For a list of known bugs, see the *Bugs* file in the Pad++ home directory.

CHANGES

For a list of changes since prior version, see the *ChangeLog* file in the Pad++ home directory.

CONTACT

Please use the following email address for contacting us:

- *pad-bug@cs.unm.edu:* Bug reports.
- *pad-comment@cs.unm.edu:* Comments, questions, suggestions, etc.
- *pad-info@cs.unm.edu:* Information requests.
- *pad-users@cs.unm.edu:* List of people currently using Pad++.pad users.

quicker in the middle, and ends slowly - resulting in smoother feeling animations. This does not affect the time the animation takes because time is effectively stolen from the middle to put at the ends. User-controlled animations are specified precisely by the user, and there is no distortion of the motion speed.

LOGO

The first Pad++ widget created within a session has a small logo in the bottom right corner of the screen.

PADDRAW SAMPLE APPLICATION

Pad++ comes with a sample application called PadDraw. It is written entirely in Tcl and allows interactive creation of multiscale items and navigation within the Pad++ dataspace. This can be executed by running the '*paddraw*' script in the top-level Pad++ directory.

PadDraw is contained in the draw subdirectory of the Pad++ distribution. It is intended to be both a sample application for experimenting with the Pad++ widget from a user's level - as well as being a model for the developer of new Pad++ applications. There are several files that may be useful to the application programmer. In particular, *events.tcl* contains all the event-handler code that can be used for creating panning and zooming behavior.

Startup Files

When PadDraw starts up, several different kinds of files are loaded that control the look of PadDraw, as well as defining more specific things. Some of these files come with the Pad++ distribution, and some are available for individual customization. They are loaded after the Pad++ System file *~/.padinit*, in the following order:

- *padrc* - Distribution file: Contains X resource file. Specifies default colors, sizes, and fonts for system in standard X resource file format.
- *~/.padrc* - Customization file: Put your own changes to the X resource file here. They will override any definitions in the padrc file.
- *paddefaults* - Distribution file: Contains Tcl code. Specifies defaults for the PadDraw application, including colors.
- *~/.padsetup* - Program generated file: Do Not Modify! This is used to store information about the setup of PadDraw between runs. For instance, the visibility and geometry of tool palettes are stored here.
- *~/.paddefaults* - Customization file: Put your own changes to the paddefaults here. They will override any definitions in the paddefaults file. For example, your could redefine the color palette.

CREDITS

The scripting language interface to the Pad++ widget was greatly inspired by John K. Ousterhout's canvas widget in Tk. The zooming concept was originally described by Ken Perlin and David Fox at New York University in SIGGRAPH'93.

Pad++ is being developed by a DARPA funded consortium led by Jim Hollan at the University of New Mexico in collaboration with New York University.

The development group is being led by Ben Bederson (UNM), and consists of people at UNM: Jim Hollan, Allison Druin, Ron Hightower, Mohamad Ijadi, Jason Stewart, David Thompson, Ying Zhao, and people at NYU: Ken Perlin, Jon Meyer and Duane Whitehurst.

In addition, other people that have been involved with the Pad++ project include: David Bacon, Duco Das, David Fox, David Vick, Eric De Mund, David Rogers, Mark Rosenstein, Larry Stead, and Kent Wittenburg.

We also especially appreciate Paul Haeberli (of SGI) who gave us code to read and render Adobe Type 1 fonts.

Pad++ is supported in part by DARPA contract #N66001-94-C-6039.

BUGS

widgets. See the description of *Portal Items* in the reference manual, as well as the description of the **bind** command for more information about portals.

• *Events*

It is possible to attach event handlers to items on the Pad++ surface so that when a specific event (such as ButtonPress) reaches an item, that event handler fires, and an associated command is executed. This system operates much as it does with the Tk Canvas widget, but there are several significant additions. See the pad **bind** command for complete details, but the extensions fall into the following three categories:

- Extra macro expansions are added
- New events are added: <Create>, <Modify>, <Delete>, <Write>, and <PortalIntercept>.
- User-specified modifiers are added.

• *Callbacks*

In addition to the event bindings that every item may have, every Pad++ item can define Tcl scripts associated with it which will get evaluated at special times. There are four types of these callbacks:

• *Render Callbacks*

A render callback script gets evaluated every time the item is rendered. See the `-renderscript` itemconfigure option for more detail.

