Chapter 6. Vem — The Graphical Editor for Oct

Authors: *David Harrison*
Rick Spickelmier

Other Contributors: *Bill Bush*
Andrea Cassotto
Christopher Hylands
Edward A. Lee

6.1 Terminology

`Vem` is an interactive graphical editor for the **`oct`** design database. It was written by David Harrison and Rick Spickelmier in the CAD group at UC Berkeley. It has been extended by Andrea Cassotto and Bill Bush. An introduction to the terminology used in the system is given in “The oct design database and its editor, vem” on page 3-18. In this chapter, we give more detailed information about **`vem`**. Most users will not need this much detail; chapter 3 will be enough.

This chapter is extracted from standard documentation for the **`octtools`** distribution. No **`oct`** documentation is included. This would be required only by programmers wishing to modify **`pigiRpc`**; it is available from the Industrial Liaison Program office, Dept. of EECS, UC Berkeley, Berkeley CA 94720.

The fundamental **`oct`** objects that we edit with **`vem`** are called *facets*. Facets are specified by three names separated by colons. This is usually written as “cell:view:facet”. The first component is the cell name; it is used to name the design. Note that cell name may not contain any spaces. In **`pigi`**, the second component, called the view name, will always be “schematic”.[1] Third, is the “facet” component, which can either be “contents” or “interface”. The former specifies a block diagram, while the latter defines an icon. This usage of the term “facet” is different from our previous usage. Thus, “facet” can mean either the **`oct`** object called a facet, or “contents” vs. “interface”. The intended meaning is usually clear from context. In commands that depend on the facet (in the latter sense), if you do not specify it, **`vem`** assumes that you mean the contents facet. Thus, “wave:schematic” refers to the facet with cell name “wave”, view name “schematic”, and facet name “contents”.

`Vem` was originally written with VLSI designs in mind. Ptolemy is an attached tool, invoked via a program called **`pigiRpc`**. **`Vem`** was originally intended for IC design. As such **`vem`** provides standard graphics editing capabilities for physical (mask-level), symbolic, and schematic designs. Ptolemy uses schematic capabilities for applications and physical capabilities for icons.

---

1. Other oct applications use other views such as “symbolic” or “physical.”

Vem may be started by simply typing "**vem**", but this will not start Ptolemy. To start the Ptolemy interactive graphical interface, simply execute the command **pigi** in $PTOLEMY/bin. This invokes a shell script that starts **vem** and the associated **pigiRpc** process. **Vem** is started in general with the following command line options:

```
vem [-F cell[:view[:facet]]] [-G WxH+X+Y] [-R [host,]path] \
     [-display host:display] [name=value ...]
```

For example, the following script could be used to start Ptolemy on a Sun 4 workstation:

```
xrdb -m $PTOLEMY/lib/pigiXRes9
vem -G 600x150+0+0 -F init.pal:schematic \
     -G 600x300+0+170 -R $PTOLEMY/bin.sun4/pigiRpc
```

The first line merges the X Windows resources defined in **$PTOLEMY/lib/pigiXRes9**. The next line starts **vem** and the associated **pigiRpc** process. The **pigi** script is simply a more elaborate version of this that ensures the existence of the "init.pal" facet and sets up the user's environment.

Vem looks at the value of the **DISPLAY** environment variable to determine what host and display to use for output. **Vem** and **pigi** may be run in the background without affecting the program operation.

The **-F**, **-G**, and **-R** command line options allow a user to specify a start-up window configuration for **vem**. These three options are considered triplets that specify the initial cell, position and size, and remote application respectively for a window. There is no limit to the number of triplets that may be specified. The **-F** flag marks the start of each new triplet. The corresponding **-G** and **-R** flags after the **-F** flag are optional. If the **-G** flag is omitted, **vem** will not specify a location for the window and most window managers will interactively prompt for the window location. If the **-R** flag is omitted, no remote application will be started in the window. The **-F** flag can be omitted from the first triplet. In this case, the **-G** and **-R** flags apply to the console window. For example, the **pigi** script above starts **vem** with its console window at (0,0) with a size of 600 by 150, and one window looking at the cell "init.pal:schematic" at (0,170) with a size of 600 by 300, running the "**pigiRpc**" remote application.

Vem is a highly customizable editor. Nearly all of the colors, font styles and fill patterns **vem** uses can be changed by the user. Normally, these parameters are read from the user's X resources (which are usually loaded when X is started from a file named **~/.Xdefaults**, or something similar). However, one can set certain parameters on the command line using the = (equal) command line option. A list of all configurable parameters can be found in the document "Customizing Vem," which is distributed with the standard **octtools** distribution.

The console window echoes user input and outputs various help and status messages. After starting, the console will display a prompt and wait for input. Ptolemy users rarely need to use this window, and eventually, it will be eliminated.

If the **init.pal** facet does not exist in the directory in which **pigi** is started, then it will be created. A blank facet will appear. Convention in Ptolemy dictates that this facet should be used to store icons representing complete applications, or universes, that are defined in the directory. If such icons already exist in the init.pal facet, the applications can be examined using the **pigi** “look-inside” command.

New windows can be created using *open-facet* command (see table 3-2 on page 7). It is also possible to open a window from the **vem** console using the *open-window* command, but this new window will not be attached to **pigiRpc** (see the command reference below). This means that you will not be able to issue the **pigi** commands in table 3-2 from these windows.

Each window has exactly one associated cell. Mouse action with the cursor positioned inside a window cause operations to occur to the associated cell. Any number of windows can be created with the same or different associated cells. More than one window may have the same associated cell. In this case, all of the windows are attached to the *same* cell. Thus, a change to one of the windows may cause updates to other windows that look at the same cell.

Vem assumes a three-button mouse. The left button is used for entry of graphics information. The middle button is used for the primary menu of commands. The right button is used to modify graphics information entered using the left button.

Commands to **vem** are specified in post-fix form. The user builds an argument list first and then selects a command. Commands can be selected in three ways: pop-up menus, single keystrokes, or by typing in the command name. Pressing and releasing the middle button in a graphics window causes a **vem** menu to appear. The user can use the mouse to riffle through the options until the desired choice is highlighted. The commands are summarized in table 3-3 on page 11. Pressing and releasing the mouse button activates the selected command. Pressing and releasing the mouse outside the menu cancels the selection. Normally, pressing and releasing the middle button causes a **vem** menu to appear. Holding the shift key and clicking the middle button causes the **pigi** menu to appear. Both menus are useful.

