	t	rotate or reflect an object
		select-terms	cntr-t	select terminals
		delete-physical	D	delete objects
	Application	rpc-any	r	start a vem application (pigiRpc is one)

TABLE 2-7: A summary of the Ptolemy commands in the vem menu in icon editing mode. These commands are obtained by clicking the middle mouse button without holding the shift button when your mouse cursor is in an icon window. The single-key accelerators for commands that have them are shown. More complete documentation can be found in chapter 20, “Vem — The Graphical Editor for Oct” on page 20-1. The command that differ significantly from those in table 2-3 are shaded.

Selection:select-objects in the menu). You can unselect with “u” or control-u. An alternative selection method is to place a point and type “s”. This usually provokes a dialog box to resolve ambiguities. Delete whatever parts of the icon you don’t want using “D” or Selection:delete-objects. WARNING: Do not delete terminals! If you accidentally delete a terminal, the easiest action is begin again from scratch, asking **`pigi`** to create a new icon.

- Bring up the **`vem`** color palette by typing “P” (or System:palette in the menu). You will get the window shown in figure 2-10.
- Draw a line by clicking the left button to place a point, and pushing and dragging the mouse button from the same point. Then move the mouse to desired color in the color palette and type “c” (or Edit:create-geometry from the menu). A line may consist of multiple line segments by just repeatedly pushing and dragging the mouse button.
- To create filled polygons, place points at the vertices, then type “c” on the appropriate solid color in the palette.
- To create a circle, place a point at the center, a point on periphery, and type “C” on the appropriate color. To create a filled circle, use a line segment instead of a pair of points.
- Objects can be moved by selecting them, dragging the mouse (using the right button) to produce an image of the object in the desired place, and typing “m”.
- You may change (or delete) the label that **`pigi`** automatically puts at the bottom of the icon. To change it, select it and type “E” (Edit:edit-label in the menu). The resulting dialog box is self explanatory. The standard Emacs-like editing commands apply.

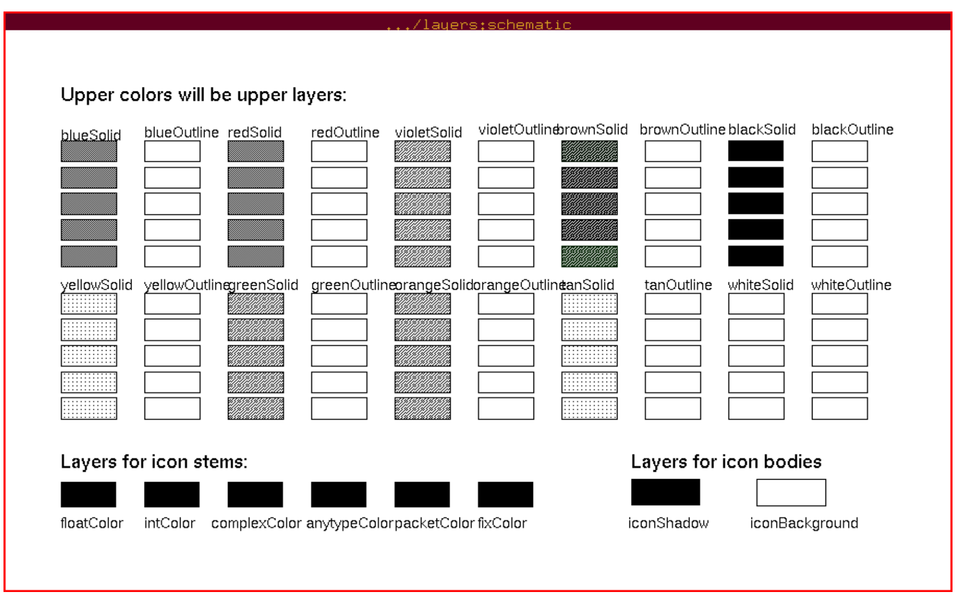


FIGURE 2-10: The palette of colors and layers that can be used to create icons. This palette is invoked by the “palette” vem command (“P”). Each color has a column of boxes. The higher the box you use, the closer to the front an object will be. The colors at the bottom are special, in that they are associated with particular data types.

- BE SURE TO SAVE YOUR ICON. This can be done by typing “S” (System:save-window in the **vem** menu). You can close your window with control-d. Note that **vem** buffers the data in the window. You can close it and reopen it without saving it, as long as the session has not been interrupted.

By convention, the data types supported by a terminal are indicated by the color of the stem that connects the terminal to the body of the icon. The following colors are currently in use:

ANYTYPE:	red
FLOAT:	blue
INT:	brown
FIX:	purple
COMPLEX:	white
PACKET:	green

The color is currently set automatically by the icon generator by using layers defined specifically for this purpose, called **anytypeColor**, **floatColor**, **intColor**, **fixColor**, **complexColor**, and **packetColor**. These colors are shown at the bottom of the color palette in figure 2-10.