• *Timer Callbacks*

A timer callback script gets evaluated at regular intervals, independent of whether the item is being rendered, or receiving events. See the `-timerscript` and `-timerrate` itemconfigure options for more detail.

• *Zooming Callbacks*

A zooming callback script gets evaluated when an item gets rendered at a different size than its previous render - where the relevant size thresholds are definable. This is a simple way of making "semantically zoomable" items - that is, items that look different when the are rendered at different sizes. See the `-zoomaction` itemconfigure option for more detail.

• *View Change Callbacks*

A view change callback gets evaluated whenever the view onto the Pad++ surface changes (as a result of commands such as **moveto**, **center**, etc.) See the `-viewscript` itemconfigure option for more detail.

• *Extensions*

It is possible to add user-defined types and options to Pad++ entirely with Tcl scripts (i.e., no C/C++ code). This provides a mechanism to define new compound item types that are treated like first-class Pad++ items. That is, they can be created, configured, saved, etc. with the same commands you use to interact with built-in items, such as lines or text. These extensions are particularly well-suited for widgets, but can be used for anything. See the *Application-Defined Item Types and Options* section above for a detailed description.

• *Animation*

Pad++ has several methods for producing animations. The **moveto** command animates the view of the surface to any new point in a specified time. Individual items can be animated with either render or timer callbacks. Finally, panning and zooming is animated under user-control, defined by scripts supplied with the PadDraw application.

All automatic animations use *slow-in-slow-out* motion. This means that the motion starts slowly, goes

command is called. Many commands implicitly damage items, but damage can be triggered manually with the **damage** command.

EFFICIENCY

Several techniques are used internally for Pad++ to work efficiently, even when there are large numbers of items. Understanding these techniques may help the application designer build faster programs. The techniques are:

- *Refinement*

  Pad++ renders the scene at different resolutions depending on how much time is available. For example, while panning or zooming and during other forms of interaction, Pad++ attempts to render the scene as quickly as possible. It does this by not drawing items that are very small, and drawing larger items at lower resolution. When the interaction is over, after a short pause, the system refines the scene a bit at a time, until it is drawn at its highest resolution.

  These iterative refinements each have numbers. Refinement level 0 (sometimes referred to as render level 0) is the fastest, and thus lowest-resolution rendering. Each ensuing refinement is labeled with the next larger integer. The starting refinement level (which is usually 0) can be controlled with the -defaultRenderLevel configuration option.

- *Level of detail*

  Some built-in items are drawn differently depending on how much time is available to draw them, and how big the item appears on the screen. The two main examples are images and text. Images are drawn at user-specifiable resolutions at low levels of refinement, and at high-resolution at higher refinement levels. Text is drawn as hashmarks, at small sizes, and as horizontal lines at extremely small sizes.

  In addition, facilities are provided so that user-defined items can be drawn with different levels of detail. The render and zooming callbacks can be used in combination with the **getsize**, and **render** commands to modify the way the item is drawn depending on the refinement level and size of the item.

- *Region management*

  See the above section on Region Management and Screen Updating.

- *Adjustable frame rate*

  Pad++ adjusts the frame rate during animations and zooming to maintain constant perceptual flow, independent of processor speed, scene complexity, or window size. For example, if a specific animation is specified to take 750 milliseconds, just enough animation frames are rendered so that the animation takes 750 milliseconds.

- *Interruptible*

  Whenever the system is doing a slow task, such as refining or animating, any event (such as a key-press or mouse-click), will interrupt the task and control will immediately be given back to the user. For example, pressing the space-bar during an animation will immediately bring you to the end of the animation. Panning during a refinement will immediately stop the refinement and go on with high-speed panning. Interruption can be turned off with the `-interruptible` configuration option.

UNIQUE PAD++ FEATURES

While Pad++ is modeled after the Tk Canvas widget, there are several unique features of Pad++, in addition to the basic multiscale concept.

- *Portals*

  Portals are a special type of item in Pad++ that sit on the Pad++ surface with a view onto a different location. Because each portal has its own view, the surface might be visible at several locations, each at a different magnification, through various portals. In addition, portals can look onto surfaces of other Pad++

a moment, the equivalent X font is loaded and replaces the Pad++ rendering of the text. For this reason, Times and Helvetica fonts look better than other Type 1 fonts. After this refinement, the fonts at that size are much prettier. You can control when and for which size X fonts are used. Font types are specified with the same style that Java fonts are specified. A single string with font family, style, and size completely specifies a font. For example, "`Helvetica-bold-14`" specifies a 14 point bold Helvetica font. If the style is not specified then plain is assumed, and if size is not specified, then a default of 12 points is used. These font strings can be used wherever Pad++ is expecting a font, such as in the **`-font`** itemconfigure option available for many item types.