A number of common commands can be selected via a single keystroke. Key bindings for various commands are shown next to the corresponding entry in the **vem** menu, are listed in the command reference below, and can be queried interactively using the *bindings* command. Typing a colon (:) allows the user to type in the command name (or a user defined alias) in the console window. The standard line editing keys can be used while typing the command name. This interface supports automatic command completion. Typing a tab will complete the command if it is unique or offer a list of alternatives if it is not unique. The command is selected by typing a carriage return.

There are five types of input to **vem**: points, boxes, lines, text, and objects. Points are entered by pressing and releasing the left button of the mouse. Boxes are entered by pressing, dragging, and releasing the left button. Lines are entered by pressing, dragging and releasing the left button over a previously created point or line. Text is entered by typing the text enclosed in double quotes. If entering a filename, typing a TAB character will cause **vem** to try to complete the name if it is unique or offer a list of alternatives if it is not unique. Objects are entered using the *select-objects* and *unselect-objects* commands. The last item on an argument list can be deleted using the standard character for delete. The last group of items can be deleted using the word erase character control-W. The entire argument list can be deleted using the standard kill-line character (usually control-U).

Once entered, graphics arguments (points, boxes, and lines) can be modified in various ways. For all arguments, a point can be moved by moving the cursor over the point and pressing, dragging, and releasing the right mouse button. New points can be added to a group of lines by moving over a point in the segment, depressing, dragging, and releasing the left mouse button. This will insert a new point after the point. It is also possible to interactively move, rotate, and mirror objects arguments (selected items). See the description of the *transform* command in the command reference below.

This version of **vem** supports three basic editing styles: physical, symbolic, and schematic. Physical editing involves the entry and editing of basic geometry and the creation of interface terminals. This style is used in **pigi** to build icons. Symbolic editing involves the placement of instances of leaf cells and the interconnection of these instances. **Pigi** does not use symbolic editing. Schematic editing is an extension of symbolic where the primitive cells are schematic symbols and wire width is insignificant. Schematic cells use used by **pigi** to represent block diagrams.

When **vem** opens a new cell directly (i.e. not via **pigiRpc**), a dialog will appear asking for three property values: TECHNOLOGY, VIEWTYPE, and EDITSTYLE. The TECHNOLOGY and VIEWTYPE properties determine the location of the technology facet, which specifies the colors and layers in the display. A standard technology facet has been designed for Ptolemy, so the defaults that appear are almost always acceptable. Layer display and design rule information is read from this facet. The EDITSTYLE property is used by **vem** to determine the set of commands available for editing the cell. Currently, the legal editing styles are PHYSICAL, SYMBOLIC, or SCHEMATIC.

6.2 Using Dialog Boxes

Some commands require information that cannot be expressed easily using post-fix notation. Examples include destructive commands that require an explicit confirmation and commands that require complex non-graphic information. **Vem** uses *dialog* boxes based on the MIT Athena widgets to handle these situations. Dialog boxes are windows that resemble business forms. These windows contain labeled fields for entering text, changing numerical values, and selecting options. This section describes how to use dialog boxes.

All dialog boxes in **vem** have the same form. An example is shown on page 3-14, and also in figures 6-1 on page 12 and 6-2 on page 14. At the top of all dialogs is a one line title indicating the purpose for the dialog. The middle of the dialog (known as the body) contains fields for displaying and editing information of various kinds. At the bottom of the dialog are a number of *control* buttons. Each control button represents a command. The arguments to the command are the values of the fields displayed in the body. Thus, operating a dialog consists of editing or changing fields in the body and then selecting a command by activating a control button. Six kinds of fields may appear in the body of a **vem** dialog: editable text, non-editable text, enumerated value, numerical value, exclusive lists, and non-exclusive lists. A description of each field type is given in the paragraphs that follow.

Editable text fields are used to enter and edit text. Visually, an edit text field consists of a box containing a caret cursor, an optional scrollbar, and a label to the left of the box indicating the purpose for the field. Only one editable text field is active in any one dialog. The active editable text field has a dark border. Typing text with the mouse positioned anywhere in the dialog inserts text into the active editable text field. Most of the basic emacs editing commands can be used to modify the text in the field, as shown in table 6-1.

The insert position in the field may also be changed by pressing the left mouse button when the mouse cursor is over the desired position. Any editable text field can be made active by clicking the left mouse button inside the editable area. Alternatively, one can use the <tab> key to make the next text field active and <meta-tab> to make the previous field active. Editable text fields that display large amounts of text have a scrollbar to the left of the text area. Pressing the left and right mouse buttons when the mouse cursor is in a scrollbar will scroll the text down and up respectively in proportion to the distance between the mouse cursor and the top of the scrollbar. As an example, pressing the left mouse button near the bottom of the scrollbar will scroll down the text almost one screen. Pressing and releasing the middle mouse button scrolls the text to a relative position based on how far the mouse cursor is from the top of the scrollbar. Holding down the middle mouse button will interactively scroll through the text.

Non-editable text fields are used to display text messages. They consist of a box containing text and an optional scrollbar. The scrollbar operates just like those used in editable text fields.

Enumerated value fields are used to specify one value out of a small list of values. They consist of a value displayed inside a box and a descriptive label to the left of the value. The border of the value highlights as the mouse cursor moves over it. Depressing and holding the left mouse button inside the value box causes a menu to appear that displays all possible values. The choices will highlight as the mouse cursor moves over them. To select a new value, release the mouse button when the desired choice is highlighted. The new value will appear in the value box. One can leave the value unchanged by releasing the mouse button outside the menu boundary.

A numerical value field is used to specify a magnitude between a predetermined minimum and maximum. Visually, it consists of a box containing a numerical value, a horizontal scrollbar to the right of the box for changing the value, and a label to the left describing the

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
meta-d	Delete next word
control-e	Move to end of line
control-f	Move forward one character
meta-i	Include a file
control-k	Kill (delete) to end of line
control-n	Next line
control-p	Previous line
control-s	Search forward
control-v	Next page
meta-v	Previous page
control-y	Yank deleted text

TABLE 6-1: Emacs-style text editing commands supported in vem dialog boxes.

value. The magnitude of the value is changed by operating the scrollbar. Pressing the left and right buttons in the scrollbar decrement and increment the value by one unit. Pressing the middle button changes the value based on the distance between the mouse cursor and the left edge of the scroll bar. The middle button may be pressed and held to interactively modify the value. Most people use the middle button to set the value roughly then use the left and right buttons to make the value precise.