You can change the color of an object manually, if you wish. To do this, select the object, type “"xxxx"”, where xxxx is replaced by the color name (the quotation marks are necessary), and then type the single character “l” (an el — or Edit:change-layer in the menu). Be sure not to change the color of a terminal! Again, be sure to save the window.

One final editing operation is a little trickier: moving terminals. **Pigi** places terminals rather arbitrarily, since it knows nothing of their function. You may wish to have a smaller icon than the default, in which case you have to move the input terminals closer to the output terminals. Or may wish to change the order of the terminals, or you may want to have terminals on the top or bottom of the icon rather than right or left. All of these can be done, but the following cautions must be observed:

- Do not move terminals of icons that have already been used in applications. Unfortunately, if you do this, the block will become disconnected in all applications where it is used. If you are tempted to move terminals in a commonly used block, consider the tedium of finding all applications (belonging to all users) and reconnecting the block. ONLY MOVE TERMINALS ON BRAND NEW ICONS.
- You must respect the default snap of 25 for schematic windows, and move terminals to a point that falls on a multiple of 25 units. Otherwise, connecting to the terminal will be very difficult. Normally, when the icon window opens, the grid lines are 10 units apart. So you can place terminals any multiple of 2.5 grid lines away from the center.
- To orient terminals so they are aiming up or down, select the terminal, type “t” (or Selection:transform in the menu), then type “m” to move it. Repeatedly typing “t” will continue to transform (rotate) the terminal.

A little more detail on the **oct** internals may be useful if you explore the files that are created by these operations. For make-schem-icon, if the schematic is called *xxx*, then the icon itself is stored in “*xxx*/schematic/interface;”. The semicolon is part of the filename (this creates some interesting challenges when manipulating this file in Unix, since the Unix shell interprets the semicolon as a command delimiter). The standard stars that are normally part of the Ptolemy distribution are stored in "**$PTOLEMY/src/domains/*dom***", where ***dom*** is the domain name such as "sdf" or "de". The icons for the stars are stored in a subdirectory called **icons**, the icons for demo systems in a subdirectory called **demo**, and the source code for the stars are stored in a subdirectory called **stars**. Feel free to explore these directories.

Changing the number of terminals in galaxy icons

Whenever the contents of a galaxy are changed so that the new definition has different I/O ports, the icon must be updated as well. You can do this by calling *make-schem-icon* again to replace the old icon with a new one. **Vem** will not allow you to overwrite the old icon if you have instances of the old icon in any open window (regardless of whether the window is iconified). Hence, you must either close those windows with "control-d" or delete the offending icon before replacing it with a new icon. Note that changing number of terminals will also change their layout, so that connections in existing schematics may no longer be valid.

2.10 Sounds

On some workstations (currently only SGI Indigos, HP 700s and HP 800s and Sun SPARCstations, on the hppa, you may need more publically available software), Ptolemy can play sounds over the workstation speaker. You can try this with the universe you just created. Replace the **XMgraph** star in **waveform** window with an instance of the **Play** star (second row of the sinks palette, right of center, with a stylized loudspeaker as an icon). Edit the parameters of the **Play** star entering 16000 for the *gain* parameter. (To see details about the **Play** star, execute the "profile" command in the "Other" menu, or type a comma (",") with the mouse on the **Play** icon). The SPARCstations speaker is driven by a codec that operates at an 8 kHz sample rate. So running this universe for 40000 samples will produce about 5 seconds of sound. The sound produced by the current parameters is not particularly attractive. Experiment with different parameter values. Try PI/1000 in place of the PI/50.

An interesting variant of this system modulates a chirp instead of a pure sinusoid with a low frequency sinusoid. A chirp is a sinusoid that sweeps over a frequency range. You could replace one of your **sinegen** instances with something that generates a chirp, and again experiment with parameters.

A chirp can be created with three stars: a **Ramp**, an **Integrator** and a **Sin**, connected in series. The *step* parameter of the **Ramp** should be very small, such as 0.0001. With this value, you will hear some aliasing if you create five seconds of sound. The **Integrator** is in the "arithmetic" palette, furthest on the right, and its default parameter values are fine for this purpose. Use the "profile" command (",") to read about it. Note that a fourth star, a **Const** (second star in "sources" palette) is needed to set the **Integrator** *reset* input to zero.

In the SDF domain, sound output is collected into a file, and then played out in real time. Another alternative, available in the CGC domain, is to generate the output in real-time. Since the CGC stars have not been optimized for real-time performance, only simple signals can be generated at this time.

