Pad++

P++ Reference Manual

Motorola PX 1006_2

Pad++ Reference Manual

(Version 0.2.7)

Introduction

This reference manual describes the complete Tcl API to Pad++. It describes how to create and modify a pad widget, and all the commands associated with a pad widget that allow you to create and modify items, attach event bindings to them, navigate within the pad widget, etc.

This document in organized into the following sections:

- Padwish synopsis
- Tcl synopsis
- Widget-Specific Options
- Widget Commands
- Overview of Item Types
- Default Bindings
- Global TCL Variables
- KPL-Pad++ Interface

Each section contains all the relevant entries in alphabetical order. Related commands and options are also grouped together here to show which commands are related. Every command and itemconfigure option are listed.

Related Commands and Options

Items

`allocimage` [8] Allocate data for an image item

`create` [17] Create new items

`delete` [19] Delete existing items

`find`[27] Search for items by various keys

`freeimage` [31] Free data from an image item

`itemconfigure`[47] Configure existing items

`lower`[49] Push an item lower in the drawing order

`pick`[53] Find the item under a point

`popcoordframesc`[54] Pop a relative coordinate frame off of the stack

`pushcoordframe`[56] Add a new relative coordinate frame to the stack

`raise`[57] Bring an item higher in the drawing order

`resetcoordrame`[61] Reset coordinate frame stack to empty

`setid`[67] Change the id of an item

`text`[76] Modify text item

`type` [78] Get the type of an item

`-arrow` [78] Some items only: Whether to draw arrow heads with this item

`-arrowshape`[52] Some items only: The shape of drawn arrow heads

`-dither` [47] Some items only: Render with dithering

`-file`[87] Some items only: File an item should be defined by

`-height`[6] Height of an item. Normally computed automatically, but can be set

`-html` [43] Some items only: The HTML item associated with an htmlanchor

`-htmlanchors`[38] Some items only: The anchors associated with an HTML page

`-image`[48] Some items only: Image data associated with item (allocated by allocimage)

`-info`[7] A place to store application-specific information with an item

`-ismap`[44] Some items only: True if an htmlanchor is an image map

`-lock`[9] Locks an item so it can not be modified or deleted

`-state`[45] Some items only: State of an item (such as visited, unvisited, etc.)

`-sticky` [14] Specifies if an item should stay put when the view changes

`-title`[70] Some items only: Title of an item

`-url`[41] Some items only: The URL associated with an item

`-width`[21] Width of an item. Normally computed automatically, but can be set

`-zoomaction`[24] A script that gets evaluated when an item is scaled larger or smaller than a set size

Item Transformations

`bbox` [9] Get the bounding box of an item `coords`[16] Change the coordinates of an item

`getsize` [39] Get the size of an item (possibly within portals)

`scale`[62] Change the size of an item relatively

`slide`[74] Move an item relatively in (x, y)

`-anchor`[2] The part of the item that -place refers to

`-place`[12] Transformation of an item - Translation (x, y), and magnification (z)

`-x`[22] X componenent of -place transformation

`-y`[23] Y componenent of -place transformation

`-z`[25] Z componenent of -place transformation

View Transformations

`center`[12] Change the view so as to center an item

`centerbbox`[13] Change the view so as to center a bounding box

`getview`[42] Get the current view (possibly within portals)

`moveto`[50] Change the view (possibly within portals)

`zoom` [83] Zoom the view around a specified point

`-viewscript` [19] A script that gets evaluated whenever the view is changed

`-view`[71] Some items only: Specifies the view this item sees

`-lookon`[67] Some items only: Specifies the pad widget this item sees

Tags

`addtag` [4] Add a tag to an item

`deletetag`[19] Delete a tag from an item

`dtag` [19] Synonym for deletetag

`gettags`[40] Get the tags an item has

`hastag`[44] Determine if an item has a particular tag

`-tags`[15] List of tags associated with an item

Events

`addmodifier`[2] Add a new user-defined modifier for future use

`bind`[10] Create, modify, access, or delete event bindings bindtags[11] Specify whether events should go to the most-specific or most-general description

deletemodifier[20] Delete a user-defined modifier

focus[28] Set the focus for keyboard events

getmodifier [36] Get the current user-defined modifier

setmodifier[71] Make the specified user-defined modifier the current one

-events [4] True if item receives events, false otherwise

Groups

addgroupmember[1] Add an item to a group

getgroup[33] Get the group an item belongs to

removegroupmember[59] Remove an item from a group

-divisible [26] True if events go through a group to its members

-members[28] The list of members of a group

Layout

grid[43] Layout pad items in a grid as with the Tk grid command

tree [77] Layout pad items with a dynamic graphical-fisheye view tree

Rendering

damage [18] Specify that a group of items needs to be redrawn

update[79] Force any requested render requests to occur immediately

-alwaysrender [1] True if the item must be rendered, even if the system is slow and the item is small

-border[31] Some items only: Specifies border color of item

-borderwidth [32] Some items only: Specifies width of border

-capstyle[53] Some items only: Specifies how to draw line ends

-faderange[5] Range over which an item fades in or out

-fill [29] Some items only: Specifies fill color of item

-font [37] Some items only: Specifies font to use for text

-joinstyle[54] Some items only: Specifies how to draw the joints within multi-point lines

-layer[8] The layer an item is on

-noisedata [55] Some items only: Specifies parameters to render item with noise

-maxsize [10] The maximum size an item is rendered it (absolute or relative to window size)

-minsize [11] The minimum size an item is rendered it (absolute or relative to window size)

-pen [30] Some items only: Specifies pen color of item

-penwidth [57] Some items only: Specifies width of pen

-relief[69] Some items only: Specifies how a border should be rendered

-transparency[18] Transparency of an item. 0 is completely transparent, 1 is completely opaque

-visiblelayers[58] The layers that are visible within this view (just for portals and pad surface, item #1)

File I/O

read[58] Read a .pad file

write[82] Write a .pad file (all the items on a widget)

Miscellaneous

configure[15] Modify the pad widget

info[45] Get type-specific information about an item

islinked [46] Determine if the top-level window that a pad widget is in has been mapped yet

setlanguage[69] Set the language to be used for future calback scripts

settoplevel[72] Set the language to be used by the top-level interpreter

windowshape[81] Modify the shape of the top-level window that a pad widget is in

Utilities

clock[14] Create a clock to measure elapsed milliseconds

getdate [32] Get the current date in unix format

getpads [37] Get a list of all pad widgets currently defined

line2spline[48] Generate points for a spline that approximate a line

noise[51] Generate 'perlin' noise

padxy[52] Convert a window point (x, y) to pad coordinates

spline2line[75] Generate points for a line that approximate a spline

urlfetch[80] Retrieve a URL over the internet in the background

-donescript[34] Some items only: A script to evaluate when a background action has completed

-errorscript [35] Some items only: A script to evaluate when a background action has an error

-updatescript [40] Some items only: A script to evaluate when a background action has made progress

Renderscripts

`allocborder`[6] Allocate a border for future rendering

`alloccolor`[7] Allocate a color for future rendering

`allocimage` [8] Allocate a image for future rendering

`drawimage` [23] Draw an image within a renderscript

`drawline`[24] Draw a line within a renderscript

`drawpolygon` [25] Draw a polygon within a renderscript

`drawtext` [26] Draw text within a renderscript

`freeborder` [29] Free a border previously allocated

`freecolor` [30] Free a color previously allocated

`freeimage`[31] Free an image previously allocated

`getlevel` [34] Get the render level within a renderscript

`getmag`[35] Get the current magnification within a renderscript

`getportals`[38] Get the list of portals being rendered within during a renderscript

`gettextbbox`[41] Get the bounding box of a text string

`renderitem`[60] Render an item in a render callback

`setcapstyle`[63] Specify how the end caps of lines should be drawn

`setfont`[65] Specify the font to be used for renderscript drawing

`setfontheight`[66] Specify the font height to be used for renderscript drawing

`setjoinstyle`[68] Specify how the joints within multi-point lines should be drawn

`setlinewidth`[70] Specify the penwidth of lines when they are drawn

`-renderscript`[13] A script that gets evaluated every time an item is rendered

`-bb` [50] A script that gets evaluated to specify the bounding box of an item

Debugging

`printtree`[55] Print all the items on the pad surface in their internal tree structure

Extensions

`addoption` [3] Create a new option for an existing type

`addtype`[5] Create a new item type

Executables

When Pad++ is built and installed correctly, there are two executable files that may be run. *padwish* runs a version of the Tcl interpreter extended with the pad widget. This is a complete superset of the standard Tk wish program. The `pad` command is the sole addition which is described below. In addition, the Pad++ distribution comes with an application written entirely in Tcl called PadDraw. This application is a general-purpose drawing and demo program that shows many capabilities of the pad widget. PadDraw is started by running the *pad* script which automatically runs *padwish* and starts the Tcl program. When running PadDraw by executing *pad*, the Tcl interpreter is not available.

Padwish Synopsis

```
padwish [options] [arg arg ...]
```

Valid options are:

`-display display`

Display (and screen) on which to display window.

`-geometry geometry`

Initial geometry to use for window.

`-name name`

Use name as the title to be displayed in the window, and as the name of the interpreter for send commands.

`-sync`

Execute all X server commands synchronously, so that errors are reported immediately. This will result in much slower execution, but it is useful for debugging.

`-colormap colormap`

Specifies the colormap that padwish should use. If colormap is "new", then a private colormap is allocated for padwish, so images will look nicer (although on some systems you get a distracting flash when you move the pointer in and out of a PadDraw window and the global colormap is updated).

`-visual visual`

Specifies the visual type that padwish should use. The valid visuals depend on the X server you are running on. Some common useful ones are "truecolor 24" and "truecolor 12", which specify 24 bit and 12 bit mode, respectively.

`-language`

Specifies what scripting language the top-level interpreter should use. Pad++ always supports Tcl, but can be compiled to use the Elk version of Scheme also. In addition, Pad++ provides a mechanism to support other interpreted scripting languages as well. Defaults to 'tcl'.

`-sharedmemory`

Specifies if Pad++ should try and use X shared memory. Some machines (notably a particular Solaris 5.4 machine) crashes and the X server dies when Pad++ is used with shared memory, so it can be disabled if there is trouble. Defaults to 1 (true).

`-help`

Print a summary of the command-line options and exit.

`--`

Pass all remaining arguments through to the script's argv variable without interpreting them. This provides a mechanism for passing arguments such as -name to a script instead of having padwish interpret them.

TCL Synopsis

```
pad [pathName [options]]
```

The `pad` command creates a new window (given by the `pathName` argument) and makes it into a Pad++ widget. If no `pathName` is specified, a unique top-level window name will be generated. Additional options may be specified on the command line or in the option database to configure aspects of the Pad++. The `pad` command returns the name of the created window. At the time this command is invoked, there must not exist a window named `pathName`, but `pathName`'s parent must exist.

Once a Pad++ widget is created, there are five ways of writing Tcl code for it. They are:

- **Configuring the widget:** Each widget has several configuration options that control the widget as a whole. For example, `-width` and `-height` control the geometry of the widget.
- **Executing widget commands:** There are many commands associated with the widget. They are actually sub-commands of the primary widget command. When a new pad widget is created, a command is also created whose name is the name of the widget. For instance, evaluating `pad .pad` creates a widget named `.pad`, and a command named `.pad`. For example, to find out what the current view on the pad widget is, use the `getview` command with: `.pad getview`.
- **Creating items on the widget:** Each pad widget can contain many graphical items, such as lines, text, etc. These are all created with the `create` sub-command. For example, `.pad create line 0 0 10 10` creates a line from the origin to the point (10, 10).
- **Configuring those items:** Once items have been created, they can be modified with the `itemconfigure` sub-command. For example, supposing that the previous line had an id of 2, we could change its pen color and width with: `.pad itemconfigure 2 -pen red -penwidth 5`
- **Accessing global Pad variables:** The pad widget declares certain global Tcl variables that can be used by applications. For example, to see the current version of Pad++, examine the *Pad_Version* variable.

This version of Pad++ works only with Tcl7.5/Tk4.1.

Note that in this reference manual, optional parameters are listed in square brackets, [...]. While this is traditional for reference documentation, the Tcl/Tk documentation uses ?...? to denote optional parameters in order to avoid confusion with the meaning of [...] in the Tcl language. We decided to risk the confusion with Tcl for the increased clarity of square brackets.

Widget-Specific Options

Name: `background`

Class: `Background`

Command-Line Switch: `-background`

Specifies the normal background color to use when displaying the widget.

Example:

```
.pad config -background gray50
```

Name: `closeEnough`

Class: `CloseEnough`

Command-Line Switch: `-closeEnough`

Specifies a floating-point value indicating how close the mouse cursor must be to an item before it is considered to be "on" the item. Defaults to 3.0.

Name: `colorCubeSize`

Class: `ColorCubeSize`

Command-Line Switch: `-colorCubeSize`

Specifies how many colors to allocate for images. Whenever images are displayed, the system tries to allocate colorCubeSize3 colors. For example, if colorCubeSize is 5, then 5*5*5 or 125 colors will allocated. If unsuccessful, smaller color cubes are tried successively. Default is 5.

Name: `cursor`

Class: `Cursor`

Command-Line Switch: `-cursor`

Specifies the mouse cursor to be used for the widget. The value may have any of the forms acceptable to Tk_GetCursor.

Name: `debugBB`

Class: `DebugBB`

Command-Line Switch: `-debugBB`

Turns on and off display of bounding boxes. Default is 0.

