Pad++ Programmer's Guide (Version 0.2.7)

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 that features 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. Pad++ provides an alternative to the Canvas widget in Tk. The syntax used to interact with a Pad++ widget is similar to the syntax used to interact with a Tk Canvas. Pad++ has special mechanisms to maintain efficiency for large numbers of graphical items, and also supports special item types such as HTML and portals. While it does not offer all of the features of a Canvas, Pad++ offers several extras.

Pad++ widgets implement structured, multi-scale graphics such as rectangles, lines, text, and images. These items can be manipulated (e.g. moved or re-colored) and/or associated with commands using event bindings. This is similar to the way that the Tk **`bind`** command uses events to associate commands with widgets. For example, a particular command/script may be associated with the <Button-1> event. This command/script is then invoked whenever button 1 is pressed with the mouse cursor over a particular item (or items).

Pad++ is connected to the Tcl scripting language. This provides a high-level interface to its complex graphics and interactions. While Tcl runs slowly, it is used as a glue language for rapidly creating interfaces and putting them together. The actual interaction and rendering on Pad++ widgets are 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.

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 it 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 Pad++, simply type "*pad*" (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 "*pad*" 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 *padwish*.

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. If the *padwish* executable you are using was built with Scheme, then you must also set the ELK_LOADPATH environment variable to point to the Elk runtime library.

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
```

STARTUP FILES

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 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.

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. It includes several demos including a dynamic tree-based Web browser and several lenses. PadDraw supplies basic drawing and zooming capabilities with lenses and semantic zooming that will allow you to create mockups of an application or visualization without any programming to motivate futher development.

PadDraw is contained in the draw subdirectory of the Pad++ distribution. It is intended to be both a user level sample application for experimenting with the Pad++ widget and 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.
- *misc.tcl* Contains many utility procedures used by the rest of PadDraw that may be useful.
- *draw.tcl* Contains the code that makes the Drawing Tools palette.
- *debugevent.tcl* Contains the code that makes the GUI event debugger.

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 windows 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, you could redefine the color palette, or the default font size of HTML pages here.

A PRIMER FOR PROGRAMMING PAD++ IN TCL/TK

The following sections provide a basic tutorial for programming Pad++ with Tcl/Tk. This is not a complete description of all of the Pad++ programming interface, but rather a description of many commonly used features. See the reference documentation for a definitive description.

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 (its id), 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. Then, all of those items can be modified by referring to that tag. 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 be a tag, and refers to all of the items on the Pad++ widget that have that tag. The symbol *tagOrId* is used to indicate that a tag or an id is to be used as an argument to the command. For example, the manual entry for the command **`slide`** looks like this: *pathName* **`slide`** *tagOrId*. An id selects a single item (which may be a group item), and a tag selects zero or more items to be used by the command. Some widget commands only operate on a single item at a time. If *tagOrId* specifies multiple items, 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 the specific tags associated with an item (**`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. In addition, tags can be accessed with the *-tags* itemconfigure option.

ITEM CONFIGURATION

Every item has several options that may be configured with the **`itemconfigure`** command. Certain options are shared by all item types (for example, see *-transparency* and *-maxsize*), 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 -sticky 1
```

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

```
.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

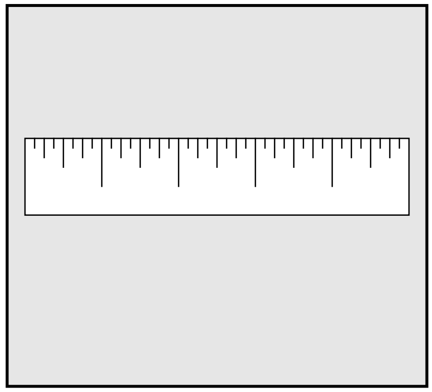
Coordinates are specified and are returned in the current Pad++ units. The default units are pixels (see `-units` configuration option). All coordinates relating to the Pad++ surface are stored as floating-point numbers. All coordinates refer to the Pad++ surface and are independent of the current view.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

The origin of the Pad++ coordinate system is in the center of the window upon startup. As with a normal cartesian coordinate system, 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.

There are two parts to pad's coordinate system. First, every item on the pad surface exists at a specific place, and can be moved and scaled on the surface. In addition, there is a view which specifies what portion of the pad surface is visible and at what magnification. Thus, the appearance of an item on the screen depends both on the position and scale of the item (the item's transformation) as well as the location and magnification of the current view.

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
```



Views are defined by a list of three numbers which specify the (x, y) point at the center of the window, and the magnification of the view, respectively. The default view, (0, 0, 1), has the origin at the center of the screen and is not magnified. The getview command returns the current view, and the moveto command modifies the current view with an optional animation.

```
.pad getview
=> 0 0 1
        # This command makes the ruler appear twice as big
.pad moveto 0 0 2
        # This command smoothly animates the view back to its
        # starting point, and takes 1000 milliseconds for the animation.
.pad moveto 0 0 1 1000
```