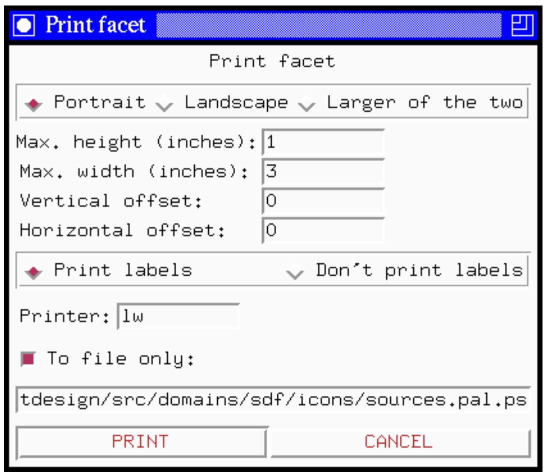
This is a good time to try out your own examples. In general, when you create new galaxies and universes that depend on each other, it is a good idea to keep them together in one directory. For example, all of the facets we have created so far are in the "example" directory. You can use the extensive Ptolemy demos as models.

2.11 Hardcopy

There are several options for printing graphs and schematics developed under Ptolemy. The first option generates a PostScript[1] description and routes it to a printer or file. The second uses the screen capture capability of the X Window system.

2.11.1 Printing oct facets

A block diagram under **`pigi`** is stored as an **`oct`** facet. To print it to a PostScript printer, first save the facet. Do this by moving the mouse to the facet and press the "S" key to save. The facet must be saved. Then, keeping the mouse in the facet, invoke the "print facet" command from the "Other" **`pigi`** menu. You will get a dialog box that looks like this:



Most entries are self-explanatory. The default printer is determined by your environment variable PRINTER, which you can set by putting the following line in your .cshrc file:

```
setenv PRINTER printername
```

You will have to restart **`pigi`** for this change to be registered.

If you select the option "To file only", then PostScript code suitable for importing into other applications will be generated. The image will be positioned at the lower left of the page. The facets displayed in this document were generated this way and imported into FrameMaker. See chapter 19, "Creating Documentation" on page 19-1 for more information.

2.11.2 Capturing a screen image

Under the X window system and compatible systems, there are facilities for capturing screen images. These can be used directly with Ptolemy. However, colors that work well on the screen are not always ideal for hardcopy. For this reason, two sets of alternative colors have been devised. The default works well with color printers, while the second works well with black and white printers. For black and white printers, use the "-bw" command line option when starting Ptolemy, as in:

```
pigi -bw
```

The screen capture command can be used effectively. For example, under X11R4 or X11R5, the following command will print a window on a black and white PostScript printer:

1. PostScript is a registered trademark of Adobe Systems Inc.

```
xwd | xpr -width 4 -portrait -device ps -gray 4 | lpr
```

Other alternatives include a program called **xgrabsc** or some equivalent that may be available on your windowing system. A simple use of this is to generate an encapsulated PostScript image using the following command

```
xgrabsc -eps -page 4x2 -o mySchematic.ps
```

where “mySchematic.ps” is the name of the file into which you would like to store the EPS image. Then with the left mouse button, draw a box around the desired portion of the screen. This command will then save an encapsulated PostScript file four inches by two inches called mySchematic.ps. This file can then be used an a wide variety of document processing systems. For example, to include this PostScript in a TeX document, include the command

```
\include{psfig}
```

in the TeX file and use the commands

```
\begin{figure}
      \centerline{
      \psfig{figure=mySchematic.ps,width=4in,height=2in}}
      \caption{Ptolemy Schematic}
\end{figure}
```

to display the PostScript as a figure within a document.

FrameMaker and some other text processing systems can import and print ordinary color X window dumps. To have these displayed in color on the screen, the following lines may need to appear in your X resources file:

```
Maker.colorImages: True
Maker.colorDocs: True
```

The FrameMaker utility “fmcolor” can be used to reduce a color window dump to a black and white window dump. This will save space and avoid any dithered imitations of color. However, since fmcolor applies a threshold based on color intensity to the image, some foreground colors may get mapped to white instead of black. To prevent this, use the “-cp” command line option when starting **pigi**, as in

```
pigi -cp
```

Then color window dumps can be converted to black and white window dumps using the following FrameMaker command:

```
fmcolor -i 90 filein fileout
```

A useful hint when using such a document editor is to turn off the labels in **pigi** before capturing the image, and then to use the document editor itself to annotate the image. The fonts then will be printer fonts rather than screen fonts. To turn off the labels, execute the **vem** command “layer-display” under the **vem** “Options” menu.

2.12 Other useful information

In this section we cover additional information which may be useful. More advanced topics will be covered in following chapters.

2.12.1 Plotting signals and Fourier transforms

The Ptolemy menu has a submenu called *Utilities* that invokes some useful, frequently used, predefined universes. For example, the “plot-signal” (“~”) command will plot a signal.