Name: `debugEvent`

Class: `DebugEvent`

Command-Line Switch: `-debugEvent`

Turns on and off debugging of events. Default is 0. When event debugging is turned on, pad outputs a description of event handlers as they fire. In addition, if a break or event in a handler stops some events from firing, those events not fired are shown. By default, the event debugging output goes to stdout, however, it can be sent to a Tcl variable with the *-debugOut* configure option. Also note that PadDraw comes with a graphical interface that creates a GUI for seeing and examining events as they fire. This graphical event debugger can be used in other pad applications. See *draw/debugevent.tcl*.

Name: `debugGen`

Class: `DebugGen`

Command-Line Switch: `-debugGen`

Turns on and off general debugging. Default is 0.

Name: `debugOut`

Class: `DebugOut`

Command-Line Switch: `-debugOut`

Controls where debug output goes. By default, debug output is sent to stdout. However, the *-debugOut* configure option can specify a Tcl variable that all debug output will be appended to. It is then possible to set a Tcl trace on that variable to be notified whenever debug output is available. Currently, only *-debugEvent* uses the *-debugOut* variable.

Example: Evaluating "`.pad config -debugOut foo`" will cause all future debug output to be appended to the Tcl variable '`foo`'.

Name: `debugRegion`

Class: `DebugRegion`

Command-Line Switch: `-debugRegion`

Turns on and off visual display of portion of the screen that actually gets re-rendered. Used to debug region management. Default is 0.

Name: `debugstat`

Class: `DebugStat`

Command-Line Switch: `-debugstat`

Turns on and off status line on the Pad (for debugging). Default is 0. The status line shows the total number of items on the pad surface, the number of items checked for rendering, and the number of items actually rendered during the most recent render.

Name: `defaultRenderLevel`

Class: `DefaultRenderLevel`

Command-Line Switch: `-defaultRenderLevel`

Specifies the default render level to use to display the Pad if no specific level is specified. The render level is generally used for efficiency where render level 0 is the fastest and least pretty way to render the pad (text is uglier, smaller items are not rendered, some items are rendered at a lower resolution). As the render level goes higher, the pad is rendered slower and prettier

Name: `desiredFrameRate`

Class: `DesiredFrameRate`

Command-Line Switch: `-desiredFrameRate`

Specifies the desired frame rate (in frames per second). This number is used by the Pad++ rendering engine to decide how to render the scene while animating. If a high frame rate is requested, small objects may not be rendered (see `-alwaysrender`) flag, and some objects may be rendered at low resolution. The default is 20 frames/second.

Name: `dissolveSpeed`

Class: `DissolveSpeed`

Command-Line Switch: `-dissolveSpeed`

Specifies how quickly dissolves should occur upon refinement. When the pad widget refines, it uses a dissolve effect instead of a simple buffer swap. The dissolve is controlled by *-dissolveSpeed*. This option may vary between 0 and 3 where 0 is a simple buffer swap, 1 is a fast dissolve, and 3 is the slowest dissolve. The default is 2.

Name: `doubleBuffer`

Class: `DoubleBuffer`

Command-Line Switch: `-doubleBuffer`

Specifies if the system should use double buffering for rendering. If doubleBuffer is set to 0 (off), rendering will be a little faster, but the screen will flash quite a bit. Mostly useful for debugging. Default is 1.

Name: `enableOpaque`

Class: `EnableOpaque`

Command-Line Switch: `-enableOpaque`

Normally, objects which are completely behind opaque objects are not rendered. Turn this flag off to turn off this efficiency method. Default is 1.

Name: `fastPan`

Class: `FastPan`

Command-Line Switch: `-fastPan`

Pad++ normally does fast pans, i.e., copying the portion of the screen that doesn't change, and re-rendering the new portion. This results in an approximation which can make the view be off by up to a half of a pixel. Fast panning can be disabled by setting this flag to 0 which results in slower but more accurate pans. Default is 1.

Name: `fontCacheSize`

Class: `fontCacheSize`

Command-Line Switch: `-fontCacheSize`

Pad++ employs a simple caching mechanism when drawing text in Type1 fonts. The caching mechanism remembers what size, font and bitmap it used when it last drew a particular character, and if that character is drawn again at the same size and font, Pad++ reuses the last bitmap image for that character rather than generating the bitmap for the character from its outline description. This greatly increases the speed of rendering large quantities of text.

You can configure the caching mechanism using the *-fontCacheSize* option. The font cache size is measured in Kilobytes (rounded to the nearest 100K). Setting *-fontCacheSize* to 0 turns off font caching, and characters are always drawn from their outline descriptions. The default value is 100 which produces significantly faster font rendering than using no font cache. Values above 100 have a lesser impact on performance, but may be effective for applications which use a lot of text with different fonts and sizes.

Name: `gamma`

Class: `Gamma`

Command-Line Switch: `-gamma`

Specifies 'gamma' used for allocating colors for images. This number controls how light or dark an image appears to be. Larger numbers will make images appear lighter. Default is 1.0.

Name: `height`

Class: `Height`

Command-Line Switch: `-height`

Specifies the height of the Pad in pixels. Defaults to 400.

Name: `heightmmofscreen`

Class: `HeightMMOfScreen`

Command-Line Switch: `-heightmmofscreen`

Specifies the height of the physical screen in millimeters. Normally, this information is given by the X server, but sometimes it is incorrect (for example, on some laptops). If it is incorrect, coordinates on the Pad++ surface will be incorrect. If this value is set to 0, the X server information will be used. Defaults to 0.

Name: `interruptible`

Class: `interruptible`

Command-Line Switch: `-interruptible`

If this flag is true (1), then animations and slow renders will be interrupted by events (mouse and keyboard). Defaults to true (1).

Name: `maxZoom`

Class: `MaxZoom`

Command-Line Switch: `-maxzoom`

This controls the maximum zoom (in and out) that any view is allowed. This way, it not possible to crash pad by zooming in or out too far. It defaults to 100,000,000 which gives 16 orders of magnitude of zooming (8 in and 8 out). Note that the amount one can zoom in is determined by the product of the (x, y) position and the zoom. So, while you can zoom into the position (0, 0, 100000000), you can only zoom into (1000, 1000, 100000). Setting *-maxzoom* to 0 disables the checking.

Name: `refinementDelay`

Class: `RefinementDelay`

Command-Line Switch: `-refinementDelay`

Specifies the delay in milliseconds after the last X event to start refinement. Default is 1000.

Name: `sync`

Class: `Sync`

Command-Line Switch: `-sync`

Specifies if X event synchronization should be turned on. When it is on, the X server executes every command as it is executed rather than caching them and executing commands in groups. Generally useful just for debugging. Default is 0.

Name: `units`

Class: `Units`

Command-Line Switch: `-units`

Specifies unit dimensions for all coordinates used by Pad++. It can be any of "points", "mm", "inches", or "pixels". Default is points.

Name: `width`

Class: `Width`

Command-Line Switch: `-width`

Specifies the width of the Pad in pixels. Defaults to 400.

Name: `widthmmofscreen`

Class: `WidthMMOfScreen`

Command-Line Switch: -`widthmmofscreen`

Specifies the width of the physical screen in millimeters. Normally, this information is given by the X server, but sometimes it is incorrect (for example, some laptops). If it is incorrect, coordinates on the Pad++ surface will be incorrect. If this value is set to 0, the X server information will be used. Defaults to 0.

Widget Commands

The `pad` command creates a new Tcl command whose name is `pathName`. This command may be used to invoke various operations on the widget. It has the following general form:

pathName option [arg arg ...]

Option and the *args* determine the exact behavior of the command. The following widget commands are possible for Pad++ widgets:

[1] `pathName addgroupmember [-notransform] tagOrId groupTagOrId`

Add all items specified by *tagOrId* to the group specified by *groupTagOrId*. If *groupTagOrId* specifies more than one item, the first one is used. The items are added to the end of the group in the order specified by *tagOrId*. Groups automatically update their bounding boxes to enclose all of their members. Thus, they will grow and shrink as their members change.

By default, items are transformed so they don't change their location when added to a group, even if the group has a transformation. This is implemented by transforming the item's transformation to be the inverse of the group's transformation. If the *-notransform* flag is specified, this inverse transformation is not applied, and the item will move by the group's transformation when added. (Also see the `removegroupmember`, and `getgroup` commands). Returns an empty string.

Example :

set id0 [.pad create line 0 0 100 100]

254

set id1 [.pad create line -10 20 80 -60]

255

set gid [.pad create group -members "$id0 $id1"]

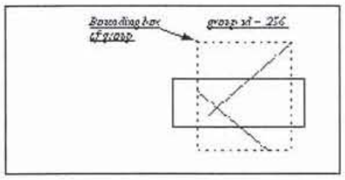
256

.pad ic $gid -members

254 255

set id3 [.pad create rectangle -20 -20 130 40]

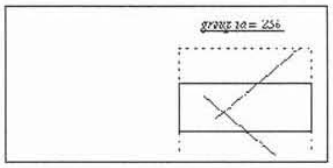
266



.pad addgroupmember $id3 $gid

.pad ic $gid -members

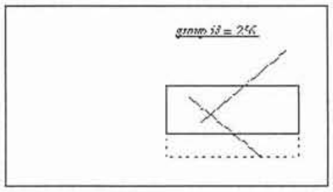
254 255 266



.pad removegroupmember $id0 $gid

.pad ic $gid -members

255 266



.pad getgroup $id2

256

[2] `pathName addmodifier modifier`

Define *modifier* to be a user-defined modifier that can be used in future event bindings. (Also see the `deletemodifier`, `setmodifier`, `getmodifier`, and `bind` commands).

[3] pathName `addoption` [-nowrite] typename optionname optionscript default

Add a new option (named optionname) to all objects of type typename. typename must either be a built-in type, a user-defined type previously defined by `addtype`, or the special word "all" which means that this option applies to all types.

When optionscript is called, the following arguments will be added on to the end of the script:

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.

optionscript must return the current (or new) value of the option. default specifies the default value of this option. This is used to determine if the option should be written out when the `write` command is executed. Note that the option will only be written out if the value is different than the default. If -nowrite is specified, then this option won't be written out. See the section APPLICATION-DEFINED ITEM TYPES AND OPTIONS in the Programmer's Guide for more information. (Also see the `addtype` command.)

[4] `pathName addtag tagToAdd tagOrId ...`

For each item specified by the list of tagOrIds, add *tagToAdd* to the list of tags associated with the item if it isn't already present on that list. It is possible that no items will be specified by tagOrId, in which case the command has no effect. This command returns an empty string.

This command is designed to be used in conjunction with the **find** command. Notice the necessity of using **eval** in this example:
```
eval .pad addtag foo [.pad find withtag bar]
```

[5] pathName `addtype` typename createscript

Add typename to the list of allowed user defined types. When a new object of type typename is created, the createscript will be evaluated, and it must return an object id. When createscript is evaluated, the pad widget the object is being created on will be added on as an extra argument, followed by any parameters before the options. See the section APPLICATION-DEFINED ITEM TYPES AND OPTIONS in the Programmer's Guide for more information. (Also see the `addoption` command.)

[6] pathName `allocborder` color

Allocates a border for future use by render callbacks. A border is a fake 3D border created by a slightly lighter and a slightly darker color than specified. *Color* may have any of the forms accepted by Tk_GetColor. (Also see the `freeborder` and `drawborder` commands).

[7] pathName `alloccolor` color

Allocates a color for future use by render callbacks. *Color* may have any of the forms accepted by Tk_GetColor. (Also see the `freecolor` and `setcolor` commands).

[8] pathName `allocimage` file [-norgb]

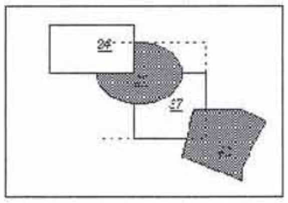
Allocates an image for future use by image objects and render callbacks. *file* specifies the name of a file containing an image. `allocimage` can always read gif file formats. In addition, if Pad++ is compiled with the appropriate libraries, `allocimage` can also read jpeg and tiff image file formats, and will automatically determine the file type. Normally, images are stored internally with their full rgb colors in addition to a colormap index. This allows images to be rendered with dithering, but takes 5 bytes per pixel. If the *-norgb* option is specified, then the original rgb information is not stored with the image and the image can not be rendered with dithering, but only takes 1 byte per pixel. The image may have transparent pixels. This returns an image token which can be used by related commands. (Also see the `freeimage`, `drawimage`, and `info` commands, and the description of `image` items.).

[9] pathName `bbox` [-sticky] tagOrId [tagOrId tagOrId ...]

Returns a list with four elements giving the bounding box for all the items named by the *tagOrId* argument(s). The list has the form "x1 y1 x2 y2" such that the drawn areas of all the named elements are within the region bounded by x1 on the left, x2 on the right, y1 on the bottom, and y2 on the top. If *-sticky* is specified, then the bounding box of the item in sticky coordinates, that is, the coordinates of a sticky item that would appear at the same location on the screen is returned. If no items match any of the *tagOrId* arguments then an empty string is returned.

If the item is sticky then `bbox` returns the bounding box of the item as it appears for the current view. That is, the bounding box will be different when the view is different. If *-sticky* is specified, then the bounding box returned is independent of the current view (i.e., it returns the bounding box as if the view was "0 0 1").

If the item is the Pad++ surface (item #1), then `bbox` will refer to the bounding box of the portion of the surface that is currently visible (based on the view and window size).



.pad bbox 27 37

-75 -55 68 79

[10] pathName `bind` tagOrId [sequence [command]]

This command associates *command* with all the items given by *tagOrId* such that whenever the event sequence given by *sequence* occurs for one of the items the command will be invoked.