Exclusive lists are used to choose one possible value out of a (possibly quite large) list of values. These values are displayed in a box with a scrollbar on the left edge of the box. Each value consists of a button box on the left and a descriptive label to the right. As the mouse moves over a button box, it will highlight to indicate it can be activated. The button box of the selected item will appear dark while all others will remain light. If there are too many values to display in the box at one time, the scrollbar can be used to scroll through the possible values. The scrollbar operates in the same way as described for editable text fields.

Non-exclusive lists are used to choose zero or more possible values from a (possibly quite large) list of values (see figure 6-2 on page 14). A non-exclusive list resembles an exclusive list both in appearance and operation. However, unlike an exclusive list, one can choose any number of items in a non-exclusive list. Visually, the two lists are distinguished by the appearance of the button boxes. Exclusive button boxes resemble radio buttons. Non-exclusive button boxes resemble check marks. Pressing the left button in a non-exclusive button box causes the value to toggle (i.e., if it was selected it becomes unselected, if it was unselected it becomes selected).

Control buttons cause the dialog to carry out some operation. They consist of a text label surrounded by a box. Control buttons are activated in one of two ways: pressing and releasing the left mouse button when the mouse cursor is positioned inside the button boundary, and through keystrokes. Not all control buttons can be activated using keystrokes. Those that can be activated in this fashion display the key in parentheses under the button label. Although there are exceptions, most dialogs support the keyboard accelerators given in table 6-2.

Dialogs may be both *moded* and *unmoded*. Moded dialogs are those requiring a response before processing can proceed. **`Vem`** uses these kinds of dialogs to ask for confirmation before proceeding. On the other hand, unmoded dialogs remain active until explicitly dismissed by the user. Other commands may be invoked freely while unmoded dialogs are visible. Most non-confirmation dialogs in **`vem`** are unmoded.

Key	Action
<return>, <meta-return>, <F1>	OK
<delete>, <meta-delete>, <F2>	Cancel
<Help>, <F3>	Help
<F4>	All
<F5>	Clear

TABLE 6-2: Keyboard accelerators supported by most vem dialog boxes.

6.3 General Commands

Below is a reference for all **vem** commands. This section outlines general commands available for editing all types of cells. Section 6.4 discusses options, section 6.5 describes the general selection mechanism. and section 6.6 describes property and bag editing features. The next three sections describe editing commands for physical, symbolic, and schematic cells respectively. The summary includes the name of the command as it appears on a menu, if it appears in the menu. The command name can be typed in as well. Place the mouse in the window where you wish to execute the command, enter the command arguments (points, objects, etc.), type a colon ":", and type the command name. The TAB character will automatically complete command names. The phrase <no-name> implies it has no default menu or command name binding.

The list below also shows the default keyboard binding for each command, if it has one, and the syntax of the argument list passed to it. The symbol <*> implies the command has no default key binding. In general, the commands used most often have key and menu bindings. Less often used commands may have only command name bindings. See table 3-3 on page 11 for a concise summary.

Some **vem** commands are not documented here because they are dangerous or conflict with the objectives of **pigi**. Those commands will not appear in the **vem** menu, and have no key binding, although all are still available by typing them in. Adventurous users may wish to consult the standard **octtools** documentation before using them.

<no-name> **<del>,^H** **Any Argument List**

This command deletes the last item of the last argument on the argument list. Thus, if the last argument is 10 boxes, it will delete the last box entered and the argument list will be modified to contain 9 boxes.

<no-name> **^W** **Any Argument List**

This command is similar to the one above, but deletes the entire last argument on the argument list. Thus, if the last argument is 5 lines, it will delete all 5 lines and leave the remaining arguments unchanged.

<no-name> **^X or ^U** **Any Argument List**

This command erases the entire argument list allowing the user to start over.

bindings **b** **Saves Arguments**

This command asks the user for a command and displays all of its current key, menu, and alias bindings. The command will display a prompt (**vem** bindings>) and the user can specify a command using any of the four means of normally specifying commands (via menu, single keystroke, type-in, or last command). The command also outputs a one line description of the command for help purposes.

close-window **^D** **No Arguments**

The *close-window* command closes the window the cursor was in when the command was invoked. This DOES NOT flush the contents of the window to disk. Even after all windows looking at a facet are closed, the contents are not saved on disk. This must be done using the *save-window* or *save-all* commands.

deep-reread **<*>** **[objs]**

The *deep-reread* command is a specialized form of the re-read command. With no arguments, it re-reads a facet and all of master facets of its subcells (instances). Both the contents and interface facets of the instances are re-read. If a set of objects is specified, the command re-reads the master cells of the instances in the object set. Only the master cells of the instances are re-read; cells are not re-read recursively when using this form.

interrupt **^C** **No Arguments**

This routine interrupts (deactivates) the window containing the cursor. No drawing will be done in the window until a full redraw requested by the user (using pan, zoom, or redraw-window) is done. The key binding for this command can also be used while a window is drawing to immediately stop drawing in that window.

kill-buffer **<*>** **"cell view {facet} {version}"**

The *kill-buffer* command flushes a facet out of memory *without* saving its contents. If the string specification of the facet is missing, the facet is determined by the window containing the cursor. *All* windows looking at this facet are destroyed. There are no key or menu bindings for the command and it will ask the user for confirmation before carrying out the command.

log-bindings **<*>** **Saves Arguments**

The *log-bindings* command writes out a description of all type-in, menu, and key bindings for all commands in the editing style of the window containing the cursor. This description is written to the log file for the session.

open-window **o** **[box] or**
"cell:{view:{facet:{version}}}"

The *open-window* command is primary way to create new graphics windows in **vem**. It takes a string specifying the cell to open. When specifying the cell portion of the name, typing a TAB will attempt to complete the string as a file or offer alternatives if the name is not unique. If this string is absent, it will duplicate the window containing the cursor. Normally, the extent of the duplicated window is the same as the parent window. However, if the user specifies a box, the duplicated window will be zoomed to that extent (see zoom-in). The string specifying the cell contains four fields. The last three are optional and default to “physical”, “contents”, and the null string respectively. It is possible to specify your own defaults for these fields. Newly created windows are always zoomed to contain all geometry in the cell. If the cell does not exist, it will be created. When creating new cells, **vem** prompts the user for required cell properties. See the introduction for details. Most of the time, the defaults presented in this dialog are acceptable and activating the *Ok* button is sufficient to proceed.

palette **P** **{"palette-name"}**

The *palette* command opens a new window onto a previously created facet which contains standard layers or instances for a given technology. This window can be used to select layers for creating geometry or instances for instantiation. The command takes one argument: the name of the palette. If omitted, it defaults to “layer”.