The signal can be read from a file, or specified using the syntax for specifying the value of a *floatarray* parameter in **pigi**. For example, if the value of the *signal* parameter is:

```
1 [10] -1 [10]
```

then the "plot-signal" command will plot ten points with value 1.0 and ten points with value -1.0. The *options* parameter can accept any options understood by the pxgraph program (see the pxgraph section of the *Almagest*). To plot a signal stored in a file, simply use the following syntax for the *signal* parameter:

```
< filename
```

You may need to specify the full path name for the file.

Another useful *Utilities* command is *DFT* ("^"), which reads a signal just as above and plots the magnitude and phase of the discrete-time Fourier transform of the signal. These are plotted as a function of frequency normalized to the sampling rate, from $-\pi$ to π. The sampling frequency is assumed to be 2π. A simple phase unwrapping algorithm is used to give more meaningful phase plots. A radix-2 FFT is used, so the order (the number of points) of the fast Fourier transform must be a power of two. That is, the user actually specifies $\log_2(order)$. The order can be longer than the signal, in which case, zero-padding will occur.

2.12.2 Moving objects

Sometimes you may want to move objects around within your schematic. Use the **vem** command *move-objects* ("m") in the *Selection* menu to do this. You can move objects as follows:

- Select the objects that you want to move.
- Using the right mouse button, drag the objects to the desired location.
- Execute *move-objects*, "m".

2.12.3 Copying objects

You can create a new instance of any object in a facet by placing a point where you want the new instance, moving the mouse to the object you wish to copy, and executing "create" ("c"). However, this does not copy the parameter values. If you wish to create a new instance of a star or galaxy that has exactly the same parameter values as an existing instance, you should use the *copy-objects* ("x") command in the **vem** "Selection" menu. To do this, first select the object or objects you wish to copy. Then place a point in the center of the object. Then place a second point in the location where you would like the new object, and type "x". The new object starts life selected, so you can immediately move it, or type "control-u" to unselect it. As of this writing, **vem** unfortunately does not allow you to copy objects from one facet to another.

2.12.4 Labeling a design

It is often useful to annotate a block diagram with titles and comments. The **vem** *edit-label* ("E") command in the "Edit" menu will do this. It takes two arguments: a point specifying the position of the label, and the name of a *layer*, which determines the color of the label. Place a point where you would like the label, and then type a layer name, such as "blackSolid" (with the quotation marks). Then type "E". An Athena widget dialog box like that on page 2-

14 will appear, offering various options. Type the text for your label in the “Label” box. It can contain carriage returns to get more than one line of text. To select a text height (font size) you can move the slider to the right of the “Text Height” box. The middle button moves the slider by large amounts, and the left and right buttons are used for fine tuning. The default, 100, is too big for all but the loudest titles. Sizes 60 and 40 work well with the overall scaling of Ptolemy facets. You can also change the justification by clicking the left button to the right of each justification box. A pop-up menu lists the options. The colors recommended for labels are:

```
blackSolid
blueSolid
brownSolid
greenSolid
orangeSolid
redSolid
violetSolid
whiteSolid
yellowSolid
```

2.12.5 Icon orientation

Most Ptolemy icons have inputs coming in from the left and outputs going out to the right. To get better looking diagrams, you may sometimes wish to reorient the icons. This can be done with the **vem** command “transform” (“t”). Select the icon you wish to transform and type “t” as many times as necessary to get the desired orientation. Each time, you get a 90 degree rotation. Then execute the move-object “m” to commit the change. Notice that a 180 degree rotation results in an upside down icon. To avoid this, reflect the icon rather than rotating it. To reflect it in the vertical direction (exchanging what’s on top for what’s on the bottom), select the object, type “my” (include the quotation marks), type “t” to transform, and “m” to move. To reflect along the horizontal direction, use “mx” instead of “my”. In summary:

To reflect an object horizontally, select it, and type:

```
"mx" t m
```

To reflect it vertically, type:

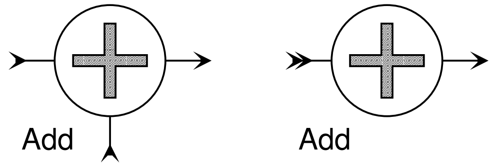
```
"my" t m
```

2.12.6 Finding the names of terminals

Some stars have several terminals, each with a different function. The documentation may refer to these terminals only by name. Unfortunately, the name of a terminal is not normally visible when an icon is viewed with normal scaling. However, zooming in will eventually reveal the name. The easiest way to do this is to draw a box around the terminal and open a new window with the “o” command. Then you can zoom in if necessary. Future versions of Ptolemy will hopefully have a better mechanism.

2.12.7 Multiple inputs and outputs

Ptolemy supports star definitions that do not specify how many inputs or outputs there are. The **Add** and **Fork** stars are defined this way, for instance. Consider the following two icons, found in the "arithmetic" palette of the SDF domain:



They both represent exactly the same star, as you can verify with the "look-inside" command. The icon on the right, however, has a peculiar double arrow at its input. This is a "multiple input" terminal that allows you to connect any number of signals to it. All the signals will be added. The icon on the left has two ordinary input terminals. It can add only two signals. Why have both kinds?