This widget command is similar to the Tk bind command except that it operates on items on a Pad++ widget rather than entire widgets. See the Tk bind manual entry for complete details on the syntax of *sequence* and the substitutions performed on *command* before invoking it. The Pad++ widget defines extensions described below, but it is implemented as a complete superset of the standard `bind` command. I.e., you can do everything you can with the canvas with exactly the same syntax, but you can also do more.

If all arguments are specified then a new binding is created, replacing any existing binding for the same *sequence* and *tagOrId* (if the first character of command is "+" then *command* augments an existing binding rather than replacing it). In this case the return value is an empty string. If both *command* and *sequence* are omitted then the command returns a list of all the sequences for which bindings have been defined for *tagOrId*.

The only events for which bindings may be specified are those related to the mouse and keyboard, such as *Enter*, *Leave*, *ButtonPress*, *Motion*, *ButtonRelease*, *KeyPress* and *KeyRelease*. In addition, Pad++ supports some extra bindings including: *Create*, *Modify*, *Delete*, *PortalIntercept*, and *Write*. The handling of events in Pad++ uses the current item defined in *Item IDs and Tags* in the Programmer's Guide. *Enter* and *Leave* events trigger for an item when it becomes the current item or ceases to be the current item; note that these events are different than *Enter* and *Leave* events for windows. Mouse-related events are directed to the current item, if any. Keyboard-related events are directed to the focus item, if any (see the `focus` command below for more on this).

It is possible for multiple bindings to match a particular event. This could occur, for example, if one binding is associated with the item's id and another is associated with one of the item's tags. When this occurs, all of the matching bindings are invoked. The order of firing is controlled by the pad `bindtags` command. The default is that a binding associated with the *all* tag is invoked first, followed by one binding for each of the item's tags (in order), followed by a binding associated with the item's id. If there are multiple matching bindings for a single tag, then only the most specific binding is invoked. A `continue` command in a binding script terminates that script, and a `break` command terminates that script and skips any remaining scripts for the event, just as for the `bind` command.

If bindings have been created for a `pad` window using the Tk bind command, then they are invoked in addition to bindings created for the pad's items using the `bind` widget command. The bindings for items will be invoked before any of the bindings for the window as a whole.

The Pad++ bind command is extended in three ways:

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

Extra macro expansions

When a command is invoked, several substitutions are made in the text of the command that describe the specific event that invoked the command. In addition to the substitutions that the Tk `bind` command makes, Pad++ makes a few more. As with the Tk bind command, all substitutions are made on two character sequences that start with '%'. The special Pad++ substitutions are:

%P: The pad widget that received the event. This is normally the same as %W, but could be different if the event goes through a portal onto a different pad widget.

- %O: The id of the specific item that received the event.
- %l: Information about this event. This has different meanings for different event types. For `<Modify>` events, it specifies the command that caused the modification. For `<PortalIntercept>` events, it specifies the name of the event type generating the PortalIntercept. Standard Tcl event names, such as ButtonPress or ButtonRelease are used. This can be used by PortalIntercept events to only let certain event types go through the portal. Note that only a single PortalIntercept event is generated for a Button, Motion, ButtonRelease sequence, so these three events can not be distinguished in this manner.
- %i: The X-coordinate of the event on the Pad++ surface. This is specified in the current units (i.e., pixels or inches) of the pad widget.
- %j: The Y-coordinate of the event on the Pad++ surface. This is specified in the current units (i.e., pixels or inches) of the pad widget.

- %z: Size of event in pad coordinates. This is dependent on the view. It effectively says how much the event is magnified. I.e., if the view is zoomed in by a factor of two, then this will have a value of two. It is also affected by portals that the event travels through.
- %U: The X-coordinate of the event in object coordinates. This means that the point will be transformed so that it is in the same coordinate system of the object (independent of the object's transformation as well as the current view). This is specified in the current units (i.e., pixels or inches) of the pad widget.
- %V: The Y-coordinate of the event in object coordinates. This means that the point will be transformed so that it is in the same coordinate system of the object (independent of the object's transformation as well as the current view). This is specified in the current units (i.e., pixels or inches) of the pad widget.
- %Z: Size of event in object coordinates. This is dependent on the view and the magnifications of the object.
- %l: The list of portal ids that the event passed through.
- %L: The list of pad surfaces of the portals the event passed through. This list corresponds to the list of portal ids from '%l'.

New Events

Several new events fire at special times, depending on the semantics of the event.

`<create>`: This event gets fired whenever new pad items are created. Because items that this is attached to don't have id's yet, it only makes sense to attach this event to a tag. Then this event gets fired immediately after any item of the relevant tag is created. Example:

```
.pad bind foo <Create> {puts "A foo was created, id=%O"}
.pad create rectangle 0 0 50 50 -tags "foo"
       => A foo was created, id=5
```

`<Modify>`: This event gets fired whenever an item is modified. Modification occurs whenever an item's configuration options are changed, and whenever the following commands are executed on an item: `coords`, `itemconfigure`, `scale`, `slide`, `text`. The %I macro specifies the command that caused the modification. Example:

```
.pad bind foo <Modify> {puts "A foo was modified, cmd=%I"}
.pad create rectangle 0 0 50 50 -tags "foo"
.pad itemconfigure foo -pen red
       => A foo was modified, cmd=itemconfigure
```

`<Delete>`: This event gets whenever an item is deleted. It is typically used to clean up application resources associated with the item that was deleted.

`<Write>`: This event fires whenever an item is written out with the pad `write` command. While Pad++ knows how to generate the Tcl code necessary to recreate itself, items are often part of an application with associated data structures, etc. When an item is written out, it is frequently necessary to write out these associated structures. Sometimes, the application may prefer to substitute its code for pad's. This event provides a mechanism to augment or replace (possibly with an empty string) the Tcl code written out to recreate a pad item.

Whatever string a `<Write>` event returns is appended on to the string pad uses to write out that object. In addition, the application may modify the special global Tcl variable, `Pad_Write` which controls whether the item will get written out. This defaults to 1 (true), but may be set to 0 (false) by the event binding. In addition, the `<Write>` event gets fired on the special tags "`preWrite`" and "`postWrite`" at the beginning and end of the file, respectively, to allow an application to write out code at the ends of the file. Example:

```
.pad bind preWrite <Write> {
```

```
return "Stuff at the beginning of the file"
}
.pad bind postWrite <Write> {
return "Stuff at the end of the file"
}
.pad bind foo <Write> {
return "Stuff after foo objects"
}
.pad bind bar <Write> {
set Pad_Write 0
return "Stuff instead of bar objects"
}
# This forces all objects with the "cat" tag
# to have nothing written out. Notice that an
# empty string must be returned, or "0", the
# result of the set command, will be written out.
.pad bind cat <Write> {
set Pad_Write 0
return ""
}
# This example also has nothing written out,
# but in addition, no other event handlers
# will fire (the object could have multiple
# tags, each with <Write> event handlers).
.pad bind dog <Write> {
Set Pad_Write 0
break
}
```

`<PortalIntercept>`: This event gets fired just before an event passes through a portal. If the event handler executes the break command, then the event stops at the portal and does not pass through. Example:

```
# Events will not go through portals of type "foo"
.pad bind foo <PortalIntercept> {
break
}
```

User-specified modifiers

Event handlers are defined by *sequences* as defined in the Tk `bind` reference pages. A sequence contains a list of *modifiers* which are direct mappings to hardware such as the shift key, control key, etc. Event handlers fire only for sequences with modifiers that are active, as defined by the hardware.

Pad++ allows user-defined modifiers where the user can control which one of the user-defined modifiers is active (if any). The advantage of modifiers is that many different sets of event bindings may be declared all at once - each with a different user-defined modifier. Then, the application may choose which set of event bindings is active by setting the active user-defined modifier. This situation comes up frequently with many graphical programs where there are modes, and the effect of interacting with the system depends on the current mode.

New modifiers must be declared before they can be used with the pad `addmodifier` command (and may be deleted if they are no longer needed with the pad `deletemodifier` command.) Then, the modifier can be used in the pad `bind` command just like a system defined modifier. There may be at most one active user-defined modifier per pad widget. The active user-defined modifier is set with the `setmodifier` command (and may be retrieved with the `getmodifier` command). The current modifier may be set to "" (the default) in which case no user-defined modifier is set. Example:

```
.pad addmodifier Create

.pad addmodifier Run

.pad bind all <Create-ButtonPress-1> {

# Do stuff to create new objects

}

.pad bind all <Run-ButtonPress-1> {

# Do stuff to interact with existing objects

}

# Now the system will be in "Create" mode

.pad setmodifier Create

...

# Now the system will be in "Run" mode

.pad setmodifier Run
```

[11] pathName `bindtags` tagOrId [type]

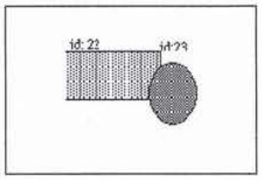
If *type* is specified, this command changes the ordering of event firings on all objects referred to by *tagOrId*. Since more than one event handler may fire for a given event, this controls what order they fire in. If type is "general", events fire most generally first. That is, a binding associated with the *all* tag is invoked first, followed by one binding for each of the item's tags (in order), followed by a binding associated with the item's id. (i.e., *all*, *tags*, *id*). If type is "specific", then events fire most specific first. That is, a binding associated with the item's id is invoked first, followed by one binding for each of the item's tags (in order), followed by a binding associated with the *all* tag (i.e., *id*, *tags*, *all*).

If *tagOrId* is *pathName*, then it does not change the ordering of any objects, but controls the default ordering of objects created in the future.

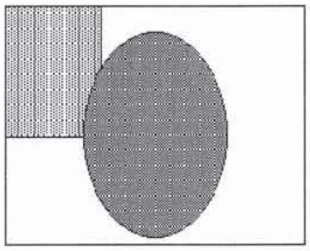
The default event firing order for all objects is "general". This command returns the current event firing order for the first item specified by *tagOrId*.

[12] pathName `center` [-twostep] tagOrId [time x y [z [portalID ...]]]]

Change the view so as to center the first of the specified items so the largest dimension of its bounding box fills the specified amount of screen (z). If *-twostep* is specified, then make the animation in two steps if appropriate (i.e., points not too close). The two steps are such that it zooms out to the midpoint between the two points far enough so that both start and endpoints are visible, and then zooms to the final destination. If *time* is specified, then make a smooth animation to the item in *time* milliseconds. The view is changed so that the item's center appears at the position on the screen specified by x and y, both in the range (0.0 ... 1.0). Here, 0.0 represents the left or bottom side of the window, and 1.0 represents the right or top side of the window. x and y default to (0.5, 0.5), i.e. the center of the screen. If a list of *portalID's* is specified, change the view within the last one specified.



.pad center 23



[13] pathName `centerbbox` [-twostep] x1 y1 x2 y2 [time [x y [z [portalID ...]]]]

Change the view so as to center the specified bounding box so that its largest dimension fills the specified amount of screen (`z`). If *-twostep* is specified, then make animation in two steps if appropriate (i.e., points not too close). The two steps are such that it zooms out to the midpoint between the two points far enough so that both start and endpoints are visible, and then zooms to the final destination. If *time* is specified, then make a smooth animation to the item in *time* milliseconds. The view is changed so that the item's center appears at the position on the screen specified by x and y, both in the range (0.0 ... 1.0). Here, 0.0 represents the left or bottom side of the window, and 1.0 represents the right or top side of the window. x and y default to (0.5, 0.5), i.e. the center of the screen. If a list of *portalID's* is specified, change the view within the last one specified.

[14] pathName `clock` [clockName [reset | delete]]

Creates a clock that is set to 0 at the time of creation. Returns the name of the clock. Future calls with *clockName* return the number of milliseconds since the clock was created (or reset). Calls with *reset* specified reset the clock counter to 0, and return an empty string. Calls with *delete* specified delete the clock, and return an empty string.

```
.pad clock

clock1

.pad clock clock1

8125

.pad clock clock1 reset

.pad clock clock1

1825

.pad clock clock1 delete
```

[15] pathName `configure` [option] [value] [option value ...]

Query or modify the configuration options of the widget. If no option is specified, returns a list describing all of the available options for pathName (see `Tk_ConfigureInfo` for information on the format of this list). If option is specified with no value, then the command returns a list describing the one named option (this list will be identical to the corresponding sublist of the value returned if no option is specified). If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s); in this case the command returns an empty string. Option may have any of the values accepted by the pad command. See the section on WIDGET-SPECIFIC OPTIONS for a description of all the options and their descriptions.

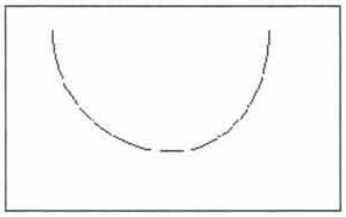
[16] pathName `coords` [-objectcoords] [-append] [-nooutput] tagOrId [x0 y0 ...]

Query or modify the coordinates that define an item. This command returns a list whose elements are the coordinates of the item named by *tagOrId*. If coordinates are specified, then they replace the current coordinates for the named item. If *tagOrId* refers to multiple items, then the first one in the display list is used. The flags may be specified in any order. Note that the `coords` command generates a `<Modify>` event on the items modified by it (see the `bind` command for a description of the `<Modify>` event). Locked items may not be modified by the coords command (see the *-lock* itemconfigure option). The `coords` command can only be used on line, rectangle, polygon and portal items.

If the flag *-objectcoords* is specified, then all coordinates are returned in the item's local coordinate system (i.e., as they were originally specified). If this flag is not specified, then all coordinates are returned in the global coordinate system (i.e., they are transformed by that item's translation and scale parameters).