Palette cells are found using the function tapGetPalette (see tap(3)). For all standard technologies and viewtypes, there is a “layer” palette. In the symbolic editing style, there are also “mosfet” and “connector” palettes which display mosfets and connectors respectively. In the schematic editing style, there are “device” and “gate” palettes which contain device level and gate level schematic primitives. New palettes can be added easily. See “Customizing Vem” for details.

pan **p** **[Any Arguments] [point]**

The *pan* command centers the window containing the cursor around the last point on the argument list. The window will be redrawn so that the argument list point is now the center of the window. The point need not be in the same window as the cursor. Thus, a user can point in a window showing a large portion of a cell and invoke the command in a more detailed window for a magnifying glass effect. If the point is omitted, the command assumes the cursor position is also the desired center point. This is the fast way to pan in a single window.

pop-context **)** **No Arguments**

This command pops off an input context from the context stack and replaces the current context with that context. See the *push-context* command for details.

push-context **(** **No Arguments**

This command pushes the current argument list context onto the context stack and gives the user a new context. This can be used to do other commands while preserving entered arguments. Note that the current arguments remain displayed. The old context can be restored using the *pop-context* command. Four context levels are supported in the current version of **vem**.

push-master **<*>** **{"facet"}**

The *push-master* command opens a new window on the master of the instance under the mouse cursor. This command can be used all editing styles. If a facet name is supplied, the command will use that facet name instead of “contents”. In Ptolemy, this command is rarely needed. The *edit-icon* command accomplishes the same objective.

recover-facet **<*>** **No Arguments**

Unless directed otherwise, **vem** saves all cells occasionally in case of a system crash or some other unforeseen disaster. Whenever a new cell is opened, **vem** checks to see if the last automatically saved version is more recent than the user saved version. If the automatically saved version is more recent, a warning is produced and the user version is loaded. One can use the *recover-facet* command to replace the cell with the more recent automatically saved version. The command displays a dialog containing a list of all of the saved versions. Generally, there are two possible alternative versions for a cell. The *autosave* version is written by **vem** automatically after a certain number of changes to the cell. The *crashsave* version is written when **vem** detects a serious error. Note that the crashsave version may itself be corrupt since a serious error occurred just before it was written. This command is destructive: it replaces the cell with the selected alternate. One can use the *Cancel* button to abort the recovery. Before using the *recover-facet* command, it is often useful to view the alternate cells. This can be done by specifying the version (either “autosave” or “crashsave” as the last field in the cell specification to *open-window.*

re-read **<*>** **No Arguments**

The *re-read* command flushes the facet associated with the window containing the cursor out of memory and reads it back in from the disk. This can be used to see changes to a cell that were done outside **vem** or to revert back to the cell before changes were made. This is a dangerous command and **vem** asks for confirmation before proceeding.

redraw-window **^L** **[box]**

This command redraws the contents of the window containing the cursor. It does not effect the argument list (i.e. it can be done regardless of the argument list contents). If a box is provided, only the portion of the window in the box is redrawn. If the window is interrupted, this command will reactivate it. However, if the box form is used, **vem** will only draw the specified area and leave the window deactivated. This can be used to selectively draw portions of a deactivated window.

same-scale **=** **[Any Arguments] [point]**

This command changes the scale of the window containing the mouse to the same scale as the window containing the last point on the argument list. It is commonly used to compare the sizes of two facets.

save-window **S** **[Any Arguments]**

This command saves the contents of the facet associated with the window where the command was invoked. It asks for confirmation and does not effect the argument list.

set-path-width **w** **[box] [line] [point] or ["size string"]**

This command sets the current path width for a given layer on a window-by-window basis. It takes one argument which may be points, lines, boxes, or text. For points and lines, the argument length must be two. The path width is set to the maximum Manhattan distance between the points. For boxes, only one box is allowed and the new width is the larger of the two dimensions. For text, the string should contain the path width in lambda. If no width is specified, the path width will be set to the minimum layer width as specified in the technology.

The layer for the path width command is determined by looking at the object under the cursor. If there are objects on more than one layer under the cursor, a dialog will be presented and the user should choose one of the listed layers and press “OK” to continue.

show-all **f** **No Arguments**

The *show-all* command causes **vem** to zoom the window containing the cursor so that all of the cell is displayed in the window. The key binding is an abbreviation for “full”. It does not effect the argument list.

switch-facet **<*>** **["cell view {facet {version}}"] or [point]**

This command replaces the facet in the window containing the mouse with a different facet. The first form replaces the window’s facet with the named facet. The second form replaces the window’s facet with the facet of the window containing the point.

toggle-grid **g** **No Arguments**

The *toggle-grid* command toggles the visibility of the grid in the window containing the cursor.

version **V** **No Arguments**

This command outputs the current version of **vem** to the console window.

where **?** **No Arguments**

The *where* command can be used to find out the position of the cursor in terms of **oct** units. It also displays a textual representation of the objects under the cursor. The command can be issued while building an argument list without effecting the list. Alternatively, if the argument list includes an object set, the where command will print textual descriptions of the selected items.

write-window **W** **"cell:view" [any arguments]**

The *write-window* command saves the contents of the cell associated with the window where the command was invoked under another name. This alternate name is specified by the "cell:view" argument.

zoom-in **z** **[Any Arguments] [box]**

The *zoom-in* command zooms the window containing the mouse to the extent indicated by the last box on the argument list. The box and the zoom window must be in the same facet. However, the extent may be in a different window from the mouse which can be used to achieve a magnifying glass effect. If the box is not provided, it zooms the window containing the mouse in by a factor of two. If provided, the command removes the box from the argument list but leaves other arguments untouched.

zoom-out **Z** **[Any Arguments] [box]**

The *zoom-out* command is the opposite of zoom-in. This command zooms the window containing the mouse out far enough so that the OLD contents of the window are contained in the extent of the box provided on the argument list. Thus, smaller boxes zoom out farther than larger ones. If the box is omitted, the window containing the mouse is zoomed out by a factor of two.

6.4 Options

The options commands below all post form windows which allow a user to change display parameters interactively. The default values for these parameters can be changed in your **~/.Xdefaults** file. See the separate document "Customizing Vem" for details.