In order for Pad++ to use Adobe Type 1 fonts, it must be able to find them. These fonts are stored in files with "*.pfa*" extension. They are typically stored in the */usr/lib/X11/fonts/Type1* or */usr/X11R6/lib/X11/fonts/Type1* directories. These directories are examined by default to see if any Type 1 fonts are stored there. If they are stored elsewhere, Pad++ can be directed to search other directories with the "`font path`" command (i.e., you could use a command such as "`.pad font path /tmp/type1`".) You can find which fonts are available on the current system with the "`font names`" command (i.e., "`.pad font names`".) See the **`font`** command for more detail.

Pad++ comes with a font named "*Line*". This is a fixed-width vector-font, modeled on courier. It is quite fast, and while readable at smaller sizes, is not very pretty when magnified.

IMAGES

Pad++ supports real-time zooming of images where the rendering speed is dependent only on the size of the resulting image. It is independent of the source image size. This real-time image zooming is only possible when the Pad++ program is being displayed on the console of the machine Pad++ is running on (i.e., it is quite a bit slower when Pad++ is being displayed across a network).

DISPLAY LISTS

There is a single display list maintained by Pad++ which controls the order in which items are drawn. The drawing order of items can be changed with the **`raise`** and **`lower`** commands. Pad++ also supports a notion of layers which can be used to force certain items to be drawn above or below other items. Every item sits on a layer, and every item on a layer is drawn sequentially before items on any other layers are drawn. In addition, the visibility of layers can be turned on and off so layers can be used to quickly control visibility. See the **`layer`** command to define new layers and the **`-visiblelayers`** itemconfigure option to control visibility of layers.

Pad++ also supports sticky items. These are items that do not move as the view changes, but instead always appear at a fixed position relative to the screen. Sticky items can be used to implement things like status lines and windows. Sticky items are put on the regular display list and can appear above or below any other item. If you want sticky items to appear above other items, then you should put them on a special layer which is raised above the main layer.

STARTUP FILE

When Pad++ starts up, the file *~/.padinit* automatically gets loaded before anything else. The file *~/.padinit* is a standard Tcl script that can contain any code the user wants. A typical use for this file is to start up a Pad++ application. To make the PadDraw (described below) application start up automatically when you run padwish, put this line in your *~/.padinit* file:

```
source $env(PADHOME)/draw/pad.tcl
```

The PadDraw application uses several other startup files as well. They are described below in the PadDraw section.

REGION MANAGEMENT AND SCREEN UPDATING

Only the portions of the screen that change get rendered. This makes many operations much faster. Specifically, modifying or dragging individual items, or panning the view is much more efficient. Notably, zooming is no faster - and still requires re-rendering the entire view.

This screen updating is controlled by *region management* which uses the concepts of damage and repair. When an item changes, the region within its bounding box is automatically damaged. The act of damaging a region adds it to a list that gets scheduled for repair. The repair doesn't happen until either the system is idle, or the **`update`**

```
> (pad '.pad 'bind 22 '<Leave> "(pad '.pad 'ic %O '-pen 'black)")
> (settoplevel 'tcl)
> %
% puts [expr 2 + 2]
4
% .pad create line 0 0 0 50
23
% .pad settoplevel scheme
scheme
% >
>
> (exit)
surf[165]
```

Adding a new interpreted scripting language to Pad++ requires creating some C++ interface code, and modifying the Pad++ C++ substrate to access that code, and build a new padwish executable. To add a language, several callback procedures must be defined, and then a new instance of the Pad_Language class must be created in tkMain.C. The necessary callback procedures are:

• *Callback when a pad widget is created.* Whenever a new Pad++ widget is created, this routines gets called so a function for accessing that new widget can be created in the new scripting language.

```
Pad_CreateProc   *create_proc;
```

• *Callback to process pad command.* From Tcl, there should be a command for accessing the new scripting language without using the top-level interpreter. For example, it should be possible to load and evaluate files, and access variables from that language. This routine gets called when that Tcl command is evaluated, and it must implement this functionality.

```
Pad_CommandProc  *command_proc;
```