If the flag *-append* is specified, then all the specified coordinates are appended on to the existing coordinates rather than replacing them.

If the flag *-nooutput* is specified, then this command returns an empty string. Typically, the *-append* and *-nooutput* flags are specified together when adding points to an item and time is of the essence.



```
set id [.pad create line -200 200]

for {set i -20} {$i <= 20} {incr i} {
   set x [expr $i * 10 ]
   set y [expr 0.5 * ($i * $i)]
   .pad coords -append -nooutput $id $x $y
}
```

[17] pathName `create` type [option value ...]

Create a new item in *pathName* of type *type*. The exact format of the arguments after type depends on *type*, but usually they consist of the coordinates for one or more points, followed by specifications for zero or more item options. See the OVERVIEW OF ITEM TYPES subsection below for detail on the syntax of this command. This command returns the id for the new item.

[18] pathName `damage` [tagOrId]

Indicates that some of the screen is damaged (needs to be redrawn). Damages the entire screen if *tagOrId* is not specified, or just the bounding box of each of the objects specified by *tagOrId*. The damage will be repaired as soon as the system is idle, or when the `update` procedure is called. Returns an empty string.

[19] pathName `delete` tagOrId [tagOrId ...]

Delete each of the items given by each *tagOrId*, and return an empty string. Note that the `delete` command generates a `<Delete>` event on the items modified by it (see the `delete` command for a description of the `<Delete>` event). Locked items may not be modified by the `delete` command (see the *-lock* itemconfigure option).

[20] `pathName deletemodifier modifier`

Delete *modifier* from the list of valid user-defined modifiers. Any event bindings that are defined with this modifier become invalid. (Also see the `addmodifier`, `setmodifier`, `getmodifier`, and `bind` commands).

[21] `pathName deletetag tagToDelete tagOrId [tagOrId ...]`

For each item specified by the list of tagOrIds, delete *tagToDelete* from the list of tags associated with the item if it isn't already present on that list. It is possible that no items will be specified by tagOrId, in which case the command has no effect. Note that `dtag` is an acceptable synonym for deletetag. This command returns an empty string.

This command is designed to be used in conjunction with the **find** command. Notice the necessity of using **eval** in this example:
`eval .pad deletetag foo [.pad find withtag bar]`

[22] pathName `drawborder` border type width x1 y1 x2 y2

Draws a fake 3D border connecting the specified coordinates. (See `allocborder` and `freeborder` commands). This command can only be called within a render callback. *Border* must have been previously allocated by `allocborder`. *Type* must be one of "`raised`", "`flat`", "`sunken`", "`groove`", "`ridge`", "`barup`", or "`bardown`". The following example creates an object that draws a border:

```
set border [.pad allocborder #803030]
        .pad create rectangle 0 0 100 100 -renderscript {
        .pad drawborder $border raised 5 0 0 100 100
}
```

[23] pathName `drawimage` imagetoken x y

Draws the image specified by imagetoken at the point (x, y). (Also see `allocimage`, `freeimage`, and `info` commands as well as the description of `image` items). This command can only be called within a render callback.

[24] pathName `drawline` x1 y1 x2 y2 [xn yn ...]

Draws a multi-segment line connecting the specified coordinates. (See `setcolor`, `setlinewidth`, `setcapstyle`, and `setjoinstyle` commands). This command can only be called within a render callback.

[25] pathName `drawpolygon` x1 y1 x2 y2 [xn yn ...]

Draws a closed polygon connecting the specified coordinates. (See `setcolor` and `setlinewidth`). This command can only be called within a render callback.

[26] pathName `drawtext` string xloc yloc

Draws the specified text at the specified location. This command can only be called within a render callback. (Also see the `setcolor`, `setfont`, and `setfontheight` commands.)

[27] pathName `find` [-groupmembers] searchCommand [arg arg ...]

This command returns a list consisting of all the items that meet the constraints specified by *searchCommand* and *arg*'s. The objects are returned in display list order, and if *-groupmembers* is specified, then group members are returned, otherwise, they are not. Note that this command does not return the pad surface (id #1). *SearchCommand* may take any of these forms:

`all`

Returns all the items on the pad.

`below tagOrId`

Returns the item just before (below) the one given by *tagOrId* in the display list. If *tagOrId* denotes more than one item, then the first (lowest) of these items in the display list is used.

`closest x y [halo] [startTagOrId]`

Returns the single item closest to the point given by *x* and *y*. If more than one item is at the same closest distance (e.g. two items overlap the point), then the top-most of these items (the last one in the display list) is used. If *halo* is specified, then any item closer than *halo* to the point is considered to overlap it. (*Halo* must be a non-negative number.) If *halo* is not specified, then only items at the point (x, y) will be found.

The *startTagOrId* argument may be used to step circularly through all the closest items. If *startTagOrId* is specified, it names an item using a tag or id (if by tag, it selects the first item in the display list with the given tag). Instead of selecting the topmost closest item, this form will select the topmost closest item that is below start in the display list; if no such item exists, then the selection behaves as if the start argument had not been specified.

`withinfo info`

Returns all the items containing the string *info* in their info itemconfigure option.

`withlayer layer`

Returns all the items on the layer *layer*.

`withname name`

Returns all the items having *name*.

`withtag tagOrId`

Returns all the items given by *tagOrId*.

```
withtext text
```

Returns all the items containing *text*.

```
withtype type
```

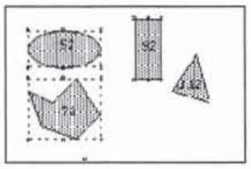
Returns all the items of type *type*.

```
enclosed x1 y1 x2 y2
```

Returns all the items completely enclosed within the rectangular region given by x1, y1, x2, and y2. x1 must be no greater then x2 and y1 must be no greater than y2.

```
overlapping x1 y1 x2 y2
```

Returns all the items that overlap or are enclosed within the rectangular region given by x1, y1, x2, and y2. x1 must be no greater then x2 and y1 must be no greater than y2.



```
.pad find withtag selected
```

52 72 92

[28] pathName `focus` [tagOrId [portalID ...]]

Set the keyboard focus for the Pad++ widget to the item given by *tagOrId*. If a list of *portalID*'s are specified, then the item sits on the surface looked onto by the last portal. If *tagOrId* refers to several items, then the focus is set to the first such item in the display list. If *tagOrId* doesn't refer to any items then the focus isn't changed. If *tagOrId* is an empty string, then the focus item is reset so that no item has the focus. If *tagOrId* is not specified then the command returns the id for the item that currently has the focus, or an empty string if no item has the focus. If the item sits on a different surface than *pathName*, then this command also returns the pathName of the item.

Once the focus has been set to an item, all keyboard events will be directed to that item. The focus item within a Pad++ widget and the focus window on the screen (set with the Tk focus command) are totally independent: a given item doesn't actually have the input focus unless (a) its pad is the focus window and (b) the item is the focus item within the pad. In most cases it is advisable to follow the focus widget command with the focus command to set the focus window to the pad (if it wasn't there already). Note that there is no restriction on the type of item that can receive the Pad++ focus.

[29] pathName `freeborder` border

Frees the *border* previously allocated by `allocborder`. (Also see the `allocborder` and `drawborder` commands).

[30] pathName `freecolor` color

Frees the *color* previously allocated by `alloccolor`. (Also see the `alloccolor` and `setcolor` commands).

[31] pathName `freeimage` imagetoken

Frees the *image* previously allocated by `allocimage`. (Also see the `allocimage` and `drawimage` commands, as well as the description of `image` items).

[32] pathName `getdate`

Returns the current date and time in the standard unix time format.

```
% .pad getdate

Wed May 29 20:01:49 1996
```

[33] `pathName getgroup tagOrId`

Return the group id that *tagOrId* is a member of. If *tagOrId* is not a member of a group, then this command returns an empty string. If *tagOrId* specifies more than one object, then this command refers to the first item specified by *tagOrId* in display-list order. (Also see the `addgroupmember`, and `removegroupmember` commands).

[34] pathName `getlevel`

Returns the current render level This command can only be called within a render callback. (See the sections on *Refinement* and *Region Management and Screen Updating* in the Programmer's Guide for more information about render levels).

[35] pathName `getmag` tagOrId

Returns the current magnification of *tagOrId* for this specific render (it could be rendered multiple times if visible through different portals). Magnification is defined as the multiplication of the current view (including portals) with the object's size (from the *-place* itemconfigure option). This command can only be called within a render callback.

[36] `pathName getmodifier`

Return the current active modifier. (Also see the `addmodifier`, `deletemodifier`, `setmodifier`, and `bind` commands).

[37] pathName `getpads`

Returns a list of all the Pad++ widgets currently defined.

[38] pathName `getportals`

Returns the list of the portals the current object is being rendered within. This command can only be called within a render callback.

[39] pathName `getsize` tagOrId ?portalID ...?

Returns the largest dimension of the first item specified by tagOrId. If a portal list is specified, then the size of the item within the last portal is returned.

[40] pathName `gettags` tagOrId

Return a list whose elements are the tags associated with the item given by *tagOrId*. If *tagOrId* refers to more than one item, then the tags are returned from the first such item in the display list. If *tagOrId* doesn't refer to any items, or if the item contains no tags, then an empty string is returned.

[41] pathName `gettextbbox` string

Returns a list with four elements giving the bounding box of *string* if it is drawn with the `drawtext` command. The list has the form "x1 y1 x2 y2" such that the text is within the region bounded by x1 on the left, x2 on the right, y1 on the bottom, and y2 on the top. The bounding box is affected by the `setfont` and `setfontheight` commands.

[42] pathName `getview` [portalID ...]

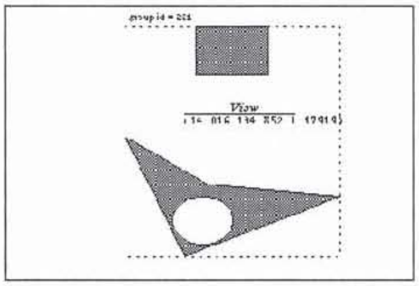
Returns the current view of the main window in "*xview yview zoom*" form. Here, (*xview*, *yview*) specifies the point at the center of the window, and *zoom* specifies the magnification. If a list of *portalID's* is specified, than the view of the last portal is returned instead of the view of the main window. (See `moveto` to set the current view).

.pad getview

14 134 2

.pad ic 221 -place

8 118 1



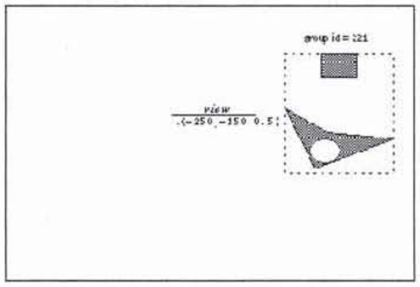
.pad moveto -250 -150 0.5

.pad getview

-250 -150 0.5

.pad ic 221 -place

8.1125 118.753 1



[43] pathName `grid` option arg [arg ...]

The `grid` command arranges one or more objects in rows and columns and treats them as a group. It is based on the Tk grid geometry manager and its behavior and Tcl syntax are very similar to it. In pad, all grid commands are sub-commands of the pad command. See the section on GRID ITEMS for a complete description of this command, and how to create and use grids.

[44] pathName `hastag` tagOrId tag

Determines if the item specified by *tagOrId* contains the specified *tag*. This command returns "1" if the item does contains the specified tag, or "0" otherwise. If *tagOrId* refers to more than one item, then the comparison is performed on the first item in the display list. If *tagOrId* doesn't refer to any items, then "0" is returned.

[45] pathName `info` subcommand

A general command for accessing information about pad and items on the pad surface. *subcommand* may be any of the following: *html* or *image*. Each subcommand may have sub-subcommands and options. All the subcommands and their options follow:

```
html getlastchangedate <tagOrId>
```

Returns the last date this page was modified as specified by the server.

```
html getlength <tagOrId>
```

Returns length of this page in bytes.

```
html getsource <tagOrId>
```

Returns HTML source of this page.

```
html gettype <tagOrId>
```

Returns Mime type of this page as specified by the server.

```
image getdim <imagetoken>
```

Returns dimensions {x y} of this image in pixels.

```
image getname <imagetoken>
```

Returns filename this image was loaded from.

[46] pathName `islinked`

WARNING: `islinked` is an obsolete command and will be removed in the next release. Replace all uses of `islinked` with the Tk 'winfo ismapped' command.

Returns a flag specifying if *pathName* has been mapped to the display yet.

[47] pathName `itemconfigure` [-nondefaults] tagOrId [option [value] ...]

This command is similar to the `configure` command except that it modifies item-specific options for the items given by *tagOrId* instead of modifying options for the overall pad widget. `ic` is an allowed synonym for `itemconfigure`. If no *option* is specified, then this command returns a list describing all of the available options for the first item given by *tagOrId*. If the *-nondefaults* flag is specified, then only those options modified by an application will be returned. If *option* is specified with no *value*, then the command returns the value of that option. If one or more option-value pairs are specified, then the command modifies the given widget option(s) to have the given value(s) in each of the items given by *tagOrId*; in this case the command returns an empty string. If *value* is an empty string, then that option is set back to its default value.

The options and values are the same as those permissible in the `create` command when the item(s) were created; see the sections below starting with OVERVIEW OF ITEM TYPES for details on the legal options. Note that the `itemconfigure` command generates a `<Modify>` event on the items modified by it (see the `itemconfigure` command for a description of the `<Modify>` event). Locked items may not be modified by the `itemconfigure` command (see the *-lock* itemconfigure option).