All of the options dialogs below are *unmoded.* This means that the user can do other things while the dialog is posted. They will not go away until explicitly closed by the user. Details on the operation of dialog boxes can be found in "Using Dialog Boxes" on page 6-4.

window-options **<*>** **No Arguments**

This command posts the dialog in figure 6-1, which presents a number of window related options each of which can be modified by the user on a window-by-window basis. Two kinds of options are presented in this dialog: flag options and value options. Flag options have a check box on the left of the option and can be either on or off. Pressing the left mouse button in the check box toggles the value of the option. Value options display a numerical value in a box with a descriptive label to the left and a scrollbar to the right for changing the value. At the bottom of the dialog are four control buttons: *Ok, Dismiss, Apply,* and *Help.* Pressing the left mouse button inside the *Ok* button saves any options you may have changed and closes the window. The *Dismiss* button does not save any options and closes the window. The *Apply* button saves any changed options but does not close the window. The *Help* button will open a window containing a brief description of the dialog. Each of the options and their meanings are given below:

"Visible Grid"

If set, a grid will be shown in the window.

"Dotted Grid"

If the grid is visible and this option is set, the grid will be drawn as dots rather than lines.

"Manhattan Argument Entry"

If this option is set, entering line arguments and dragging option sets will be restricted to Manhattan angles. This option is set by default in the symbolic and schematic editing styles.

"Argument Gravity"

If set, all lines entered using the left button whose endpoints are near an actual terminal will be snapped to that terminal. This is especially useful when editing schematic diagrams (**vem** automatically turns this option on when the edit style is set to schematic). The .Xdefault parameter *vem.gravity* specifies the maximum distance between line endpoints and terminals for gravity to have an effect (by default, 10 pixels).

"Show Instance Bounding Boxes"

If this option is set, **vem** will display bounding boxes around all instances. The instance name will be displayed in the center of these bounding boxes.

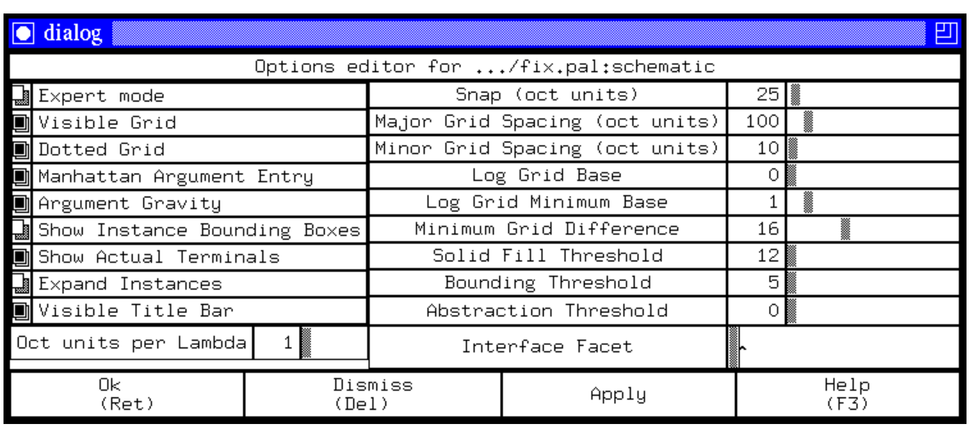

FIGURE 6-1: The "window options" dialog box in vem.

"Show Actual Terminals"
: When viewing very large designs, drawing the highlighting around actual terminals can be expensive. If this option is turned off, **`vem`** will not draw highlighting around actual terminals.

"Expand Instances"
: If on, the contents facet of instance masters will be displayed. Otherwise, the interface facet will be displayed. This has the same effect as the *toggle-expansion* command.

"Visible Title Bar"
: If this option is set, **`vem`** will display its own title bar above each graphics window. If the option is turned off, the title bar will be turned off.

"**`Oct`** units per Lambda"
: This parameter specifies the number of **`oct`** units per lambda. The output of the where command and the coordinate displays in the title bar are displayed according to the value of this parameter. By default, **`vem`** uses 20 **`oct`** units per lambda.

"Snap"
: All graphic input into the window will be snapped to multiples of this parameter (given in **`oct`** units). By default, **`vem`** snaps to one lambda (20 **`oct`** unit) intervals.

"Major Grid Spacing"
: This parameter specifies the grid spacing of major grid lines in **`oct`** units. If logarithmic grids are turned on, it specifies a multiplying factor for major grid line spacing (see Log Grid Base below).

"Minor Grid Spacing"
: This parameter specifies the grid spacing of minor grid lines in **`oct`** units. If logarithmic grids are turned on, it specifies a multiplying factor for minor grid line spacing (see Log Grid Base below).

"Log Grid Base"
: If non-zero, this option selects a logarithmic grid. Normally, there are two grids drawn at a fixed number of **`oct`** units (major and minor grid lines). In a logarithmic grid, grids are drawn at some constant (specified by the Major Grid and Minor Grid parameters above) times the nearest integral power of the grid base. For example, if the constant is two and the base 10, grids will be drawn at 2, 20, 200, etc.

"Minimum Grid Threshold"
: This parameter specifies the smallest allowable space (in pixels) between grid lines before **`vem`** stops drawing them.

"Log Grid Minimum Base"
: This parameter specifies the smallest base to be used for drawing logarithmic grids (see above).

"Solid Fill Threshold"
: This parameter specifies the size (in pixels) before a shape is drawn using solid fill rather than stipple fill. A large number specifies all geometry should be drawn solid regardless of its size.

"Bounding Threshold"
: Label text drawn in bounding boxes (e.g. instances or terminals) may or may not be drawn depending on the size of the bounding box. This parameter specifies how many times wider the text may be than the box before the label is not drawn.

"Abstraction Threshold"
: This parameter specifies the maximum size of a bounding box (in pixels) where it is acceptable to draw a filled box rather than an outline to speed the drawing process.

"Interface Facet"
: This string parameter specifies the name of the displayed interface facet. This facet will be used to draw instances in unexpanded mode.