• *Callback specifies if command is complete and should be evaluated.* This routines, given a command in the scripting language, typically counts parenthesis, quotes, and braces, and determines if the command is partial or complete. This routine is called whenever the user presses the return key at the top-level interpreter to determine if the command should be evaluated yet.

```
Pad_CompleteProc *complete_proc;
```

• *Callback to generate a prompt.* Given a flag specifying whether the command is complete or not (according to the complete_proc, this routine should generate a distinctive prompt for this language.

```
Pad_PromptProc   *prompt_proc;
```

• *Callback to evaluate a string.* This routine should evaluate the specified string as a command in the scripting language.

```
Pad_EvalProc     *eval_proc;
```

There are several relevant C++ files that should be examined to see how Scheme is currently connected to Pad++. The files are all in the PADHOME/src directory. *pad-scheme.C* implements all of the callback routines that form the connection between C++ and scheme. *script.h* declares the Pad_Language and Pad_Script classes. *script.C* implements those classes. *tkMain.C* instantiates the Pad_Language class for each available scripting language.

FONTS

Pad++ supports Adobe Type 1 fonts. The Times and Helvetica fonts are special in that when the system sits idle for

```
                $PAD ic $option_id -text "$value: "
            } else {
                set value [$PAD ic $option_id -text]
                set len [expr [string length $value] - 3]
                set value [string range $value 0 $len]
            }
        }
        -value {      ;# Handle "-value" option
            set option_id [lindex [$PAD ic $id -members] 1]
            if {$got_value} {
                $PAD ic $option_id -text "$value"
            } else {
                set value [$PAD ic $option_id -text]
            }
        }

        default {return -code error "Unknown option: $option"}
    }

    return $value
}
```

The property item can be created and accessed just like any built-in item. The following code shows how one might use a property item.

```
set prop [.pad create property]
.pad itemconfig $prop -option "color"
.pad itemconfig $prop -value "blue"
set color [.pad itemconfig $prop -value]
puts "color of property is $color"
```

SCRIPTING LANGUAGES

Pad++ is a prototyping system, and as such, is intrinsically connected to an interpreted scripting language for writing programs that create items and interact with them. By default Pad++ comes with the Tcl scripting language, however, other languages may be added. Note that these other scripting languages can access the full functionality of Pad++, but can not access any of the Tk interface system. The substrate supports a fairly general mechanism for incorporating new scripting languages. We have done this with the Elk version of Scheme. The README.SCHEME file describes specifically how to build Pad++ with Scheme support, and how to access Pad++ from scheme. If Pad++ is built with Scheme included, then the **setlanguage** and **settoplevel** commands will apply to Scheme as well as Tcl. These commands control which language is to be used.

The **setlanguage** command specifies what language is to be used to evaluate all callback scripts that are created in the future. The **settoplevel** command specifies what language the toplevel interpreter should use. In addition, the padwish executable has a -language option that specifies what language the interpreter should start using. It defaults to Tcl. The following session trace shows how the two languages work together:

```
surf[164] padwish -language scheme
Real-time image zooming supported.
> (+ 2 2)
4
> (pad '.pad 'create 'line 0 0 50 50)
22
> (pad '.pad 'itemconfig 22 '-penwidth 5)
> (pad '.pad 'bind 22 '<Enter> "(pad '.pad 'ic %O '-pen 'red)")
```

regular way with the **itemconfigure** command, and will get written out with the **write** command. Similar to the **addtype** command, **addoption** defines a script that gets evaluated whenever the user-defined option is accessed for an item of the specified type. The script must return the new value of the option. When the script is evaluated, two or sometimes three extra parameters are added on to the end of the string. They are:

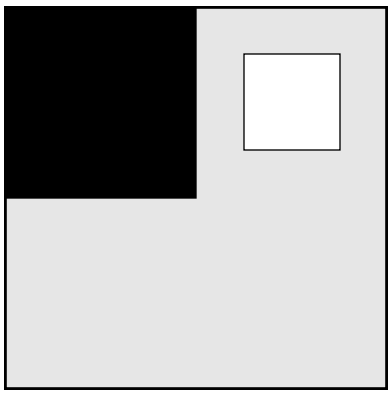
- *pathName:* The name of the pad widget the item is on.
- *item:* The id of the item being configured.
- *[value]:* Optional value. If value is specified, then the option must be set to this value.