[48] pathName `line2spline` error x1 y1 ... xn yn

Takes the coordinates for a line, and uses an adaptive curve fitting algorithm to generate the coordinates for a spline that approximates the line. The spline coordinates are returned. *error* is a floating point number indicating how closely the spline curve should follow the line. Using a smaller error will tend to generate a spline made with more bezier segments that follow the line more accurately. Using a larger error will produce fewer bezier segments but the fit will be less accurate. See the section on SPLINE ITEMS on how splines are specified in Pad++. (Also see `spline2line`.)

[49] pathName `lower` [-one] tagOrId [belowThis]

Move all of the items given by *tagOrId* to a new position in the display list just before the item given by *belowThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *belowThis* is a tag or id; if it refers to more than one item then the first (bottommost) of these items in the display list is used as the destination location for the moved items. If *belowThis* is not specified, then *tagOrId* is lowered to the bottom of the display list. If the *-one* flag is specified, then *tagOrId* is lowered down one item in display order which may or may not have a visible effect. *-one* and *aboveThis* may not both be specified. If any items to be lowered are group members, they are lowered within their group rather than being lowered on the pad surface. Returns an empty string.

[50] pathName `moveto` [-twostep] xview yview zoom [time [portalID ...]]

Change the view so that the point "*xview yview*" is at the center of the screen with a magnification of *zoom*. If *xview*, *yview*, or *zoom* is specified as "", then that coordinate is not changed. If *-twostep* is specified, then make animation in two steps if appropriate (i.e., points not too close). The two steps are such that it zooms out to the midpoint between the two points far enough so that both start and endpoints are visible, and then zooms to the final destination. If *time* is specified, then the change in view will be animated in enough evenly spaced frames to fill up *time* milliseconds. If a list of *portalID*'s are specified, then the view will be changed within the last specified *portalID* rather than within the main view. The return value is the current view. (See `getview` to get the current view).

[51] pathName `noise` index

Returns a repeatable noise value based on the floating-point value of index. This noise function is equal to 0 whenever index is an integer. Typically, noise is called with slowly incrementing values of index. The closer the consecutive values of index are, the higher the frequency of the resulting noise will be. This noise function is from Ken Perlin at New York University (http://www.mrl.nyu.edu/perlin).

Example:

```
set coords ""
set noiseindex_x 0.1928
set noiseindex_y 100.93982
set noiseincr 0.052342
for {set i 0} {$i < 100} {incr i } {
        set x [expr 500.0 * [.pad noise $noiseindex_x]]
        set y [expr 500.0 * [.pad noise $noiseindex_y]]
        lappend coords $x
        lappend coords $y
        set noiseindex_x [expr $noiseindex_x + $noiseincr]
        set noiseindex_y [expr $noiseindex_y + $noiseincr]
}
eval .pad create line $coords
```

[52] pathName `padxy` [-sticky] [-portals] winx winy [-gridspacing value]

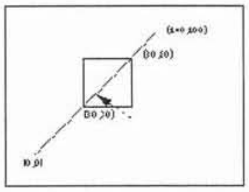
Given a window x-coordinate *winx* and y-coordinate *winy*, this command returns the pad x-coordinate and y-coordinate that is displayed at that location. If *-sticky* is specified, the coordinate transform is done ignoring the current view (i.e., as for sticky objects.) If *-portals* is specified, then the point (*winx*, *winy*) is passed through any portals it on. If *-gridspacing* is specified, then the pad coordinate is rounded to the nearest multiple of value units.

[53] pathName `pick` [-divisible] [-indivisible] winx winy

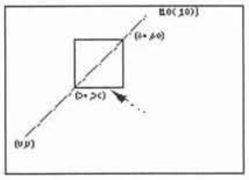
Given a window coordinate (*winx*, *winy*), it returns the visible object underneath that point. If the point should pass through any portals, a `<PortalIntercept>` event will be fired which will determine if the event will pass through that portal. By default, the `pick` command uses the divisibility of individual groups to determine if group members should be picked. However the *-divisible* or *-indivisible* flags (only one of which may be specified) override group's divisibility. If *-divisible* is specified, then group members will be picked from any group the point hits. If *-indivisible* is specified, then group objects and not group members will be picked.

```
% .pad create line 0 0 100 100
22
.pad create rectangle 30 30 80 80
23

.pad addmodifier  Pick
.pad bind all <Pick-ButtonPress-1> {
        event_Press  %i %j %x %y %O
}

proc event_Press {i j x y obj} {
                                        # Get the group object not the group members
                                        # underneath the point x y
        set container [.pad pick -indivisible $x $y]
        puts "container $container object: $obj coords: ($i, $j)"
}

.pad setmodifier Pick
```

```
Now, group the line and rectangle:

                 % .pad create group -members "22 23"
                 24
```



```
Now, click on the line, the system response with:
                 container 24 object: 22 coords: (37.5, 36)
```



```
Now, click on the rectangle,  system response with:
                 container 24 object: 23 coords: (66.5, 28)

Now, change the pick command as:
                 set container [.pad pick -divisible $x $y]:

Then click on the line:
                 container 22 object: 22 coords: (52.5, 52)

Click on the rectangle:
                 container 23 object: 23 coords: (63.5, 30)
```

[54] pathName `popcoordframe`

Pops the top frame off the stack of coordinate frames. The resulting frame on the top of the stack becomes active. Also see `pushcoordframe` and `resetcoordframe`. Returns the frame popped off the stack.

[55] pathName `printtree`

Prints the current hierarchical tree of items to stdout (used for debugging). Returns an empty string.

[56] pathName `pushcoordframe` tagOrId

`pathName pushcoordframe x1 y1 x2 y2`

Pushes a coordinate frame onto the stack of coordinate frames. When any coordinate frames are on the stack, all coordinates are interpreted relative to the frame instead of as absolute coordinates. A frame is a bounding box, and all coordinates are specified within the unit square where the unit square is mapped to the frame.

Note that the *-penwidth* and *-minsize* and *-maxsize* itemconfigure options are also relative to the coordinate frame. In these cases, a value of 1 refers to the average of the frame dimensions.

Text and images are scaled so that one line of text, or the height of the image is scaled to the height of the coordinate frame at a scale of 1 (using the *-place* or *-z* itemconfigure options).

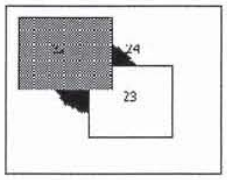
For example, the following code makes 50 nested rectangles. Note that the width of the rectangles shrinks proportionally.

```
for {set i 0} {$i < 50} {incr i} {
        set id [.pad create rectangle 10 10 80 80 -penwidth 2]
        .pad pushcoordframe $id
}
.pad resetcoordframe
```

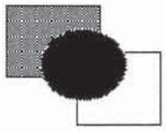
Also see `popcoordframe` and `resetcoordframe`. Returns the current coordinate frame.

[57] pathName `raise` [-one] tagOrId [aboveThis]

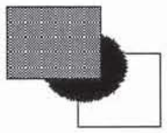
Move all of the items given by *tagOrId* to a new position in the display list just after the item given by *aboveThis*. If *tagOrId* refers to more than one item then all are moved but the relative order of the moved items will not be changed. *aboveThis* is a tag or id; if it refers to more than one item then the last (topmost) of these items in the display list is used as the destination location for the moved items. If *aboveThis* is not specified, then *tagOrId* is raised to the top of the display list. If the *-one* flag is specified, then *tagOrId* is raised up one item in display order which may or may not have a visible effect. *-one* and *aboveThis* may not both be specified. If any items to be raised are group members, they are raised within their group rather than being raised on the pad surface. Returns an empty string.



.pad raise 24



```
If we use the -one option:
.pad raise -one 24

    The original position turns to be:
```



[58] pathName `read` filename

Executes the tcl commands in the filename. If filename is created with the write command, then this command reads the pad scene back in. Returns an empty string.

[59] `pathName removegroupmember [-notransform] tagOrId`

Remove all items specified by *tagOrId* from the group they are a member of, and return them to the pad surface. If any of the items were members of hierarchical groups, they are removed from all groups. If any of the items are not a member of a group, then they are not affected. Items removed are added to the pad surface just after the group in terms of display-list order.

By default, items are transformed so they don't change their location when removed from a group - even if the group has a transformation. This is implemented by transforming the item's transformation to be the inverse of the group's transformation. If the *-notransform* flag is specified, this inverse transformation is not applied, and the item will move by the group's transformation when removed. *(*Also see the `addgroupmember`, and `getgroup` commands). Returns an empty string.

[60] pathName `renderitem` [tagOrId]

During a render callback triggered by the *-renderscript* option, this function actually renders the object. During a *-renderscript* callback, if `renderitem` is not called, then the object will not be rendered. If *tagOrId* is specified, then all the items specified by *tagOrId* are rendered (and the current item is not rendered unless it is in *tagOrId*). This function may only be called during a render callback. Returns an empty string.

[61] pathName `resetcoordframe`

Pops all the frames off of the coordinate stack. Results in an empty stack, so all coordinates are back to absolute coordinates. Also see `pushcoordframe` and `popcoordframe`. Returns an empty string.

[62] pathName `scale` tagOrId [scaleAmount [padX padY]]

Scale each of the items given by *tagOrId* by multiplying the size of the item with *scaleAmount*. Scale the items around the item's center, or around the point (padX, padY), if specified. This command returns the scale of the first item. Note that the `scale` command generates a `<Modify>` event on the items modified by it (see the `scale` command for a description of the `<Modify>` event). Locked items may not be modified by the `scale` command (see the *-lock* itemconfigure option).

[63] pathName `setcapstyle` capstyle

Sets the capstyle of lines for drawing within render callbacks. *Capstyle* may be any of: "butt", "projecting", or "round". This command can only be called within a render callback.

[64] pathName `setcolor` color

Sets the color for future drawing with render callbacks. *Color* must have previously been allocated by `alloccolor`. This command can only be called within a render callback. (Also see the `alloccolor` and `freecolor` commands).

[65] pathname `setfont` fontname

Sets the font for future drawing with render callbacks. This affects the result of the `gettextbbox` command. *Fontname* must specify a filename which contains an Adobe Type 1 font, or the string "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`". (Also see the `setfontheight` command).

[66] pathname `setfontheight` height

Sets the height of the font for future drawing with render callbacks. *Height* is specified in the current pad units. This affects the result of the `gettextbbox` command. (Also see the `setfont` command).

[67] pathname `setid` tagorid id

Sets the id of an existing item to *id*. If tagord specifies more than one item, then the first item is used. Returns an empty string. This generates an error if an invalid id is specified (i.e., if it is in use), or if *tagorid* does not specify an object.

[68] pathName `setjoinstyle` joinstyle

Sets the joinstyle of lines for drawing within render callbacks. *Joinstyle* may be any of: "bevel", "miter", or "round". This command can only be called within a render callback.

[69] pathName `setlanguage` language

Sets the language to be used for callback scripts that are created in the future. All callback scripts that have already been created will be evaluated in the language that was active at the time they were created. This command refers to all callback scripts including event handlers, render scripts, timer scripts, zoom actions, etc. Pad++ always includes at least the Tcl scripting language, but others may be active, depending on how Pad++ was built. This command controls whatever languages are currently installed. The language defaults to "automatic" where it tries to guess the language based on the syntax of the script. See the SCRIPTING LANGUAGES section in the Programmer's Guide for more details. (Also see the `settoplevel` command.)

[70] pathName `setlinewidth width`

Sets the linewidth (in current units) to *width* for future drawing with render callbacks. The actual width of the line will depend on the size of the object and the magnification of the view. If width is 0, then the line is always drawn 1 pixel wide. This command can only be called within a render callback.

[71] `pathName setmodifier modifier`

Make *modifier* be the current active modifier for this pad widget. *modifier* must have been previously defined with the setmodifier command. (Also see the `addmodifier`, `deletemodifier`, `getmodifier`, and `bind` commands).

[72] `pathName settoplevel language`

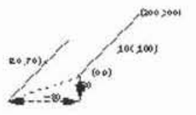
Sets the language that the top-level interpreter should use. Pad++ always includes at least the Tcl scripting language, but others may be added. Returns an empty string. See the SCRIPTING LANGUAGES section in the Programmer's Guide for more details. (Also see the `setlanguage` command.)

[73] pathName `shape` [innercoords outercoords]

WARNING: `shape` has been renamed to windowshape, and will be removed in the next release. Replace all uses of `shape` with the `windowshape` command.

[74] pathName `slide` tagOrId [dx dy]

Slide each of the items given by *tagOrId* by adding *dx* and *dy* to the x and y coordinates of the item's transformation (i.e., their *-place* itemconfigure option). This command returns a string with the (x, y) position at the item's anchor point. Note that the `slide` command generates a `<Modify>` event on the items modified by it (see the `slide` command for a description of the `<Modify>` event). Locked items may not be modified by the `slide` command (see the *-lock* itemconfigure option).



.set id [.pad create line 0 0 200 200]

.pad slide $id -80 30

20.000000 70.000000

[75] pathName `spline2line` error x1 y1 ... xn yn

Takes the coordinates for a spline and uses an adaptive bezier algorithm to generate the coordinates for a line that apprxoimates the spline. *error* is how much error is allowed - a small error produces a greater number of points and more accuracy. A large error yields fewer points but the line is less accurate. See the section on SPLINE ITEMS for details on how splines are created. (Also see `line2spline`.)

[76] pathName `text` tagOrId option [arg ...]