`layer-display` **<*>** **No Arguments**

This command posts the dialog shown in figure 6-2, which can be used to selectively turn on or off the display of any layer. At the bottom of the dialog are six control buttons. The *Ok, Dismiss, Apply,* and *Help* buttons work in the same way as described for *window-options.* The *All* button automatically selects all of the layers displayed in the body of the dialog. The *Clear* button automatically unselects all of the layers displayed in the body of the dialog. Above the control buttons there is a list of all layers displayed in the window. The layers currently shown in the window have buttons to the left of the layer name that have check marks. Those that are not shown have buttons to the left of the layer name that appear empty. The state of a layer can be changed

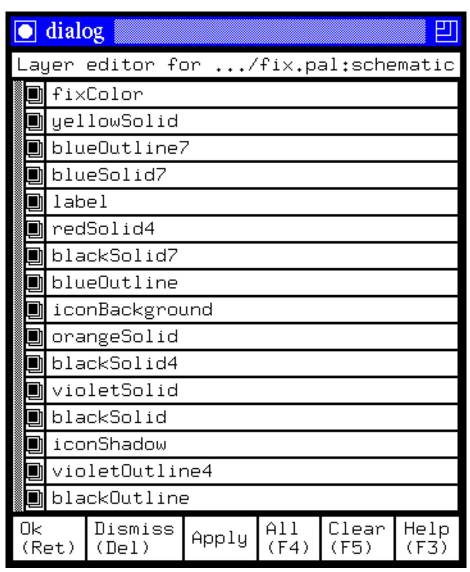

FIGURE 6-2: Layer display options in vem.

by moving the cursor over the corresponding button and depressing the left mouse button. Note that no window update will occur until either *Ok* or *Apply* are pressed.

6.5 Selection

The selection commands described below are used to manipulate object arguments on the argument list.

`select-layer` **`.`** **`[objs] [pnts] [lines] [boxes] "layer"`**

The *select-layer* command is similar to the *select-objects* command but allows the user to select only the geometries on a particular layer. This layer can be specified in two ways: the layer name can be typed in as the last argument or the command will try to determine the layer by looking at the geometry under the cursor when the command is invoked. If the spot for the layer is ambiguous, **`vem`** will post a dialog presenting a choice between the layers.

`select-objects` **`s`** **`[objs] [pnts] [lines] [boxes] "layer"`**

The *select-objects* command is used for placing collections of objects on the argument list for further processing by other commands. It takes as arguments any number of points, lines, or boxes. Points select items under the point, lines select objects which cross the line, and boxes select objects inside the box. If *select-objects* is not given any point, line, or box arguments, it will try to select items under the cursor where the command was invoked. These semantics are described in detail below.

For each point, the command adds zero or more of the objects under the point to the list. If there is more than one object under the point, a dialog will be posted with buttons representing each of the objects under the point. Clicking the mouse in one of the buttons highlights the object. Once the user has clicked on the desired objects, the *Ok* button is used to actually select the items.

For each line argument, the command adds all objects which intersect the line. This is useful for schematic drawings where paths (wires) are zero width. Selection using lines works best if the entered lines are Manhattan. Non-Manhattan lines may select more objects than intended.

For each box, the command adds all objects completely contained in the box to the list. Note that an object is considered contained if and only if its *bounding* box is *completely* inside the given box. The *select-objects* command is incremental; i.e. it may be called many times, each time adding to the selected set. All items selected are highlighted in the **`vem`** highlight color.

`select-terms` **`^T`** **`[objs][points][lines][boxes]`**

This command selects all terminals (both actual and formal) whose implementations intersect the objects found by examining the argument list. The semantics for specifying the objects is identical to that described for *select-objects.* The items on the argument list will be replaced with the set of terminals found by examining the items. This command is useful for deleting formal terminals, specifying actual terminals for use by *edit-property* or *select-bag,* or creating new symbolic formal terminals using *promote.* As such, *it is extremely dangerous in Ptolemy.*

`transform` **`t`** **`[objs]`**

This command takes a selected set of objects built by selection commands and transforms them. The objects remain on the selected set. It is important to note that the actual objects in the database are not affected by this command. The transformation is associated with the object set not with the objects themselves.

The command takes (up to) three arguments: a set of objects to work on, a text rotation specification, and two points indicating a relative translation. The object set must be supplied. The rotation specification is a list of keywords, enclosed in quotation marks, separated by colons:

"mx"	Mirror around the Y axis.
"my"	Mirror around the X axis
"90"	Rotate counter-clockwise 90 degrees.
"180"	Rotate counter-clockwise 180 degrees.
"270"	Rotate counter-clockwise 270 degrees.

If both a rotation and a translation are given, the rotation specification should come first. If no rotation and translation are given, the command rotates the items 90 degrees counter-clockwise.

The *transform* command is incremental. This means it can be applied many times with each command having a relative effect on the current selected set. For example, invoking *transform* without arguments twice is the same as invoking it once and specifying the rotation string "180".

Once the command completes, the highlighted form of the objects will reflect the specified transformation. This can be used as a reference for further rotations or translations.

Translations are specified by two points. A relative translation is applied to the current transformation based on the vector formed by the two points. It is also possible to interactively specify the translation of a selected set. This is done by moving the cursor to a reference point and pressing, dragging, and releasing the right mouse button. While the right button is depressed, the selected objects will track the cursor motion. This can be used to drag items around interactively.

After completing a transformation with transform, the objects are not actually changed until a command that manipulates objects is invoked (e.g. move-objects, copy-objects, etc.). See the *move-objects* and *copy-objects* commands below for more information.

`unselect-objects` **`u`** **`[objs] [pnts] [lines] [boxes]`**

The *unselect-objects* command is used to remove items from a previously created object argument (*select-objects* operation). Any number of points, lines, and boxes may be specified. The semantics for mapping these arguments to objects is the same as *select-objects.* For each point, zero or more items beneath the point that are part of the selected set are removed. For each line, all items in the selected set intersecting the line are removed from the set. For each box, all items completely contained in the box that are part of the selected set are removed from the set.

It is important to note that this command is intended to be used to unselect small sets of already selected objects. To unselect all items on the argument list, use control-W or control-U.

6.6 Property and Bag editing

The **`oct`** Data Manager supports two kinds of annotation: properties and bags. Properties are named objects which may have an arbitrary value and can be attached to any other **`oct`** object. Properties are used by **`pigi`** to store parameters. Bags are named objects which can contain any number of other **`oct`** objects, and are not used by **`pigi`**. Bags are used to represent common collections of objects (instances for example) that can be accessed efficiently. There are a number of **`vem`** commands used to create, edit, and delete properties and bags. However, since only one of these is useful in **`pigi`**, only one is documented here.

`show-property          <*>            [objs]`

The *show-property* command produces a list of all the properties associated with the object under the cursor when the command was invoked (or all objects in the selected set). If there are many objects under the cursor, the command will present a dialog which lists each object. The user can select one by clicking the mouse in the check box next to the object name. The object will be highlighted. When the right object is found, the user can click *Ok* to show the properties attached to the object. If the cursor is not over an object and nothing is in the selected set, the properties attached to the facet are shown. The properties are shown in the form: xid: name (type) = value. The *xid* is the external identifier for the property and can be used in evaluated labels. The command also echoes the type of the object each property is connected to.