Sometimes, multiple input terminals are not convenient. A rather technical reason is given below, in the section "Auto-forking" on page 2-43. A more mundane reason is simply that schematics often look better with two-input adders.

Let's go through an example of how to create a star icon that has multiple input terminals based on an existing Ptolemy star that supports multiple inputs. Suppose you need an icon for an adder with eight inputs. As of this writing, unfortunately, you need to have write permission in the directory in which Ptolemy icons are stored to create this new icon. Alternatively, you can create your own version of the star in your own directory (see the programmer's manual). If you have write permission in the directory where the icons are stored, then you can create a new icon with eight inputs as follows.

In any facet, execute the **`pigi`** command "Extend:make-star" ("*"). A dialog box appears. Enter "Add.input=8" for the star name, "SDF" for the domain, and "$PTOLEMY/src/domains/sdf/stars" for the star src directory (assuming this is where the source code is stored). Note that "input" is the name of the particular multiple input that we want to specify. If you do not know where the source code is stored, then just look-inside ("i") an existing instance of the star. The **`vem`** console window and the header of the editing window that open both tell where the star source code is. A second dialog box appears asking you where you would like the icon put. Accept the default, "./user.pal". Then open user.pal using "O" to see the new eight-input adder icon. You may edit this icon, as explained in "Editing Icons" on page 2-33.

2.12.8 Using delays

In several domains, delays can be placed on arcs. A delay is not a star, but rather is a property of the arc connecting two stars. The interpretation of the delay in the dataflow domains (SDF, DDF, BDF, and most code generation domains) is as an initial particle on the arc. An initial particle for the scalar data types is one whose value is zero. When the arc passes particles containing "message" type data, a delay on the arc will create an "empty" message. Most often, the destination star of the arc must be able to interpret such "empty" messages explicitly in context of the user-defined type because a "zero" might have different meanings depending on the type. Any feedback look in the SDF domain must have a delay, or the computation in the loop would not be able to begin.

To use these delays in **`pigi`**, the user places a delay icon on top of the wire connecting two instances. The delay icon is a white diamond with a green border in the SDF and system palettes. You can specify the number of delays by executing *edit-params* with the cursor on top of the delay icon.

Other domains (besides dataflow) also use delays, but the meaning can be quite different. See the appropriate chapter describing the domain.

A new feature added to Ptolemy releases greater than 0.5 is the support of initializable delays for simulation domains. These delays use a different icon from the old white diamond with green borders. The new delays use an icon that is a green diamond with a white border and has an "I" in the middle of the diamond to signify that it is initializable. We have kept around the old delays for backward compatibility, but the syntax for the two is quite different and the user should probably use just one type to prevent confusion.

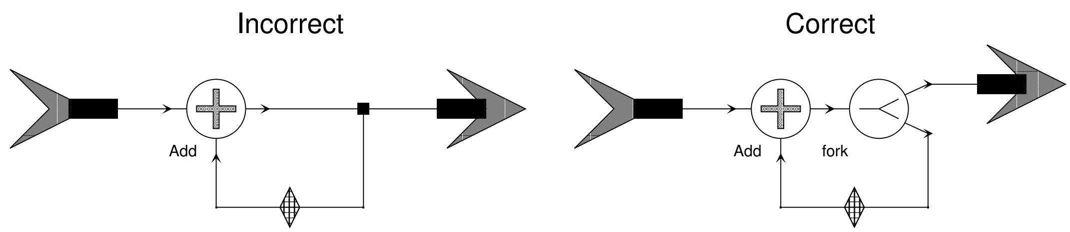
The syntax for the new delays is that the arguments to the delay are the initial value themselves. There is no value in the argument that signifies the number of delay particles. Instead, a count of the number of values in the delay arguments is the number of delay particles that will be added to the buffer of the arc corresponding to the delay. These arguments are speficied as a string and are parsed according to the data type associated to the arc. For example, an initializable delay with parameter "1 0 1" on an arc passing float particles will have a buffer with three initial particles. The three particles will have the values 1.0, 0.0, and 1.0 respectively. If the arc was working on complex particles instead, an error would be given since complex numbers must be specified using a pair of numbers. A proper argument list fo the delay in that case would be "(1,0) (0,0) (1,1)". The shorthand for declaring multiple values in the argument list is valid, just as in the arraystate case. For example, an argument list of "2 [5]" would specify five initial particles with value 2.