Controls all interaction with a text item. See TEXT ITEMS for a description of indices and marks. *tagOrId* specifies the text item to apply the following command to. *Option* and the *args* determine the exact behavior of the command. Note that the `text` command generates a `<Modify>` event on the items modified by it (see the `text` command for a description of the `<Modify>` event). Locked items may not be modified by the `text` command (see the *-lock* itemconfigure option). The following command options are available:

-
- `compare index1 op index2`

  Compares the indices given by *index1* and *index2* according to the relational operator given by *op*, and returns 1 if the relationship is satisfied and 0 if it isn't. *Op* must be one of the operators <, <= ==, >=, >, or !=. If op is == then 1 is returned if the two indices refer to the same character, if op is < then 1 is returned if *index1* refers to an earlier character in the text than *index2*, and so on.
- `delete index1 [index2]`

  Delete a range of characters from the text. If both *index1* and *index2* are specified, then delete all the characters starting with the one given by index1 and stopping just before *index2* (i.e. the character at *index2* is not deleted). If *index2* doesn't specify a position later in the text than *index1* then no characters are deleted. If *index2* isn't specified then the single character at *index1* is deleted. The command returns an empty string.
- `get index1 [index2]`

  Return a range of characters from the text. The return value will be all the characters in the text starting with the one whose index is *index1* and ending just before the one whose index is *index2* (the character at *index2* will not be returned). If *index2* is omitted then the single character at *index1* is returned. If there are no characters in the specified range (e.g. *index1* is past the end of the file or *index2* is less than or equal to index1) then an empty string is returned.
- `index index [char]`

  Returns the position corresponding to index in the form line.char where line is the line number and char is the character number. If char is specified, then the position is returned in the form char which is the character index from the beginning of the file. Index may have any of the forms described under INDICES.
- `insert` index chars

  Inserts chars into the text just before the character at *index* and returns an empty string.
- `mark option [arg arg ...]`

  This command is used to manipulate marks. The exact behavior of the command depends on the *option* argument that follows the mark argument. The following forms of the command are currently supported:

```
mark names
```

Returns a list whose elements are the names of all the marks that are currently set.

```
mark set markName index
```

Sets the mark named *markName* to a position just before the character at index. If *markName* already exists, it is moved from its old position; if it doesn't exist, a new mark is created. This command returns an empty string.

```
mark unset markName [markName ...]
```

Remove the mark corresponding to each of the *markName* arguments. The removed marks will not be usable in indices and will not be returned by future calls to `pathName mark names`. This command returns an empty string.

[77] pathName `tree` subcommand [args ...]

This command creates, maintains, and animates dynamic trees of Pad items. Items are created by other pad functions, and are placed into hierarchical tree structures to be managed by this code. These trees support a focus + context viewing structure, multiple focii, and a focus function which has a controlled level of influence on the tree.

Each node has a layout object associated with it which controls the position and resizing of the pad item at that node during a layout. Each layout controls a link item - a pad item created by the tree code, which graphically connects the node to its parent. This link item is maintained automatically by the tree code, but may be accessed and manipulated through the `tree` subcommand.

Each pad has a treeroot object, which is a list of all pad tree nodes on the surface. Each of these "root nodes" is an invisible treenode which controls certain subtrees on the pad surface. This organization is necessary to keep trees independent. Animation done at a node affects that node and its children, so we need to be careful to organize the nodes in such a way that all nodes we wish to "know" about each other are connected in some manner. Separate hierarchies can be made to "avoid" each other during animation by connecting them togethe under an invisible root node. When the layout function is called on the root node, both hierarchied will be laid out according to the layout object which resides at the root node.

A dynamic tree supports an abitrary number of foci. Management of these foci is left up to the user. A node's focus is spread by a function which has several parameters. See the `setfocus` subcommand for more information.

Manipulation of the tree structure falls into four parts - tree management, layout, animation control, and parameter control.

Tree Management

A tree can be added to by creating new nodes and adding them to the existing tree structure. Nodes and subtrees can be moved within trees. Nodes and subtrees can be deleted, which will also delete the pad item associated with the treenode. Nodes and subtrees can be removed, which simply removes the treenode associated with the object, but leaves the object itself alone.

Layout

The default layout provided with the current version of this code creates a hierarchical tree in which a node's children are laid out to the right of the node. This layout prevents any overlapping of nodes by calculating the bounding box of the subtree rooted at a node, and laying out nodes so that these bounding boxes do not intersect.

Animation control

A tree always animates its members. It may also animate the view at the same time the members are being animated.

Parameter control.

There are a variety of parameters associated with the layout at a node, and the control of animation of a tree.

Trees are created and manipulated through the tree subcommands:

`addnode` childtagOrId parenttagOrId

Adds *childtagOrId* to *parenttagOrId* as a child. If *childtagOrId* already has a parent, this command also removes *childtagOrId* from that parent. When it is added to the tree, the item's current zoom is recorded, and is used in all future calculations in the dynamic tree layouts. This means that an item's size when it is added to the tree is the size that it will have when it has a focus of 1.0. (See the tree `setscale` command to modify the size of an item after it has been added to a tree.)

`animatelayout` tagOrId [-view view]

Used in conjunction with computelayout, this command performs the animated layout of a tree. It may be given a view, which forces the system to animate the system view while the tree animation is taking place. Use `getlayoutbbox` to calculate a view for the finished animation. See `computelayout` for specific implementation instructions.

Using `animatelayout` with the *-view* option forces an animation of the view as the tree is animating. The view animates from the current view to the one specified as the tree animation is taking place.

`animateview` tagOrId [value]

Sets the animateView flag at *tagOrId*. Controls whether or not a layout will animate the view when layout is called at *tagOrId*.

`connect` tagOrId

Draws links from *tagOrId* to its parent, and from *tagOrId*'s children to *tagOrId.*

`computelayout` tagOrId

Computes the final layout state for a dynamic tree. This places final layout state information in the tree, some of which can be accessed in order to control the layout. For information on accessing some of this information, see the `getlayoutbbox` command.

This code computes the future layout of a tree, then animates its view so that the center of the tagOrId's future position is in the center of the screen at the end of the animation. Note that any treenode which is a descendant of *tagOrId* will return valid information on a call to get `layoutbbox`. Other nodes are not guaranteed to have valid information.

```
.pad tree computelayout $node
set futureBbox [.pad tree getlayoutbbox $node]
set view [bbcenter $futureBbox]
.pad tree animatelayout -view $view
```

`create` tagOrId

Creates a treenode to monitor *tagOrId*. Creates default layout for treenode. Adds *tagOrId* to the padroot, in preparation for placement somewhere else in the hierarchy.

`createroot`

Creates an invisible root node which is used to organize subtrees of information, and returns the pad id of the dummy object at that node. Used to connect several nodes together so that they appear to be root nodes at the same level. Because this is an invisible node, no links will be drawn to it.

`delete` [-subtree] tagOrId

Delete the *tagOrId* and its associated pad object, layout, and link. By default, when there is no subtree option, *tagOrId*'s children are promoted to be children of *tagOrId*'s parent. If the *-subtree* option is used, the entire subtree and all of its associated data and pad objects are deleted.

`getchildren` tagOrId

Returns a list of the ids of the pad objects which are children of *tagOrId*

`getfocus` tagOrId

Returns the focus value at a *tagOrId*, which is a number on the interval [0.0, 1.0]

`getlayoutbbox` tagOrId

Returns the approximate bbox *tagOrId* will have at the end of the current animation. This is only valid when used after `computelayout`, and before any manipulation of any member of the tree. Moving or resizing any object affected by `computelayout` will cause a few bugs in the animation of those objects when `animatelayout` is called. The system will not break, but any moved object will first instantly move to the position it held when `computelayout` was called, and then will animte to the position `computelayout` determined for that object. Relative sizing of objects will be ignored by the system.

`getlink` tagOrId

Return the id of the item which graphically links *tagOrId* to its parent.

`getparent` tagOrId

Return the id of the parent of *tagOrId.*

`getroot` tagOrId

Gets the root node of *tagOrId*'s hierarchy - the node which resides just below the padroot.

`isnode` tagOrId

Returns a boolean indicating whether or not *tagOrId* has a treenode attached to it, and is therefore a member of a hierarchy.

`layout` tagOrId [-view view]

Performs a recursive layout of the subtree rooted at *tagOrId*. If the *-view* option is used, the tree will animate to the view provided.

`lower` tagOrId [belowtagOrId]

Controls the position of *tagOrId* in the order of its siblings. If *belowtagOrId* is not provided, *tagOrId* is moved to the bottom of the list. If *belowtagOrId* is provided, *tagOrId* is moved to a position just above (after) *belowtagOrId.*

`raise` tagOrId [abovetagOrId]

Controls the position of *tagOrId* in the order of its siblings. If *abovetagOrId* is not provided, *tagOrId* is moved to the top of the list. If *abovetagOrId* is provided, *tagOrId* is moved to a position just above (after) *abovetagOrId.*

`removenode` [-subtree] tagOrId

Removes the treenode and layout objects associated with *tagOrId*. If the *-subtree* is not included, *tagOrId*'s information is removed, and *tagOrId*'s children are promoted. If the *-subtree* option is used, the entire treenode hierarchy is removed.

`reparent` [-subtree] tagOrId parenttagorid

Reparents *tagOrId* to belong to *parenttagorid*. The default case, in which the *-subtree* option is not used, reparents *tagOrId*, and promotes any children *tagOrId* may have to be children of *tagOrId*'s original parent. If the *-subtree* option is used, the subtree rooted at *tagOrId* is moved.

`setanimatespeed` tagOrId milliseconds

Sets the time for an animation to occur. If this number is 0, the animation will proceed immediately to the end state. During an animation, if any event is detected, the animation will proceed to the end state. Thus, a double click on a treenode forces the animation to happen instantaneously.

`setfocus` tagOrId [value [levels [falloff]]]

Set the focus value at a *tagOrId*. This must be a number on the range [0,1]. If no *value* is provided, the focus is set to 1.0. The *levels* parameter controls the number of levels this focus is allowed to spread. The *falloff* parameter is a multiplier which controls the portion of focus which is passed on to the next level of recursion. For example, if this number is 0.75, then *focus**0.75 of the focus is passed on at the next level of recursion.

`setfocusmag` tagOrId value

Recursive set command - works on the entire subtree of the *tagOrId* is is given. Set the magnification difference between an object of focus 0 and an object of focus 1.

`setscale` tagOrId value

Set the scale that an object will have when its focus is 0. This is the smallest size that an object will have in a dynamic tree. When a tree *tagOrId* is created, this value is automatically set to the z value of the object.

`setspacing` tagOrId xvalue [yvalue]

Set the x and y spacing at a *tagOrId*. This is the amount of spacing between a *tagOrId* and its spatial neighbors.

[78] pathName `type` tagOrId

Returns the type of the item given by *tagOrId*, such as rectangle or text. If *tagOrId* refers to more than one item, then the type of the first item in the display list is returned. If *tagOrId* doesn't refer to any items at all then an empty string is returned.

[79] pathName `update` [-dissolve speed [withRefinement]]

This forces any outstanding updates to occur immediately. If the -dissolve flag is specified, then speed determines how quickly the update is done. If speed is 0, the update will happen quickly with a swap buffer. If speed is between 1 and 3, the update will happen with a dissolve effect where 1 is the fastest and 3 is the slowest. If the *withRefinement* flag is specified, this forces all refinements to occur immediately as well - which could be a slow process. Returns an empty string.

[80] pathName `urlfetch` URL ?option value ...?

```
pathName urlfetch Token
   where valid options are:
        -file <filename>
                -var <variable>
        -updatescript <updateScript>
        -donescript <doneScript>
        -errorscript <errorScript>
```

Retrieves the specified *URL* (Universal Resource Locator) from the World Wide Web. This command returns immediately, and the retrieval is done in the background (within the same process using a file handler.) As portions of the data comes in, *updateScript* will be executed, and *doneScript* will be executed when all of the data has completely arrived. If there are any errors retrieving the data, then *errorScript* will be executed. `urlfetch` returns a token that can be used to interact with this retrieval. This token is appended to *updateScript*, *doneScript* and *errorScript* when the scripts are executed.

There are three methods to access the data retrieved by urlfetch. The first method is to specify a file (with *-file*) in which case the data is written to that file as it is retrieved. The second method is to specify a Tcl variable (with *-var*) in which case the data is stored in that global variable as it is retrieved. The variable will be updated with the current data before *updateScript* and *doneScript* are executed. Note that the variable is not cleared by `urlfetch` and it is the responsibility of the caller to free it (with `unset`). The third method is to use the second form of `urlfetch` by passing it url token during an updatescript callback in which case it will return the data retrieved by that fetch. Three code segments follow which show the use of urlfetch.

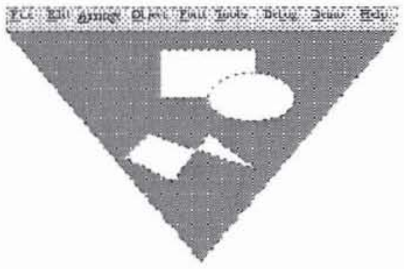
```
#
# urlfetch example using a file
#
proc done {filename token} {
        set file [open $filename "r"]
        ...  # handle file
}
set file "foo"
.pad urlfetch http://www.cs.unm.edu -file $file \
        -donescript "done $file"

#
# urlfetch example using a Tcl global variable
#
proc done {token} {
        global foo

        ...  # handle data in "foo"
        unset foo     ;# no longer need URL data
}
.pad urlfetch http://www.cs.unm.edu -var foo \
        -donescript "done"
#
# urlfetch example using a token to incrementally
# handle data as it comes in.
#
proc update {token} {
        set data [.pad urlfetch $token]
        ...  # handle incremental data
}
.pad urlfetch http://www.cs.unm.edu \
        -updatescript "update" -donescript "done"
```

[81] pathName `windowshape` [innercoords outercoords]

Changes the shape of the top-level window containing the pad widget specified by *pathName*. The two parameters each specify lists of coordinates that specify the shape of the window. All coordinates are scaled to fit the existing width of the window, larger numbers in X go to the right, and larger numbers in Y go up. *innercoords* represents the area that can be painted in, and *outercoords* represents the overall window shape. The difference between these two shapes becomes the windows border. If *innercoords* and *outercoords* are both empty strings, then the window returns to its default rectangular shape. This command returns the current window shape.