The following example shows how to define a new "property" type that has two slots, -option and -value. When an item of type property is created, It appears as text with the option on the left and the value on the right separated by a colon and some space.

```
#
# Add new "property" type
#
.pad addtype property propCreate

#
# Define script to handle creation of property item
#
proc propCreate {PAD} {
    set option [.pad create text -anchor e -text "option: "]
    set value [.pad create text -anchor w -text "value"]
    set group [.pad create group -members "$option $value"]

    return $group
}

#
# Add "-option" and "-value" options to the property type
#
.pad addoption property -option "propConfig -option" "option"
.pad addoption property -value  "propConfig -value"  "value"

#
# Handle property item configuration
#
proc propConfig {args} {
    set option [lindex $args 0]
    set PAD [lindex $args 1]
    set id [lindex $args 2]
    set got_value 0

# Access arguments
    if {[llength $args] >= 4} {
        set value [lindex $args 3]
        set got_value 1
    } else {
        set value ""
    }
    switch -exact -- $option {
        -option {     ;# Handle "-option" option
            set option_id [lindex [$PAD ic $id -members] 0]
            if {$got_value} {
```

If we zoom in a bit with the **moveto** command, both items will appear larger. This, however, is not the same as magnifying each item with the **scale** command. Changing the view affects the way all items are rendered (except *sticky* items, see below). Transforming items changes just the way those items are rendered.

```
.pad moveto 0 0 2
```



We can find the current view with the **getview** command:

```
.pad getview                                    => "0 0 2"
```

EVENT BINDINGS

It is easy to attach Tcl scripts to items so that when the user interacts with that item (via a mouse button press, key press, or whatever), the Tcl script is evaluated. This is implemented with the **bind** command. For example, the following code creates two squares. Clicking on the left one zooms in a bit, and clicking on the right one zooms out a bit.

```
.pad create rectangle 0 0 50 50 -tags rect1 -fill black
.pad create rectangle 100 0 150 50 -tags rect2 -fill white
.pad bind rect1 <ButtonPress> {.pad moveto 0 0 2 1000}
.pad bind rect2 <ButtonPress> {.pad moveto 0 0 1 1000}
```

APPLICATION-DEFINED ITEM TYPES AND OPTIONS

Pad++ may be extended entirely with Tcl scripts (i.e., no C/C++ code). This provides a mechanism to define new Pad++ types and a way to define options for those types (or built-in types). These user-defined types and options are treated like first-class Pad++ objects. That is, they can be created, configured, saved, etc. with the same commands you use to interact with built-in objects, such as lines or text. These extensions are particularly well-suited for widgets, but can be used for anything.

For example, the PadDraw application defines a few sample widgets such as a checkbutton with the type mechanism. It also defines `-roughness` and `-undulate` options for the built-in line type.

Types and options are defined with the **addtype** and **addoption** commands, respectively. The **addtype** command defines a new type with a script that gets evaluated whenever a new item of that type is created with the **create** command. The pathname of the pad widget is added on to the script as an extra parameter when the script is evaluated. The script must return the id of an item that it creates that is to be treated as the new type. Any item type can be created for this purpose, and it will be treated as the new type. If a `-renderscript` is attached to this item, then this item type can have any desired visual look. Alternatively, the script might create a group with members that define the item's look.

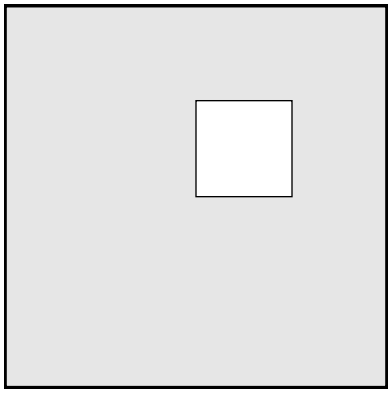
The **addoption** command defines a new option for a built-in or user-defined type. This option is accessed the

position and size. Items can be moved and scaled with the above mentioned commands, but the anchor and anchor point can also be accessed directly with the `-anchor`, and `-position` itemconfigure options. `-position` consists of a list of three values where the first two values specify the position of an item relative to its anchor, and the third value specify its size. For example, an image with a `-position` of "25 30 2" and an `-anchor` of "`center`" will appear centered at the point (25, 30) with a magnification of 2. Changing its anchor to "**`w`**" will make the west side of the image appear at the point (25, 30).