6.7 Physical editing commands

The physical editing style in **`vem`** is used by **`pigi`** for editing icons. The commands described below are available in addition to the common commands described in the previous section.

`alter-geometry         a              [box] [lines] or [pnts]`

This command replaces the box, path, or polygon under the cursor with the new specification supplied on the argument list. This can be used to "stretch" geometry or change their composition. For example, to make a box slightly larger, enter the slightly larger box onto the argument list, move the mouse over the old box, and invoke *alter-geometry.*

`change-layer           l              [objs] or [pnts][lines][boxes] "layer"`

The *change-layer* command detaches geometry from its current layer and attaches it to a different layer. The geometry can be specified either as an object set constructed by the selection commands, or directly by drawing points on them, drawing lines through them, or drawing boxes around them. Normally, the target layer is determined by looking at the geometry under the cursor when the command was invoked. However, if the last argument to the command is text, it is interpreted as the name of the target layer.

copy-objects **x** **[objs] {pnt pnt}**

The *copy-objects* command copies a set of objects from one place to another. The command takes an object argument that should contain a list of objects to be moved (this is built with *select-objects* and *unselect-objects*). The command assumes the object set has been transformed using the *transform* command or interactively dragged to a new location with the right mouse button. The command makes a copy of the objects where are transformed according to this translation. For example, to copy objects from one location to another, the user first selects the objects using *select-objects,* then interactively drags the objects using the right button (transformation), then invokes the *copy-objects* command to make a copy at the new location. Since the items remain selected, new copies can be made without reselecting the objects.

The optional second argument should be two points which specify the source and destination points of the copy. This alternative can be used if the object set is too large for interactive dragging or one wishes to copy objects from one facet into another. If the copy is from one facet to another facet, terminals will not be copied and the objects will be copied in a manner that preserves the position of the objects relative to the source point. The default key binding for this command is short for “xerox”.

create-circle **C** **[line] [pnts] "layer"**

Since **vem** does not have a circle argument type, a special circle drawing command has been added. For most types of geometry, the user should use *create-geometry.* Circles are specified in one of two ways. The first is a line followed by up to two points. The line specifies a filled circle with the first point being the center and the second its outer radius. If the first point is supplied, an arc is assumed with an angle formed by the second point of the line, the center point and the newly specified point. The angle is measured counter-clockwise. If the last point is supplied, it specifies the inner radius of the circle (otherwise the inner radius is zero). The second form takes two points and an optional point. It specifies a circle where the inner and outer radius are the same. If the last point is supplied, an arc with the same semantics as the first form is assumed. Finally, the layer of the circle is determined from the cursor position or by a final text argument specifying the layer directly.

create-geometry **c** **[pnts] [lines] [boxes] [text] "layer"**

The *create-geometry* command creates new geometry. It takes any number of points, lines, boxes, or text and a layer specification. A points argument creates a closed polygon. A line argument creates a multi-point path. Box arguments create boxes. Finally, text arguments create labels. When creating labels, the point set after the label is interpreted as the target points for the label. All the geometry is created on the same layer. This layer may be specified as the final text argument to the command or by invoking the command over an object attached to a layer. If the layer is ambiguous, the command will present a choice of layers in a dialog box. The palette command can be used to create a window which offers all possible layers for creating geometry.

create-instance **<*>** **[pnt] {"master:view name"}**

In most cases, the leaf cells designed with the physical editing style are not hierarchical. Instead, instances of the low-level cells are connected together using the symbolic editing style. However, those who would like to use **vem** as a purely physical design editor require instances in physical cells. This command places instances in the physical editing style.

The instance is placed relative to the point supplied to the command. The master of the instance can be specified in two ways. In the first form, the user supplies a text argument which contains the cell, view, and instance names separated by spaces. The instance name is optional. The second form determines the master of the instance by looking at the instance under the cursor when the command is invoked. This instance can be in the same cell or in another cell. A common practice is to build a cell of primitives and use this cell as a menu for placing physical instances (see the *palette* command).

`create-terminal      <*>          ["term name"]`

This command creates a new formal terminal named “term name”. The implementation of the terminal is determined by constructing a list of all geometries under the spot where the command was invoked and choosing the smallest coincident boxes from this list. *This command is dangerous in* **`pigi`**.

`delete-objects       D            [objs] or [pnts] [boxes] "layer"`

The delete command removes objects from a cell. The command has two forms. The first takes an object argument constructed using selection commands and deletes all of the items in this set. The second takes any number of point, line, and box arguments and a layer specification. This form deletes all objects under the points, all objects which intersect the lines, and all objects completely contained in the boxes that are attached to the specified layer. The layer may also be specified by placing the cursor over some other object attached to a layer when the command is invoked. If no layer is specified, all of the geometry is deleted.

`edit-label           E            [pnt] {"LAYER"} or [objs]`

The *edit-label* command creates and edits labels. The command has two forms. The first form creates a new label at the specified point on the given layer. If the layer is not specified, it will be determined by looking at the object under the cursor. The second form edits labels selected using the *select* command.

Labels in **`oct`** are represented as a box where the text is drawn entirely inside the box subject to justification and text height parameters. The edit-label command builds the box automatically by examining the text height and text itself. Thus, the user can control the justification, text height, and the label text. These parameters are set using the edit-label dialog box. This is a modeless dialog box that is posted when the user invokes the edit-label command.

The edit-label dialog box, shown on page 14, consists of three check-box areas for adjusting the label justification, and two type-in fields for adjusting the text height and the text itself. The justifications are computed in relation to the point the user specified when the label was first created. Thus, the horizontal justification specifies whether the point should be to the left, center, or right of the text. Similarly, the vertical justification specifies whether the point should be on the bottom, center, or top of the text. Finally, the line justification specifies how the lines should be justified within the text block when there is more than one line. The text height of the text is given in **`oct`** units. Note that the X window system does not directly support fully scalable fonts.

Thus, **vem** uses a strategy where it will pick the closest font from a set of fonts that can be specified as a start-up parameter (see the document "Customizing Vem" for details). Finally, the last type in field can be used to enter the text for the label. The label can have as many lines as necessary.