For example, the following command changes the top-level window shape to an inverted triangle.

```
.pad windowshape {0 50 50 50 25 0} {0 50 50 50 25 0}
```



[82] pathName `write` filename [tagOrId tagOrId ...]

Writes the Tcl commands necessary to recreate the items on the Pad++ surface into *filename*. If *tagOrId's* are specified, then just those items are written out. The file that is written out should be read back in with the **read** command. If *filename* is an empty string, than this command returns the string instead of writing it to a file. If a valid filename is specified, then this command returns an empty string.

Only non-default slots of each object are written out.

As the **write** command writes out objects on the pad, it generates a `<Write>` event for each item it writes. The return string from the `<Write>` event handler will be appended to whatever string this function writes out for each item. See the **bind** command for more information on this.

[83] pathName `zoom` zoomFactor padXloc padYloc [animateTime [portalID ...]]

Zoom around specified pad position by multiplicitive factor. If *animateTime* is specified, then animate the zoom and take *animateTime* milliseconds for the animation. If an optional list of portals is specified, then change the view within the last portal. The entire list is necessary in case the last portal is sitting on a different surface then this function is called with. Returns an empty string.

Overview of Item Types

The sections below describe the various types of items supported by Pad++ (grid, group, handle, html, image, kpl, line, polygon, portal, rectangle, spline, tcl, text, and textfile). Each item type is characterized by two things: first, the form of the command used to create instances of the type; and second, a set of itemconfiguration options for items of that type, which may be used in the `create` and `itemconfigure` widget commands. See the `itemconfigure` command for the syntax to use these options.

- All Item Types
- Grid Items
- Group Itemss
- Handle Itemss
- HTML Items
- Image Itemss
- KPL Items
- Line Items
- Pad Items
- Polygon Items
- Portal Items
- Rectangle Items
- Spline Items

All Item Types

These are the options that are supported by all item types:

[1] -alwaysrender boolean

(available only for all item types)

The rendering engine may decide to not render an item for reasons of efficiency (although it may get rendered at higher levels of refinement). When this flag is set (i.e., equals 1), the item will be rendered no matter how big it is (as long as it is bigger than its *-minsize*. Defaults to false (0).

[2] -anchor anchorPos

(available only for all item types)

AnchorPos tells how to position the object relative to the positioning point for the item (see `-place`); it may have any of the forms accepted by Tk_GetAnchor. For example, if anchorPos is "center" then the object is centered on the point; if anchorPos is "n" then the object will be drawn so that its top center point is at the positioning point. This option defaults to center.

[3] -clipping boolean

(available only for all item types)

By default, built-in items (such as lines, text, etc.) do not get clipped to their bounding box, and procedural items (items with *-renderscripts*) do. This flag turns clipping on or off. Be warned, that turning off clipping for a procedural object is dangerous. If you draw outside the object's bounding box, you can end up with screen garbage. Defaults to true (1) for items with *-renderscripts*, and false (0) for all other items.

[4] -events boolean

(available only for all item types)

Controls whether an item receives input events. If set to false (0), it does not respond to events. Defaults to true (1).

[5] -faderange value

(available only for all item types)

Controls over how long a period an item fades out as it approaches its minimum or maximum size. *value* specifies this period as a percentage of the object's size (from 0.0 to 1.0). Where 0.0 means that the item doesn't fade out all, it just blinks off when its extreme is reached, and 1.0 means that it slowly fades out over its entire range of sizes. Defaults to 0.3. (Also see the *-minsize* and *-maxsize* itemconfigure options.)

[6] -height height

(available only for all item types)

By default, the height of every item is automatically computed based on its contents. If the -height option is set, however, then this overrides the automatically computed value. Items are centered within the specified *height*. If the dimensions are specified as smaller than the default values, the item is clipped to those dimensions. (Also see the -width itemconfigure option.)

[7] -info info

(available only for all item types)

A generic info field where the user may place any string. (See the **find** withinfo command).

[8] -layer layer

(available only for all item types)

Specifies the layer the item is on. Every item sits on a layer (which is specified by a string), and each view (top-level window and portals) specifies which layers are visible within that view. This gives control over objects are visible where and can be used with portals to implement very simple filters. (See the *-visiblelayers* itemconfigure option of portals and the top-level window which is specified by the surface (item 1). Defaults to "main".

[9] -lock lock

(available only for all item types)

When an item is locked, it can not be deleted or modified (except for changing the lock status). Note that attempting to modify or delete a locked item does not generate an error. It fails silently. This is so it is easy to modify all items associated with a tag and if certain items are locked they will just not get modified. The restricted commands on locked items are: `coords`, `delete`, `itemconfigure`, `scale`, `slide`, and `text`.

[10] -maxsize size

(available only for all item types)

Specifies the maximum size (in current units) this item should be rendered at. That is, if the view is such that the largest dimension of this object is greater than size units, it will not be displayed. When an object is not displayed because it is too large, it does not receive events. When an object approaches its maximum size it will fade out until it completely disappears when it reaches its maximum size. If *size* is -1, then it has no maximum size and will never disappear because it is too large. See the *-faderange* itemconfigure option to control how quickly an item fades out.

size may also be specified as a percentage of the view it is visible in (top-level window or portal). To specify size as a percentage, it should be in the range from 0 to 100 and end with a "%". Example:

```
.pad ic 5 -minsize 55%
```

size defaults to 10,000 pixels.

Also note that the rendering engine may decide to not display an item for reasons of efficiency if it is reasonably small. See the *-alwaysrender* flag to avoid this.

[11] -minsize size

(available only for all item types)

Specifies the minimum size (in current units) this item should be rendered at. That is, if the view is such that the largest dimension of this object is less than size units, it will not be displayed. When an object is not displayed because it is too small, it does not receive events. When an object approaches its minimum size it will fade out until it completely disappears when it reaches its minium size. See the *-faderange* itemconfigure option to control how quickly an item fades out.

size may also be specified as a percentage of the view it is visible in (top-level window or portal). To specify size as a percentage, it should be in the range from 0 to 100 and end with a "%". Example:

```
.pad ic 5 -minsize 55%
```

size defaults to 0.

Also note that the rendering engine may decide to not display an item for reasons of efficiency if it is reasonably small. See the *-alwaysrender* flag to avoid this.

[12] `-place`: Place sets the anchor position of the object.

(available only for all item types)

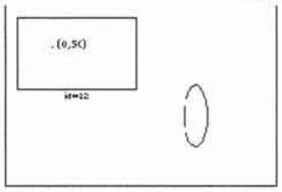
Every object has a `-place` which specifies the anchor point of that object (see `-anchor`). The place specifies the object's position and size. The size is multiplicitive. Place can be one of:

-
- "x y size"

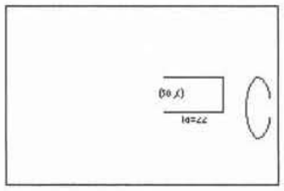
  Specifies (x, y, size) where 'x y size' is a single string specifying anchor point and item size. Items that have coordinates (lines, rectangles, polygons, and portals) have a default -place which depends on the coordinates of the item. For a "center" anchor (the default), the place will be the center of the coordinates. Other items (that don't have coordinates) have a default of "0 0 1".
- "center"

  Center of screen. The object is positioned and sized so that its biggest dimension fills up 75% of the window, and it is centered. (This is dependent on the current view, and the current window dimensions.)

```
.pad ic 22 -place -150 150 1
```



```
.pad ic 22 -place "-50 20 0.5"
```



A synonym for the third (z) component of `-place`.

[13] -renderscript TclScript

(available only for all item types)

Specifies a Tcl script that will be evaluated every time the object is rendered. The script gets executed when the object normally would have been rendered. By default, the object will not get rendered. The script may call the renderitem function at any point to render the object. An example is:

```
.pad itemconfigure 22 -renderscript {
        puts "Before"
        .pad renderitem
        puts "After"
}
```

It would be possible to get in an endless render loop with the `-renderscript` option. If a

`-renderscript` callback triggers a render which causes that item to be redrawn, the system will be in an endless render loop. To avoid this problem, items do not implicitly trigger damage within a

`-renderscript` callback. If you do want to explicitly damage an item within a `-renderscript` callback, you must use the **damage** command. Be very careful to avoid infinite render loops.

[14] -sticky boolean

(available only for all item types)

Specifies if this item should be "sticky". Sticky items are rendered independent of the current view. That is, as the view pans and zooms, sticky items appear effectively stuck to the screen. All sticky items are rendered after non-sticky items, thus sticky items always are on top of non-sticky items. (See the **getview** and **moveto** commands.) Defaults to 0 (false).

[15] -tags tagList

(available only for all item types)

Specifies a set of tags to apply to the item. *TagList* consists of a list of tag names, which replace any existing tags for the item. *TagList* may be an empty list.

[16] -timerrate rate

(available only for all item types)

Specifies the frequency in milliseconds that the object's timerscript should be evaluated. If it is set to 0, the timer is turned off. Defaults to off (0). (see -timerscript).

[17] -timerscript TclScript

(available only for all item types)

Specifies a Tcl script that will be evaluated regularly, every rate milliseconds as specified by *-timerrate* (if *-timerrate* is greater than zero). This evaluation is independent of rendering and events. Returns the current TclScript for the object. (see *-timerrate*).

[18] -transparency value

(available only for all item types)

Specifies the transparency an item is drawn with. *value* must be a value between 0.0 and 1.0 where 0.0 is completely transparent and 1.0 is completely opaque. 1.0 is the default. If a portal or group is partially transparent, all of its member or visible objects, respectively, will have their transparency multiplied by the portals or groups.

[19] -viewscript TclScript

(available only for all item types)

Specifies a Tcl script that will be evaluated every time the view onto the Pad++ surface is changed. This script gets executed after the view coordinates have changed, but before the new scene gets rendered. Returns the current viewscript.

[20] -visible boolean

(available only for all item types)

WARNING: -visible is an obsolete option and will be removed in the next release. Replace all uses of the *-visible* option with *-transparency* which is more general.

Specifies whether this item is visible. Note that invisible items receive events and respond to commands such as **find**. Defaults to true.

[21] -width width

(available only for all item types)

By default, the width of every item is automatically computed based on its contents. If the -width option is set, however, then this overrides the automatically computed value. Items are centered within the specified *width*. If the dimensions are specified as smaller than the default values, the item is clipped to those dimensions. (Also see the -height itemconfigure option.)

[22] `-x` x

(available only for all item types)

A synonym for the first (x) component of `-place`.

[23] `-y` y

(available only for all item types)

A synonym for the second (y) component of `-place`.

[24] -zoomaction {size growScript shrinkScript}

(available only for all item types)

Specifies a pair of Tcl scripts that gets evaluated when an item grows or shrinks so that its size crosses size. This is a simple way of making "semantically zoomable" objects - that is, objects that look different when the are rendered at different sizes. When the item grows larger than size, *growScript* is evaluated, and when it shrinks smaller than size, *shrinkScript* is evaluated.

Any number of pairs of scripts may be associated with different sizes. Each use of *-zoomaction* may specify a different size, or modify scripts for an existing size. If both scripts are empty strings, then that zoomaction is deleted. This returns a list of zoomaction *size*, *growScript*, *shrinkScript* triplets.

The script gets executed when the object normally would have been rendered. By default, the object will not get rendered. The script may call the renderitem function at any point to render the object. See the description of *-renderscript* for an example. The deletion of items during a zoomaction is delayed until after the current render is finished.

Here is an example that turns a rectangle into an image when it is zoomed in, and back into the rectangle when zoomed out:

```
proc grow {} {
        .pad ic rect -visible 0
        .pad pushcoordframe rect
        set image_token [.pad allocimage images/unm_logo_orig.gif]
        .pad create image -image $image_token -anchor sw -tags "image"
        .pad popcoordframe
        .pad renderitem
}

proc shrink {} {
        .pad ic rect -visible 1
        set image_id [.pad find withtag image]
        if {$image_id != ""} {
                set image_token [.pad ic image -image]
                .pad freeimage $image_token
                .pad delete image
        }
        .pad renderitem
}

proc testzoomaction {} {
        .pad create rectangle 0 0 341 222 -pen black -fill yellow3 \
        -zoomaction {250 grow shrink} -tags "rect"
}
```

[25] `-z` z

(available only for all item types)

A synonym for the third (z) component of -place

Grid Items

Items of type grid arrange one or more items in rows and columns and treats them as a group. It is based on the Tk grid geometry manager and its behavior and Tcl syntax are very similar to it. In pad, all manipulations of a grid once it is created are affected through the `grid` sub-command. Note that rows and columns start from the top left corner of the grid (as in the Tk grid). The complete grid sub-command is described in this section.