Coordinates can also be managed in another way. Sometimes it is useful to specify coordinates relative to something else, instead of with the absolute coordinate system just described. Pad++ maintains a stack of coordinate frames. Suppose you wanted to create a layout of nested boxes. These coordinates would not be quite as easy to compute if we remain in the original coordinate system. But we can use relative coordinate frames to do this. 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. Instead they are relative to the specified frame. Coordinate frames may be specified by any bounding box, or by an item (thereby using the item's 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.

The following example draws a box with four boxes inside of it, and four boxes inside of each of those, and so on, four levels deep. New coordinate frames are placed on the top of the coordinate frame stack using the **pushcoordframe** command and removed from the top of the stack with the **popcoordframe** command. Note that from now on, the examples assume a pad widget called .pad has already been created with pixels as the current units, and packed with:

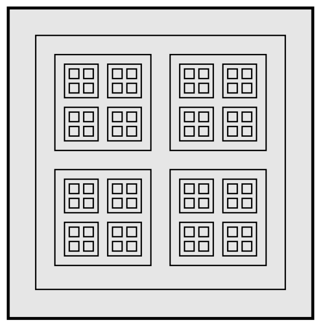
```
pad .pad
pack .pad


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 view is specified by the point currently 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, relative to the *surface*, using the widget commands **slide** and **scale**, respectively. Currently, rotation is not supported for either individual items, or the view.

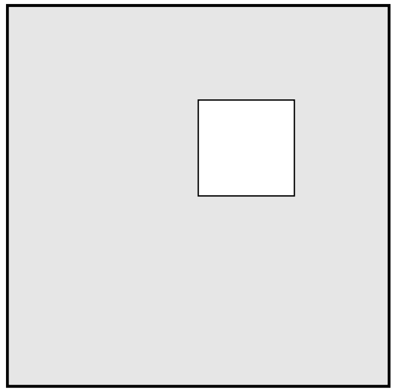
Every item on the Pad++ surface has an anchor and an anchor point associated with it that controls the item's position and size. The anchor and the position of the anchor point can be accessed directly with the `-anchor`, and `-place` itemconfigure options. The `-place` consists of a list of three values where the first two values specify the position of the anchor point of the item, and the third value specifies its size. For example, an image with a `-place` 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 -place of (0, 0, 1). Items with coordinates, however, (lines, rectangles, polygons, splines, and portals) are special in that these items get created with a -place that is determined by the coordinates. So, creating a rectangle from (0, 0) to (50, 50) with a center anchor gets a -place of "25 25 1".

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

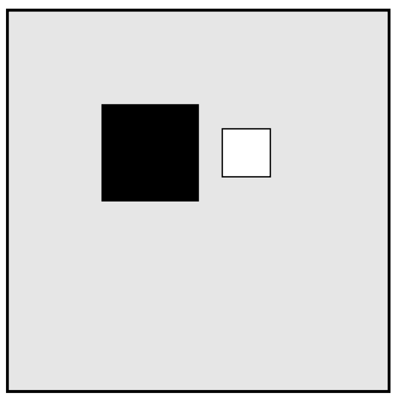
```
.pad create rectangle 0 0 100 100 -tags "rect1" -fill black
```

```
.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
```

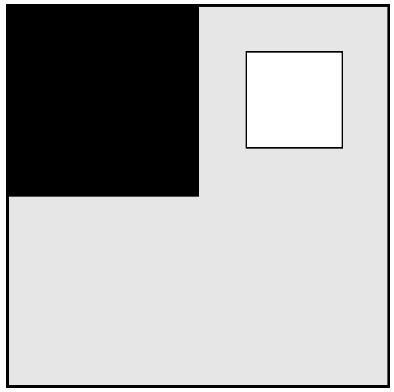


These commands operate by modifying the item's configuration. We can see this by using the **`itemconfigure`** command to look at them.

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

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
```



Note that the item configurations do not change when the view is changed:

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

We can retrieve 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++ offers some unique features for the **bind** command. See the Pad++ Reference Manual for details).

```
.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}
```

FONTS

Pad++ supports Adobe Type1 fonts, however, this font support is still under development. You may notice that fonts are not rendered very well when they are small, but look good when they are large. While these fonts are rendered relatively quickly, they are expensive. The system will slow down when a lot of text is used with one of these fonts. Fonts for text items are specified with the *-font* itemconfigure option. The system font is specified with "System", and Type1 fonts are specified by the filename that contains the font. This will be changed in a future version to allow a system-independent way of specifying fonts.