At the bottom of the dialog are four control buttons: *Ok, Apply, Dismiss,* and *Help.* The *Ok* button updates the value of the label and causes the dialog to close. The *Apply* button updates the value of the label (showing the effects in the graphics window) but does not close the dialog. This allows the user to adjust the label several times if necessary. The *Dismiss* button closes the dialog without updating the value of the label. Finally, the *Help* button displays some help about how to use the dialog.

move-objects   m   [objs] {pnt pnt}

The *move-objects* command moves a set of objects from one place to another. The command takes an object argument that should contain a list of objects to be moved (this is built with *select-objects* and *unselect-objects*). The command assumes the object set has been transformed using the *transform* command or interactively dragged to a new location using the right mouse button. The command moves the objects to a new location based on this transformation. For example, to move objects from one location to another, the user first selects the objects using *select-objects,* then interactively drags the objects using the right button (transformation), then invokes the *move-objects* command to actually move the items to the new location. The items remain selected for further moves if necessary.

The optional second argument should be two points which specify the source and destination points of the move. This alternative can be used if the object set is too large for interactive dragging. Unlike the *copy-objects* command, objects cannot be moved from one facet to another.

6.8 Symbolic editing commands

The symbolic editing commands in **vem** allow the user to edit **oct** symbolic views. Symbolic views are used to represent layout in a form suitable for compaction and simulation. Since they are not used by **pigi**, they are not documented here.

6.9 Schematic editing commands

The schematic editing style is an extension of symbolic. In addition to the general, selection, and options editing commands, the following commands are specific to the schematic editing style:

delete-objects   D   [objs] or
[pnts, lines, boxes] ["layer"]

This command takes either an object list created with the *select-objects* and *unselect-objects* commands, or points, lines, and boxes with an optional layer-name. The resulting objects are deleted from the cell.

select-major-net   ^N   pnts, lines, or boxes ["net name"]

This command finds the net associated with the object under the points, intersecting the lines, or inside the boxes and highlights all objects on that net. If no points, boxes, or lines are specified, the object under the cursor will be examined. Alternatively, if a net name is provided, objects on the named net are highlighted. This command can be used to check the connectivity of a symbolic cell. The command is incremental (i.e. multiple nets can be selected).

`move-objects` **`m`** **`[objs] {pnt pnt}`**

The *move-objects* command in schematic editing mode is similar to the same command in physical editing mode above. One difference, however, is that the connectivity of the items moved by this command is not changed. This means an instance moved using this command also causes the segments attached to its terminals to move as well. Moving objects between facets is not supported.

`copy-objects` **`x`** **`(See Below)`**

The *copy-objects* command copies a set of objects from one place to another. This command has the same form as *move-objects* (see above) except that the objects are copied not moved. Like *move-objects,* the objects copied remain on the argument list for further move and copy operations. However, unlike *move-objects,* the connections to the objects in the selected set are not copied. Instead, the connectivity between the selected items is copied along with the objects.

`create` **`c`** **`(See Below)`**

The *create* command allows the user to add new objects to a schematic cell. Different arguments given to *create* will produce different objects.

Formal Terminals — "terminal-name [type] [direction]" : create

When *create* is given a single argument string, the name of a new formal terminal, a formal terminal is created. The implementation of the formal terminal is taken to be the actual terminal currently under the mouse (note: a connector terminal can also be used for this purpose). Since terminals in schematic may be quite small, this routine will try to find nearby terminals if it doesn't find a terminal directly beneath the cursor.

Formal terminal names must be unique within the cell. If a formal terminal of the given name already exists, **`vem`** will display a dialog box asking whether or not you wish to replace the old terminal.

Two optional pieces of annotation can be placed on the terminal: type and direction. The type can be one of SIGNAL, CLOCK, GROUND, SUPPLY, or TRISTATE. If not provided, SIGNAL will be assumed. The direction can be one of IN, OUT, or INOUT. If not specified, INOUT will be assumed. If either of these values are not provided in the terminal name specification, **`vem`** will post a dialog box containing fields for entering both the terminal type and direction. Pressing the left mouse button in the value area for these fields will cause a menu of the possible choices to appear. New values can be selected by releasing the mouse button over the desired value. Once appropriate values are selected, activating the *Ok* button at the bottom of the dialog will save the annotations. Activating the *Dismiss* option will leave the annotations unspecified. These annotations can be edited later using the edit-property command.

Instances — pnts [[master:view] [instance-name] : create

If the arguments to *create* are a number of points followed by an optional string, instances will be created with their origins at those points. If the name of the master is not specified textually, it will be inferred from the instance under the mouse.

The string argument has two parts: the instance specification and name. Both of these fields are optional. Specifying a null string is considered to be the same as no string at all. The instance specification is a master-view pair, such as “amp:schematic”. If this field is left out, the master is inferred from the instance under the mouse. Otherwise, an attempt is made to locate the instance by the master-view pair. If the name field is given, the instance will be given the specified name.

NOTES: Newly created instances whose actual terminals intersect actual terminals of other instances will be automatically connected. In this case, no path is required between the terminals. Rotated and mirrored forms of instances can be created by instantiating a new instance and using the transform and move-objects or transform and copy-objects commands.

Paths — lines : create

This command creates new segments for connecting together instance actual terminals. A new series of segments will be created on a predefined layer (WIRING). Connector instances will be placed automatically at all jog points. The width of the new path is always zero.

Normally, the schematic editing style has a feature turned on called “gravity”. When you draw segments with gravity turned on, **`vem`** will try to connect the segments to a nearby terminal if you miss a terminal by a small amount. This is useful when editing a large cell.

`edit-label` **`E`** **`[pnt] {"layer"} or [objs]`**

The *edit-label* command creates and edits labels, and is identical to the version in physical editing mode, documented above.

6.10 Remote application commands

The following commands apply only if a remote application, like **`pigi`**Rpc, is running.

`kill-application` **`<*>`** **`No Arguments`**

The *kill-application* command kills the remote application which has control of the window where the command was invoked. This can be used to terminate runaway remote applications. Note it has no standard menu or key bindings; it is a type-in command only. This command may not work on all machines.

`rpc-any` **`r`** **`"host pathname"`**

This command starts up a remote application which is not on the standard list of applications in the **`vem`** menu. “host” specifies the network location for the application and “path” specifies the path to the executable on “host”. If the host specification is omitted, the local machine is assumed.

`rpc-reset` **`<*>`** **`No Arguments`**

If an application terminates abnormally, **`vem`** may not recognize that the window no longer has an application running in it. This command forces **`vem`** to reset the state of the window so that new applications can be run in it.

6.11 Bugs

- Opening a facet that is inconsistent (either out of date or one with conflicting terminals) is not handled very gracefully.
- Bounding boxes may not be drawn if there is no geometry in the cell.
- The set path width command doesn't work if you use a palette to specify the layer.