Some items (such as text and images) get created with a `-position` of (0, 0, 1). Items with coordinates, however, (lines, rectangles, polygons, and portals) are special in that these items get created with a `-position` that is determined by the coordinates. So, if we create a rectangle from (0, 0) to (50, 50) with a center anchor gets a `-position` of "25 25 1". Note that **`-anchorpt`** is a shorthand way of accessing the first two components of the `-position`, and `-scale` accesses the last.

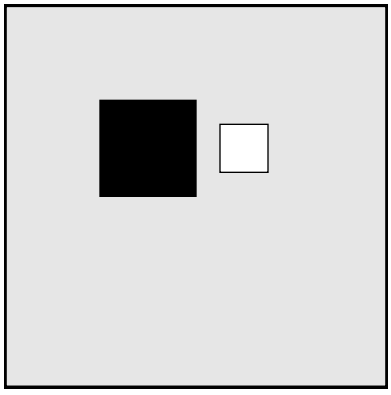
Let's look at how the various transformations work. We'll start by creating two squares at the origin.

```
.pad create rectangle 0 0 100 100 -tags "rect1" -fill black
.pad create rectangle 0 0 100 100 -tags "rect2" -fill white
```



Notice that only the white rectangle is visible because both rectangles are drawn at the same place, and the one drawn last appears on top. We now slide the black rectangle to the left and shrink the white one.

```
.pad slide rect1 -100 0
.pad scale rect2 .5
```



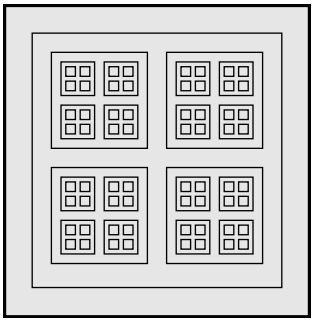
These commands operate by modifying the item's transformation. We can see this by using the **`itemconfigure`** command to look at them.

```
.pad itemconfigure rect1 -place             => "-50 50 1"
.pad itemconfigure rect2 -place             => "50 50 .5"
```

But we can use relative coordinate frames to do this. Pad++ maintains a stack of coordinate frames. Each coordinate frame is a bounding box on the Pad++ surface. All coordinates are specified as a unit square within this coordinate frame, i.e., (0, 0)-(1, 1). That is, when a coordinate frame is specified, coordinates are no longer absolute units, and instead, are relative to the specific frame. Coordinate frames may be specified by an item, or as any bounding box. Note that pen width and minsize and maxsize are also relative to the coordinate frame. In these cases, a value of 1 refers to the average of the width and height of the frame.

This example draws a box with four boxes inside of it, and four boxes inside of each of those, and so on, four levels deep. (Note that from now on, the examples assume a pad widget called .pad has already been created with pixels as the current units.)

```
proc draw_a_box {x1 y1 x2 y2 level} {
   set id [.pad create rectangle $x1 $y1 $x2 $y2 -penwidth .01]
   puts $id
   .pad pushcoordframe $id
   draw_nested_boxes [expr $level + 1]
   .pad popcoordframe
}

proc draw_nested_boxes {level} {
   if {$level >= 5} {return}
   draw_a_box .1  .1  .45 .45 $level   ;# Draw lower-left box
   draw_a_box .55 .1   .9 .45 $level   ;# Draw lower-right box
   draw_a_box .1  .55 .45  .9 $level   ;# Draw upper-left box
   draw_a_box .55 .55  .9  .9 $level   ;# Draw upper-right box
}

draw_a_box 0 0 300 300 1
```



See the **pushcoordframe**, **popcoordframe**, and **resetcoordframe** commands for more information.

TRANSFORMATIONS

Pad++ maintains a distinction between the surface and the view. All graphical items sit at a distinct location on the surface with a given size. The view shows any given location on the surface at any magnification.

Initially the view onto the Pad++ surface looks at the origin (0, 0) with a magnification of 1.0. The view is always represented by a list of 3 numbers representing the (x, y) position and magnification, respectively. The point specifying the view is always rendered at the center of the window. It is possible to adjust the view onto the surface by using the **moveto** widget command. The **getview** command returns the current view. Individual items may be moved or scaled using the widget commands **slide** and **scale**, respectively. Currently, rotation is not supported for either individual items, or the view.