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

IMAGES

Pad++ supports real-time zooming of images. The rendering speed is dependent only on the size of the resulting image and 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 same machine that is running Pad++ (i.e., real-time zooming is much slower across networks). Pad++ will print the message "Real-time image zooming supported", when starting up, if real-time image zooming is available.

TREES

Pad++ supports dynamic hierarchical trees of pad objects which animate to show a focus + context information space. Dynamic trees are currently utilized in the Web browser utility, found in the Demos menu of PadDraw.

A dynamic tree is a hierarchical information space which is managed by layout objects which control the relative positioning of the pad items they are associated with. Currently, pad++ supports one default type of layout - a hierarchical tree layout which places child nodes to the right of the node.

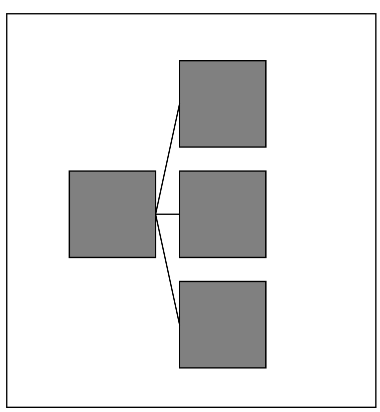
Each pad has a treeroot slot, which is the master root of all dynamic trees for that pad. It is possible to create special children of this master root node are "invisible" roots - nodes which have pad objects associated with them, but are not used for anything except for managing their subtrees. These nodes separate the dynamic information space into managed subtrees that do not "know" about each other. That is, these separate subtrees may interfere when a layout occurs, because they are not connected by a functioning layout. The only way to get these top level nodes to "cooperate" is to call layout on the master tree node owned by the pad.

Creating a tree

First, we will make some simple rectangles to represent the nodes of a tree. Then the following code creates a tree attached to the master root, with one parent and three children, and lays out the tree.

```
set root [.pad create rectangle 0 0 50 50 -fill white]
set child1 [.pad create rectangle 0 0 50 50 -fill red]
set child2 [.pad create rectangle 0 0 50 50 -fill green]
set child3 [.pad create rectangle 0 0 50 50 -fill blue]

        # Create the dynamic tree data nodes
.pad tree create $root
.pad tree create $child1
.pad tree create $child2
.pad tree create $child3
        # Create the tree structure
.pad tree addnode $child1 $root
.pad tree addnode $child2 $root
.pad tree addnode $child3 $root
        # Layout the tree
.pad tree layout $root
```



The tree structure can be accessed with the **`getparent`** and **`getchildren`** commands. These return the ids of the appropriate members of the hierarchy.

```
.pad tree getparent $child1
  => Returns $root
.pad tree getchildren $root
  => Returns a list of child1 child2 child3
.pad tree getchildren $child1
  => Returns an empty string
```

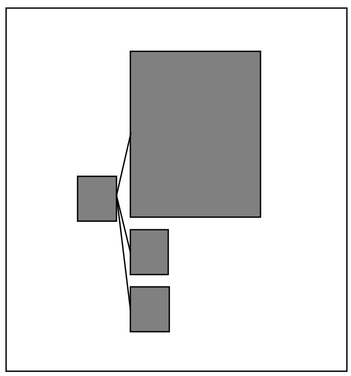
Layout

The layout at a node is invoked by the following command which causes a layout of all nodes in the subtree rooted at node.

```
.pad tree layout $root
```

The tree maintains a notion of focus. By default, no nodes have focus, and all nodes are the same size. However, making a node get focus will increase it's size (by an amount specified by **`setfocusmag`**). Executing the following code will warp the tree layout as shown:

```
.pad tree setfocus $child1 1.0
.pad tree layout $root
```



Normally, when invoking the **`layout`** command on a tree, the tree will animate without changing the view. The -*view* option in the **`layout`** command is used to animate the view while a layout is taking place. Because the user may want to animate the view to a position based on the future position of a node, the **`getlayoutbbox`** provides the bounding box of a node at the end of the current animation. These commands must be used in a specific order, so that the information referenced by both the layout code and the user is valid.