Initializable delays also work on arcs which handle matrix particles. The argument string in this case is parsed differently than above. The first two values in the last specify the number of rows and columns in the initial matrix, respectively. For example, an initializable delay with parameter "1 2 3 [2]" on an arc passing integer matrices would place one matrix with dimension one row by two columns, whose entries all have the value three, in the buffer for that arc. For the case where multiple initial matrices are desired, simply give enough entries in the delay argument string to fill multiple numbers of initial matrices of the given size. For example, an initializable delay with paramenter "1 2 3 3 4 4 5 5" on an arc passing integer matrices would create three matrices, all of dimension one row by two columns, such that the first initial matrix on the buffer has all entries equal to three, the second has all entries equal to four, and the last matrix has all entries equal to five.

2.12.9 Auto-forking

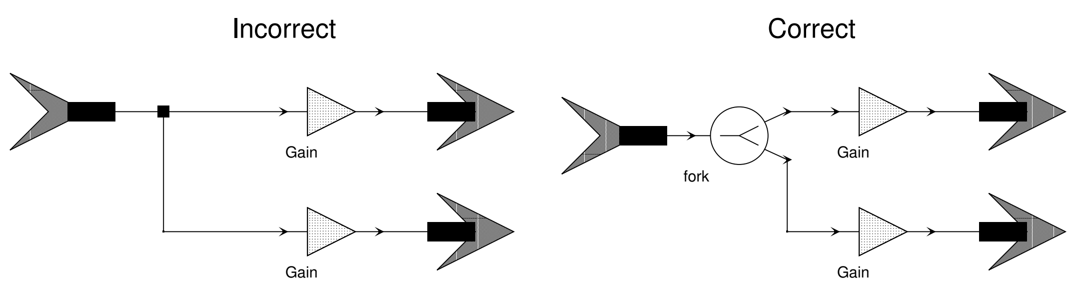
In **`pigi`**, a single output can be connected to any number of inputs, as one would expect. The interpretation in most domains is that the one output is broadcast to all the inputs. There are several point-to-point connections, therefore, represented by the net.

However, there are restrictions. To understand these restrictions, it is worth explaining that **`vem`** stores connectivity information in the form of netlists, simply listing all terminals that are connected together. If a delay appears somewhere on a net, and that net has more than one point-to-point connection, then it is not easy to determine for which connection(s) the delay is intended. Consequently, at the time of this writing, delays are disallowed on nets with more than one connection. If you attempt to put a delay on such a net, then when you try to run the system, an error message will be issued, and the offending net will be highlighted. To get rid of the highlighting, execute the **pigi** command “Edit:clear-marks”. To fix the problem, delete the offending net, and replace it with one or more **fork** stars and a set of point-to-point connections. An incorrect and correct example are shown below:

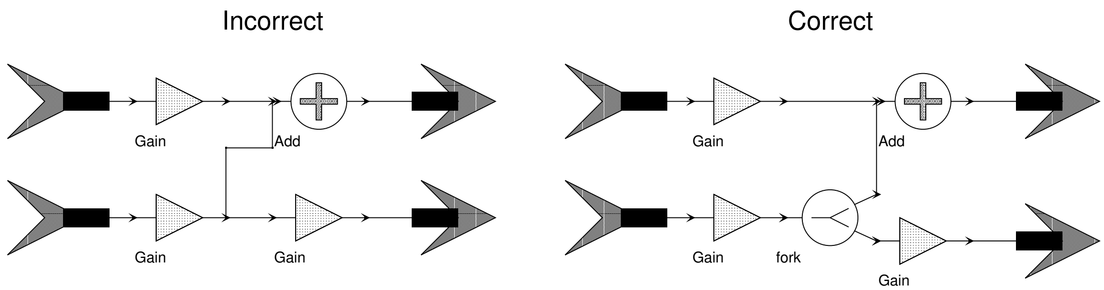


This example also illustrates the use of a delay on a feedback loop. The delay is required here, assuming we are in the SDF domain, because without it, deadlock would ensue. This is due to the fact that the **fork** star cannot fire until the **add** star does, but the **add** star cannot fire until the **fork** star produces its output.

A second restriction is that forks must be explicit when connected directly to input or output terminals of a galaxy. An incorrect and correct example are shown below:



There is also a more subtle restriction. Suppose two outputs are connected to a single multiple-input terminal. Then neither of these outputs can also be connected to some other input terminal. If it is, Ptolemy will issue the error message “multiple output ports found on the same node.” The reason this happens is simple. **Vem** knows nothing about multiple inputs, so it sees a net with more than one output and more than one input. Ptolemy is not given enough information to reconcile this and figure out which outputs should be connected to which inputs. To avoid this problem, it is again necessary to use explicit fork stars, as shown below:



All of the above restrictions may be eliminated in future versions.

2.12.10 Dealing with errors

Ptolemy is composed of several components, as shown in figure 2-8 on page 21. When errors occur, it helps to know which component detected the error so that it can be corrected.

When errors occur in **`vem`**, **`vem`** prints the error in the console window. For example, if you enter a point argument and execute *create* when the cursor is not over an instance, then **`vem`** displays the message "Can't find any instance under spot." Usually, **`vem`** errors are easy to fix. In this case, **`vem`** expects the user to specify the instance to be created.