Every item on the Pad++ surface has an anchor and an anchor point associated with it that controls the item's

```
.pad itemconfigure 3 -penwidth
```

returns the current penwidth of item #3. If no options are specified, then a list of all the options and values are returned. This is a good way to find out what options are available for a specific item type. Note that **`ic`** is an alias for **`itemconfigure`**, so an equivalent command is:

```
.pad ic 3 -penwidth
```

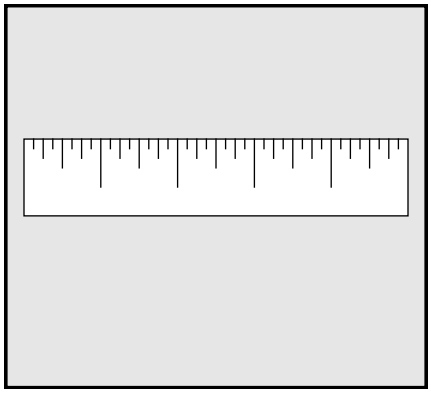
COORDINATES

All coordinates relating to the Pad++ surface are stored as floating-point numbers. Coordinates are specified and are returned in the current Pad++ units, and defaults to pixels (see `-units` configuration option). All coordinates refer to the Pad++ surface and are independent of the current view. Therefore if the view happens to have a magnification of 2.0, and you create an item that is 50 pixels wide, it will appear 100 pixels wide for this specific view.

Larger y-coordinates refer to points higher on the screen; larger x-coordinates refer to points farther to the right. Notice that the y coordinate is inverted as compared to the Tk canvas.

The following code draws a simple ruler:

```
pad .pad -units inches
pack .pad
.pad create rectangle 0 0 5 1 -fill white -penwidth .05
for {set x 0} {$x <= 5} {set x [expr $x + 0.125]} {
   if {[expr int($x) == $x]} {
                     # Draw inch markers
         .pad create line $x .5 $x 1 -penwidth .04
   } elseif {[expr int($x / .5) * .5 == $x]} {
                     # Draw half-inch markers
         .pad create line $x .65 $x 1 -penwidth .04
   } elseif {[expr int($x / .25) * .25 == $x]} {
                     # Draw quarter-inch markers
         .pad create line $x .8 $x 1 -penwidth .04
   } else {
                     # Draw eighth-inch markers
         .pad create line $x .9 $x 1 -penwidth .04
   }
}
.pad config -units pixels   ;# Return to default units
```



Suppose you wanted to create a layout of nested boxes. These coordinates wouldn't be quite as easy to compute.

padwish. This can be done automatically by running the "*pad*" script.

The environment variables to set are `TCL_LIBRARY` and `TK_LIBRARY` (which point to the Tcl/Tk run-time libraries), and `PADHOME` (which points to the Pad++ run-time library and the PadDraw application). Looking at the "*pad*" script will show you what to set these environment variables to.

Once you've run *padwish*, the Pad++ windowing shell, you can start writing your own applications, or you can run PadDraw by typing in the interpreter:

```
source $env(PADHOME)/draw/pad.tcl
```

ITEM IDS AND TAGS

Items on a Pad++ widget may be named in either of two ways: by *id* or by *tag*. Each item has a unique identifying number which is assigned to that item when it is created. The id of an item never changes and id numbers are never re-used within the lifetime of a pad widget. The surface itself always gets an id of '1'. (It is sometimes useful to manually set the id of an item. This is possible with the **`setid`** command.)

Each item may also have any number of tags associated with it. A tag is just a string of characters, and it may take any form except that of an integer. For example, "x123" is OK but "123" isn't. The same tag may be associated with many different items. This is commonly done to group items in various interesting ways; for example, all items associated with one user might have a tag with that user's name. Note that Pad++ also has a special **`group`** item for creating hierarchical groups.

The tag *all* is implicitly associated with every item on the Pad++ widget; it may be used to invoke operations on all the items in the pad.

The tag *current* is managed automatically. It applies to the current item, which is the topmost item whose drawn area covers the position of the mouse cursor. Note that events only go to the *current* item. Since an item gets the *current* tag only if the cursor is over the drawn area, this means that an item receives events only when the cursor is over the drawn area. For example, a rectangle with no fill color will not respond to events when the cursor is over the undrawn interior. If the mouse is not in the Pad++ widget or is not over an item, then no item has the current tag. Portal items are treated somewhat differently, however, as described in the **`bind`** command and in the description of *Portal Items* in the reference manual.