To animate the view of a layout to center child1, use the following commands.

```
.pad tree computelayout $root
set bb [.pad tree getlayoutbbox $child1]
set x [lindex $bb 0]
set y [lindex $bb 0]
set z [lindex [.pad getview] 2]
.pad tree animatelayout $root -view "$x $y $z"
```

Maintainance

A tree can be reorganized, added to, or parts of a tree can be deleted. A single node can be reparented, or its entire subtree. The relevant commands are **`delete`**, **`removenode`**, and **`reparent`**. The following code will rearrange the previously created tree, moving one child under another.

```
.pad tree reparent $child3 $child2
.pad tree layout $root
```

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. 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 (written in Tcl), defines several widgets including buttons and sliders 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. In addition, any other parameters before the first option (defined by a '-') are passed as extra arguments after the pad widget pathname. 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 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 args} {
    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" -z 20]

    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} {
                   $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"
```

DISPLAY LISTS

There are two display lists maintained by Pad++, the regular display list and the *sticky* display list. Items are put on the regular display list by default. The actual size and location of the items on the screen depends upon the view. Items put on the *sticky* display list are unaffected by the view and thus seem to stick to the screen. Sticky items act as if the view is always (0, 0) with a magnification of 1.0. Items can be moved between the display lists with the `-sticky` itemconfiguration option.

Each display list maintains its own sense of drawing order. Items that are created later are drawn later, and thus appear over other items. All sticky items are rendered after (i.e., on top of) non-sticky items. Drawing order within display lists can be changed with the **`raise`** and **`lower`** commands.

REGION MANAGEMENT AND SCREEN UPDATING

Only the portions of the screen that change get rendered. This makes many operations, such as modifying and dragging items, or panning the view, much more efficient. Panning is optimized by shifting the pixels in the appropriate direction and then re-rendering just the strips left blank. This optimization can be turned off with the -*fastPan* widget configuration option. Zooming speed is not improved by this process because it requires re-rendering the entire view.

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 occur until either the system is idle, or the **`update`** command is called. Many commands implicitly damage items, but damage can be triggered manually with the **`damage`** command.

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++ 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) is applied to an item, an associated command is executed. This system is similar to the way that the Tk **`bind`** command uses events to associate commands with widgets; but the Pad++ **`bind`** command has several significant additions. The extensions fall into the following three categories (See the pad **`bind`** command in the Pad++ Reference Manual for complete details):

  - 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 (the relevant size thresholds are definable). This is a simple way of making "semantically zoomable" items (i.e., items that look different when they 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. The user-defined types 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, the `-place` **itemconfigure** option, or the **slide** command. 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 quicker in the middle, and ends slowly. This results in smoother feeling animations. *slow-in-slow-out* does not affect the amount of time that 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 in the speed of the motion.

EFFICIENCY

Pad++ uses several techniques to maintain efficiency, even when handeling a large number of items. Understanding these internal techniques may help the application designer to build faster programs. The techniques are:

- *Spatial Indexing and Clustering*

  The items are stored in a hierarchy based upon the bounding box of each item. In effect, it is a kind of spatial indexing. This hierarchy is used to quickly locate the item(s) when the Pad++ widget is rendered, or when an item must be found to match an event, based on its position on the Pad++ surface. The hierarchy is defined by the invariant that every item whose bounding box is completely enclosed by another item's bounding box becomes that item's child. Pad++ automatically performs cluster analysis and adds invisible *wrapper* nodes around groups of items in order to keep down the fan-out of this hierarchy. These wrappers are managed automatically and are invisible in all respects to the application developer.

• *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 by drawing larger items at lower resolution. The system refines the scene to it highest resolution, in steps, after a short pause, once the interaction is over.

These iterative refinements each have numbers. Refinement level 0 (sometimes referred to as render level 0) is the lowest-resolution, and thus the fastest. 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 undithered at low levels of refinement, and dithered at higher refinement levels. Note that it is possible to stop images from ever being dithered with the *-norgb* option on the **`allocimage`** command. 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`**, **`getmag`**, and **`getlevel`** 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. This flow is independent of processor speed, scene complexity, or window size. For example, if a particular 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 give control back to the user. For example, pressing the space-bar during an animation will bring you to the end of the animation. Panning during a refinement will stop the refinement and perform high-speed panning. Interruption can be turned off with the `-interruptible` configuration option.

SCRIPTING LANGUAGES

Pad++ is a prototyping system. It 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 Pad++ 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)")
> (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, modifying the Pad++ C++ substrate to access that code, and building a new padwish executable. Several callback procedures must be defined to add a language. 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.* This routine gets called whenever a new Pad++ widget is created. This leads to the creation of a function for accessing the new widget in the new scripting language.

```
Pad_CreateProc   *create_proc;
```

- *Callback to process pad command.* There should be a command for accessing the new scripting language from Tcl, 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 the Tcl command is evaluated, and it must implement this functionality.

```
Pad_CommandProc  *command_proc;
```

- *Callback that specifies if command is complete and should be evaluated.* Given a command in the scripting language, this routine 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 a command is, or is not complete (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.

LOGO

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

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 an ARPA 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, Mohamad Ijadi, David Rogers, David Proft, Jason Stewart, and people at NYU: Ken Perlin, and Jon Meyer.

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

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

BUGS

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++. This is currently the best way to contact other pad users.