Errors in the **`pigiRpc`** process can occur when any of the **`pigi`** commands are invoked. The error messages, in this case, are displayed in a popup window, which is much more helpful. Error messages may also be displayed in the xterm window in which **`pigi`** was started. In addition, **`pigi`** often highlights in red the object in the schematic associated with the error. When this happens, you can execute the *clear-marks* command to clear the highlighting. If such an error occurs and the reason for the error is not obvious, try deleting the indicated objects and redrawing them.

2.12.11 Copying and moving designs

In one of our examples, we used "cp -r" to make a copy of a facet. In general, however, copying entire designs this way does not work. For it to work in the general case, you must also change some data in the facets that you copy. In particular, each facet has pointers to the icons it uses. If you move a galaxy, for example, then any pointer to the icon for that galaxy becomes invalid (or "inconsistent" in **`oct`** terminology).

A utility program called "**`masters`**" has been provided for this purpose. This replaces the program from the **`octtools`** distribution, called "**`octmvlib`**", that was used with earlier versions of Ptolemy.

Palettes, star icons, galaxies, and universes are stored as **`oct`** facets. Special care is required when moving or copying **`oct`** facets. First, as emphasized before, every **`oct`** facet is stored as a directory tree, so a copy should use "cp -r". Next, keep in mind that there may be pointers to the moved object in other facets. If you know where all these pointers might be, then moving facets is easy. If you do not know where all the pointers are, then your only practical choice is to leave a symbolic link in place of the old location pointing to the new.

Moving facets

Suppose you have developed a fantastic new galaxy called **alphaCentaur**, and you wish to install it in a directory that is available for general use. Since you have developed the galaxy, you know where it is used. The galaxy icon itself is stored in two facets:

alphaCentaur/schematic/contents;

alphaCentaur/schematic/interface;

The first of these stores the schematic, the second stores the icon. The peculiar semicolon at the end is actually part of the file name. First move the icon:

mv alphaCentaur destinationDirectory

This moves the entire directory tree. You must now change all references to the icon so that they reflect the new location. Suppose you have a test universe called **alphaTest**. This should be modified by running the **masters** program as follows:

```
% masters alphaTest
Running masters on wave
Pathname to replace (? for a listing):
```

User input is shown in bold type; program output is shown in regular (not bold) type. Enter a question mark to get a list of all icons referenced in the facet:

```
Pathname to replace (? for a listing): ?
Pathnames currently found in the facet:
     ~yourname/oldDirectory/alphaCentaur
     $PTOLEMY/src/domains/sdf/icons/Ramp
     $PTOLEMY/src/domains/sdf/icons/Sin
     $PTOLEMY/src/domains/sdf/icons/XMgraph
Pathname to replace (? for help):
```

The last three items are pointers to official Ptolemy icons. There is no need to change these. You should now enter the string you need to replace and the replacement value:

```
Pathname to replace (? for help): ~yourname/oldDirectory
New pathname: ~yourname/destinationDirectory
```

Next, use **masters** the same way to modify any palettes that reference the moved icon. For instance, the “user.pal” palette in the directory in which you developed **AlphaCentaur** is a likely candidate. If you miss a reference, **oct** will issue an error message when it tries to open the offending palette, indicating that it is inconsistent.

2.12.12 Environment variables

The following environment variables can be set to customize certain behavior. These should be set (normally) in the user’s **.cshrc** file.

PRINTER Determines the default printer used for hardcopy output. This is used to determine the default printer when printing **vem** facets. If you use the provided makefile to print Ptolemy documentation, then this environment variable will determine the printer used. In **$PTOLEMY/.cshrc** the pertinent line reads:

```
setenv PRINTER lw
```

You should replace **lw** with whatever printer name you are using.

PT_DISPLAY Determines the text editor used to display text files. This determines how text files will be displayed to the user. The value of this variable is a **printf** format string with one "%s" in it. That %s is replaced with the name of the file to be viewed. The default value is "**xedit -name ptolemy_code %s**", specifying that the program "xedit", part of the standard X Windows distribution, should be used with the command-line option "-name ptolemy_code". Note that, as is lamentably true with so many X Windows programs, **xedit** has ludicrous defaults that yield a useless display. If your X installation has not been configured properly, you will want to change this environment variable to specify your favorite text editor. For example, to view files in a new xterm window with the **vi** editor, put the following line in your **.cshrc** file

```
setenv PT_DISPLAY "xterm -e vi %s"
```

and source the file before starting **pigi**.