When specifying items in Pad++ widget commands, if the specifier is an integer then it is assumed to refer to the single item with that id. If the specifier is not an integer, then it is assumed to refer to all of the items on the Pad++ widget that have a tag matching that specifier. The symbol *tagOrId* is used to indicate that an argument specifies either an id that selects a single item or a tag that selects zero or more items. Some widget commands only operate on a single item at a time; if tagOrId is specified in a way that names multiple items and the command only operates on a single item, then the normal behavior is for the command to use the first (lowest) of these items in the display list that is suitable for the command. Exceptions are noted in the widget command descriptions.

There are commands to find what the tags of a specific item are (**`gettags`**), and to find all the items that share a tag (**`find`**). Tags can be added to items with the **`addtag`** command or deleted with the **`deletetag`** command, and the **`hastag`** command determines if an item has a particular tag.

ITEM CONFIGURATION

In addition to the commands that are available for manipulating items, every item also has several options that may be configured with the **`itemconfigure`** command. Certain options are shared by all item types (for example, see *-transparency* and *-position*), while some options are specific to item types (like *-font*). Multiple options can be changed with a single itemconfigure command by alternating options and values. If the value of the last option is not specified, then the current value of that option is returned. Thus

```
.pad itemconfigure foo -maxsize 50 -transparency 0.5
```

sets the maximum size of all items with the tag *foo* to 50, and makes them all transparent.

Pad++ Programmer's Guide (Version 0.9)

URL: http://www.cs.unm.edu/pad++

KEYWORDS

Pad, Pad++, Widget, Zoom, Graphical User Interface, Multi-Scale, Zoomable Interfaces, Tcl, Tk.

INTRODUCTION

Pad++ is a structured graphics widget for Tcl/Tk based on zooming. It adds scale as a first class parameter to all items, as well as mechanisms for navigation through the multi-scale space of the Pad++ widget. It has special mechanisms to maintain efficiency for large numbers of graphical items, and also supports special item types such as HTML and portals.

Pad++ is connected to Tcl, a scripting language that provides a high-level interface to the complex graphics and interactions available. While the scripting language runs slowly, it is used as a glue language for rapidly creating interfaces and putting them together. The actual interaction and rendering is performed by the Pad++ substrate (written in C++). This approach allows people to develop applications for Pad++ at a high level while avoiding the complexities inherent in this type of system. Pad++ also supports Scheme as an alternative to Tcl.

Pad++ provides an alternative to the Canvas widget in Tk. While it does not offer everything that the Canvas does, Pad++ offers several extra features, and is designed with the Canvas spirit in mind, maintaining similar syntax for interacting with it.

Pad++ widgets implement structured, multi-scale graphics. It displays any number of items, which may be things like rectangles, lines, text, and images. Items are manipulated (e.g. moved or re-colored) and commands may be associated with items in much the same way that the Tk **`bind`** command allows commands to be bound to widgets. For example, a particular command may be associated with the <Button-1> event so that the command is invoked whenever button 1 is pressed with the mouse cursor over an item. This means that items in a pad can have behaviors defined by the Tcl scripts bound to them.

Note that change bars appear wherever this document differs from the previous version.

AVAILABILITY

Pad++ is Free Access Software. It is not public-domain, but is available for free for education, research, and non-commercial use. You must obtain a Free Access License to use Pad++.

As a Free Access Licensee, you have the right to use the Pad++ System as long as it is not combined with any product or service in any way. Use of the Pad++ System with commercially acquired software that depends on the Pad++ System in any way requires the commercial software supplier to have negotiated a Distribution License with the Pad++ Consortium.

See the files *License* and *LicenseTerms* for more information.

RUNNING PAD++

To run a sample Pad++ application, simply type "*paddraw*" (assuming Pad++ has been properly installed.) This runs a demo application, *PadDraw*. PadDraw shows off many of Pad++'s abilities. It is written entirely in Tcl, and looking at its code is a good way to learn how to create Pad++ applications. There is currently no documention for PadDraw.

Running PadDraw in this fashion does not give access to the Tcl interpreter. This is because the "*paddraw*" program is actually a shell script that runs the Pad++ executable (which is named "*padwish*"), and then loads the Tcl files to run PadDraw. To access the Tcl interpreter, you must set a few environment variables, and then run