PIGIRPC Specifies an alternative executable file for Ptolemy. Ptolemy is an extensible, modifiable system. Many users will wish to create their own versions to incorporate their own extensions. Details on how to write extensions are given in the programmer's manual, volume 3 of the Almagest. Once you (or someone else) has created a customized version, you can invoke it by specifying the precise name of the executable (complete with its full path, or path relative to an environment variable or user's name). The default executable is **$PTOLEMY/bin.$ARCH/pigiRpc**. An alternative specification might be:

```
setenv PIGIRPC ~myname/Ptolemy/bin.sun4/pigiRpc
```

PTX11DIR This specifies the directory name to search for include files and library files for the X Window System. For instance, on Solaris 2.3, the directory should probably be **/usr/openwin**. The directory must have as subdirectories **include** and **lib**. This is used in the CGC domain by stars that require X11 libraries to compile (such as those implementing graphical widgets).

PT_DEBUG If set, this specifies the script to execute when starting pigi in debug mode (using the -debug option). An example of a suitable script is **ptgdb**, located in **$PTOLEMY/bin**. This script invokes **gdb**, the Gnu debugger, inside **emacs**.

2.12.13 Command-line options

The **pigi** program is actually a **csh** script, located in **$PTOLEMY/bin**. That script starts two processes: **vem** and **pigiRpc**. The usage is

pigi [*options*] [*facet-name*]

The optional facet name specifies a **vem** facet that should be opened upon starting the system. The command-line options are:

-bw Use black and white, even on a color monitor. This is useful for generating readable hardcopy from X Window dumps.

-cp Fine tune the colors to improve the quality of hardcopy made on a color printer from X Window dumps.

-console Open a command console through which the user can issue Tcl commands.

-debug Invoke Ptolemy running under **gdb**, a symbolic debugger. If a version of the **pigiRpc** executable with debug symbols can be found, the script will it. The program looks for

$PTOLEMY/bin.$ARCH/pigiRpc.debug,

or if the **PIGIRPC** environment variable is set, **$PIGIRPC.debug**. If neither is found, then **$PIGIRPC** is invoked, if set. Otherwise,

$PTOLEMY/bin.$ARCH/pigiRpc

is used. If the **PT_DEBUG** environment variable is set, then its value is the name of a script used to invoke the debugger. For example, the script **ptgdb**, located in **$PTOLEMY/bin**, invokes **gdb** under **emacs**.

-ptiny Invoke the smallest version of Ptolemy, if it can be found. The executable that is used is called **ptinyRpc**. This version contains only the SDF, DE, and HOF domains.

-ptrim Invoke an intermediate-sized version of Ptolemy, if it can be found. The executable used is called **ptrimRpc**. This version contains only SDF, BDF, DDF, DE, HOF, and CGC, but without the parallel targets.

-display *display-name*
Specify an alternative display to use. If this option is missing, then the **DISPLAY** environment variable is used.

-help print out the usage information

-rpc *ptolemy-executable*
Specify an alternative Ptolemy executable to use. The default is **$PTOLEMY/bin.$ARCH/pigiRpc**.

-xres *X-resource-filename*
Specify an X resource file to merge before running Ptolemy. The standard X program **xrdb** is used with the **-merge** option.

2.13 X Resources

A large number of X window resources can be set by a user to customize various aspects of the user interface. The best way to explore these is to examine the file $PTOLEMY/lib/pigiXRes9 for the defaults. These defaults can typically be overridden in the user's .Xdefaults file, and incorporated into the X environment using the program **xrdb**. For example,

```
Vem*font:  *-times-medium-r-normal--*-120-*
```

changes the font in the **vem** console window, menus, dialog boxes, etc., to something smaller than the default. Also,

```
Vem*background: antiqueWhite
```

changes the background in the **vem** console window and dialog boxes to the color "antiqueWhite."

2.14 Tk options

A number of user interface options are specified through Tk options rather than directly through X resources. These are defined in the file $PTOLEMY/lib/tcl/ptkOptions.tcl. One way to override these is to start pigi with a console window:

```
pigi -console
```

and in the console window, change the options. For example, the command

```
option add Pigi*background gray98
```

changes the dialog box backgrounds to a very light gray. This option was used to create the X window dumps used in this manual.

2.15 Multi-domain universes

The domain of an application is set using the **pigi** "Edit:edit-domain" or "d" command. This command causes a checklist to appear listing all domains currently linked into the system. All examples in the SDF Demo palette are one-domain applications, using only SDF. Several examples of multi-domain applications can be found in the DDF and DE Demo palettes. It is instructive to explore these applications, using the edit-domain command at all levels of the hierarchy to see what domains are used. In addition, the section "Wormholes" on page 9-3 in the DE chapter contains a useful discussion on mixing the DE domain with other domains in Ptolemy.

Recall that a **Wormhole** in Ptolemy is a block that has a different domain on the outside than on the inside. In **pigi**, wormholes look exactly like galaxies. The only difference is that the domain is different on the inside than on the outside.

To build multi-domain applications, it is necessary to understand the models of computation in each domain, to ensure that application will behave consistently at the domain boundaries. For this, it is necessary to refer to the domain chapters in this user's manual.