Grids are created with widget commands of the following form:

```
pathName create grid [slaves...]
```

Grid creation is slightly different from creation of other pad objects. Instead of the normal command-line option-value pairs a list of slaves and their grid configuration can be specified (see the section below on sub-commands and slave configuration). Grids are special group objects and inherit much of the group functionality and support the "-divisible" option which can be set (using itemconfigure) once the grid is created:

[26] -divisible boolean

(available only for grid, group, and HTML item types)

Specifies whether events should go to the grid members. If *-divisible* is 1 (true), events never go to the grid object, but pass through it to the members. If the event is within the bounding box of the group, but does not hit any members, then it will be ignored by the group. If *-divisible* is 0 (false), then the event will go to the group if it is within the bounding box of the group whether there is a member at the place the event points to or not. Defaults to 1 (true).

The syntax of the grid sub-command is:

```
pathNname grid slave [slave...] option value [option value...]
pathName grid command arg [arg...]
```

If the first argument of the grid command is a slave object then the remainder of the command line is processed in the same way as the grid configure command. The "-in" option can be used to add a slave to a grid. The following grid sub-commands are allowed:

```
$PAD grid arrange master
```

Forces arrangement of the given grid. Any pending layout request for the grid is removed. This can be useful when an application has done several grid configuration and wants them to take effect immidiately. Normally, grid arrangement is done at "idle" times.

```
$PAD grid bbox master column row
```

The bounding box (in pixels) is returned for the space occupied by the grid position indicated by column and row. The return value consists of 4 integers. The first two are the pixel offset from the master window (x then y) of the top-left corner of the grid cell, and the second two are the width and height of the cell.

```
$PAD grid columnconfigure master index [-option value...]
```

Query or set the column properties of the index column of the geometry master, master. The valid options are -minsize and -weight. The -minsize option sets the minimum column size, in screen units, and the -weight option (a floating point value) sets the relative weight for apportioning any extra spaces among columns. If no value is specified, the current value is returned.

```
$PAD grid configure slave [slave ...] [options]
```

The arguments consist of one or more slaves followed by pairs of arguments that specify how to manage the slaves. The characters -, x and ^, can be specified instead of a window name to alter the default location of a slave, as described in the ``RELATIVE PLACEMENT" section, below. If any of the slaves are already managed by the grid then any unspecified options for them retain their previous values rather than receiving default values. The following options are supported:

-column n

Insert the slave so that it occupies the nth column in the grid. Column numbers start with 0. If this option is not supplied, then the slave is arranged just to the right of previous slave specified on this call to grid, or column "0" if it is the first slave. For each x that immediately precedes the slave, the column position is incremented by one. Thus the x represents a blank column for this row in the grid.

-columnspan n

Insert the slave so that it occupies n columns in the grid. The default is one column, unless the slave is followed by a -, in which case the columnspan is incremented once for each immediately following -.

-in other

Insert the slave(s) in the grid object given by other (which must be an existing grid).

-padx amount

The amount specifies how much horizontal external padding to leave on each side of the slave(s). The amount defaults to 0.

-pady amount

The amount specifies how much vertical external padding to leave on the top and bottom of the slave(s). The amount defaults to 0.

-row n

Insert the slave so that it occupies the nth row in the grid. Row numbers start with 0. If this option is not supplied, then the slave is arranged on the same row as the previous slave specified on this call to grid, or the first unoccupied row if this is the first slave.

-rowspan n

Insert the slave so that it occupies n rows in the grid. The default is one row. If the next grid command contains ^ characters instead of slaves that line up with the columns of this slave, then the rowspan of this slave is extended by one.

-sticky style

If a slave's parcel is larger than its requested dimensions, this option may be used to position (or stretch) the slave within its cavity. Style is a string that contains zero or more of the characters n, s, e or w. The string can optionally contains spaces or commas, but they are ignored. Each letter refers to a side (north, south, east, or west) that the slave will "stick" to. If both n and s (or e and w) are specified, the slave will be stretched to fill the entire height (or width) of its cavity. The sticky option subsumes the combination of -anchor and -fill that is used by pack. The default is {}, which causes the slave to be centered in its cavity, at its requested size.

```
$PAD grid forget slave [slave ...]
```

Removes each of the slaves from their grid.

```
$PAD grid info slave
```

Returns a list whose elements are the current configuration state of the slave given by slave in the same option-value form that might be specified to grid configure. The first two elements of the list are ``-in master'' where master is the slave's master.

```
$PAD grid location master x y
```

Given x and y values in screen units relative to the master object, the column and row number at that x and y location is returned. For locations that are above or to the left of the grid, -1 is returned.

```
$PAD grid rowconfigure master index [-option value...]
```

Query or set the row properties of the index row of the geometry master, master. The valid options are -minsize and -weight. Minsize sets the minimum row size, in screen units, and weight sets the relative weight for apportioning any extra spaces among rows. If no value is specified, the current value is returned.

```
$PAD grid size master
```

Returns the size of the grid (in columns then rows) for master. The size is determined either by the slave occupying the largest row or column, or the largest column or row with a minsize or weight.

```
$PAD grid slaves master [-option value]
```

If no options are supplied, a list of all of the slaves in master are returned. Option can be either -row or -column which causes only the slaves in the row (or column) specified by value to be returned.

Relative Placement

The grid command contains a limited set of capabilities that permit layouts to be created without specifying the row and column information for each slave. This permits slaves to be rearranged, added, or removed without the need to explicitly specify row and column information.

When no column or row information is specified for a slave, default values are chosen forcolumn, row, columnspan and rowspan at the time the slave is managed. The values are chosen based upon the current layout of the grid, the position of the slave relative to other slaves in the same grid command, and the presence of the characters -, ^, and ^ in grid command where slave names are normally expected.

- This increases the columnspan of the slave to the left. Several -'s in a row will successively increase the columnspan. S - may not follow a ^ or a x.

x This leaves an empty column between the slave on the left and the slave on the right.

^ This extends the rowspan of the slave above the ^'s in the grid. The number of ^'s in a row must match the number of columns spanned by the slave above it.

Restrictions on Master Windows

In pad, the master for each slave is the slave's parent (which is a grid object). This means if an object belongs to an existing group then it cannot be added to a grid.

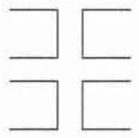
Differences Between Pad++ and TK Grid Commands

- The *-ipadx* and *-ipady* grid item configuration options are not available in pad.
- Master window geometry propagation flag is not available in pad.
- The parent-child and stacking restrictions and rules for master and slave items are not supported in pad (slaves can only be in the master group).
- If the grid is not positioned then it places itself around its first item. Once all grid items have been positioned the grid bounding box will be computed to enclose them all.
- Added the `arrange` command for forcing grid arrangement.
- Items that are removed from grids are not unmapped.

Examples

1) put four objects in a 2x2 grid with 10 pixels horizontal and vertical pading:

```
set obj1 [.pad create rectangle 0 0 50 50]
set obj2 [.pad create rectangle 50 50 100 100]
        set obj3 [.pad create rectangle 100 100 150 150]
set obj4 [.pad create rectangle 150 150 200 200]
set thegrid [.pad create grid $obj1 $obj2 -padx 10 -pady 10]
.pad grid $obj3 $obj4 -in $thegrid -row 1 -padx 10 -pady 10
```



2) read objects from pad files in a directory and place them in a Nx2 grid (this can be useful for creating palettes):

```
proc read_files {PAD dir} {
                set objs ""
```

```
                                # Go though list of files
                foreach file [glob $dir/*.pad] {
                                # Read file and put all its object in a group (Pad_ObjectList will be
                                # set to list of objects read from file).
                        $PAD read $file
                        set group [$PAD create group -members $Pad_ObjectList]
                        lappend objs $group
                }
                return $objs
}

proc create_palette {PAD objs} {
                                # Create the grid object
                set thegrid [$PAD create grid]
                set row 0
                set col 0

                                # Go through objects and place them two per row
                foreach obj $objs {
                                # Add obj to the grid
                        $PAD grid $obj -in $thegrid -row $row -column $col -padx 10 -pady 5

                                # Set row and column position for next object
                        if {$col == 0} {
                                incr col
                        } else {
                                set col 0
                                incr row
                        }
                }

                                # Have the grid arrange itself now
                $PAD grid arrange $thegrid

                return $thegrid
}

        create_palette .pad [read_files .pad $env(PADHOME)/draw/scrapbook]
```

Alternatively,

```
proc create_palette {PAD objs} {
                # create the grid object
                set thegrid [$PAD create grid]

                # go through list of objects and place them two per row
                set numobjs [llength $objs]
                for {set i 0} {$i < $numobjs} {incr i 2} {
                        set obj1 [lindex $objs $i]
                        if {$i < [expr $numobjs-1]} {
                                set obj2 [lindex $objs [expr $i+1]]
                        } else {
                                set obj2 ""
                        }
                        $PAD grid $obj1 $obj2 -in $thegrid -padx 10 -pady 5
                }

                $PAD grid arrange $thegrid
                return $thegrid
}

        create_palette .pad [read_files .pad $env(PADHOME)/draw/scrapbook]
```

3) Draw horizontal and vertical grid lines and a bounding rectangle for an existing grid. Make a group for the line objects and the existing grid. Assume the grid is a normal MxN table (i.e. all rows have N columns and all columns have M rows).

```
proc create_gridlines { PAD thegrid } {
                                        # Get bounding box, width and height and location of the grid
                set gbbox [$PAD bbox $thegrid]
                set gwidth [expr [lindex $gbbox 2] - [lindex $gbbox 0]]
                set gheight [expr [lindex $gbbox 3] - [lindex $gbbox 1]]
                set gx [lindex $gbbox 0]
                set gy [lindex $gbbox 1]

                                        # Get number of rows and columns
                set numrows [lindex [$PAD grid size $thegrid] 1]
                set numcols [lindex [$PAD grid size $thegrid] 0]
```

```
                                        # Create the bounding rectangle
            set grect [eval $PAD create rectangle $gbbox]

            set items "$grect"
            set scale [$PAD scale $thegrid]

                                        # Create horizontal lines by looking at the <r, 0> grid elemments.
            for {set r 1} {$r < $numrows} {incr r} {
                                        # Get location of the <r, 0> element (including padding)
                    set rinfo [$PAD grid bbox $thegrid 0 $r]
                    set x1 [expr [lindex $rinfo 0]*$scale + $gx]
                                        # Transform the y coord for pad (grid's is from top left corner)
                    set y1 [expr ($gheight - [lindex $rinfo 1]*$scale) + $gy]
                    set x2 [expr $x1 + $gwidth]
                    set y2 $y1
                    lappend items [$PAD create line $x1 $y1 $x2 $y2 -tags gridrowline_$thegrid]
            }

                                        # Draw vertical lines by looking at the <0, c> elements
            for {set c 1} {$c < $numcols} {incr c} {
                    set cinfo [$PAD grid bbox $thegrid $c 0]
                    set x1 [expr [lindex $cinfo 0]*$scale + $gx]
                    set y1 [expr ($gheight - [lindex $cinfo 1]*$scale) + $gy]
                    set x2 $x1
                    set y2 [expr $y1 - $gheight]
                    lappend items [$PAD create line $x1 $y1 $x2 $y2 -tags gridcolline_$thegrid]
            }

                                        # Create a group for all the grid lines
            set glines [$PAD create group -members $items -divisible 0 \
                                         -tags gridlines_$thegrid]

                                        # Create a group for the lines and the grid
            set newgrp [$PAD create group -members "$glines $thegrid" -tags grid_$thegrid \
                                        -divisible 1]

            return $newgrp
}

        set thegrid [create_palette .pad [read_files .pad ./draw/scrapbook]]
create_gridlines .pad $thegrid
```

Group Items

Items of type group are special items that group other items. Group items do not have any visual appearance, but rather are used just for creating structure. Groups are implemented very efficiently, and may be hierarchical (i.e., contain other groups). Modifying the position of a group implicitly affects all of the members of the group, recursively. Pad++ also supports "tags" which are implicit way of grouping items - but this only works for events. That is, giving several items the same tag allows them all to respond to the same event handlers. Groups explicitly bring items together. Group members are rendered sequentially in the display list. That is, no other objects can appear inbetween group members - they are always above or below all the group members. Raising or lowering a group object raises or lowers all the group members. Raising or lowering a group member raises or lowers the member within the group.

Groups automatically resize themselves to contain all of their members - thus adding, removing, or repositioning a member implicitly changes the size of the group. See the pad `addgroupmember` and `removegroupmember` commands and the *-member* itemconfigure option below for setting group membership, and the `getgroup` command for testing group membership.

When an event hits a group, it normally passes through the group object to its members. However, it is possible to configure a group object so that it grabs the events and does not pass them through. See the *-divisible* flag.

Groups are created with widget commands of the following form:

```
pathName create group [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for groups:

[27] -divisible boolean

(available only for grid, group, and HTML item types)

Specifies whether events should go to group members. If *-divisible* is 1 (true), events never go to the group object, but pass through it to the members. If the event is within the bounding box of the group, but does not hit any members, then it will be ignored by the group. If *-divisible* is 0 (false), then the event will go the group if it is within the bounding box of the group whether there is a member at the place the event points to or not. Defaults to 1 (true).

[28] -members members

(available only for group and HTML item types)

members is a list of object ids that specify the list of members of this group. Setting the members of a group first removes all existing members, and then inserts the new members. The members are rendered in the order they are specified in *members*.

Handle Items

Items of type handle are special items that are designed to be used for selection handles around items when manipulating them within an application. Handles are similar to rectangles, except unlike every other item in Pad++, handles do *not* zoom. Handles are always rendered at the same size. Handles, however, do slide around like other items. They just don't zoom. Handles default to being five by five pixels, but this can be changed with the `-width` and `-height` itemconfigure options.

Handles are created with widget commands of the following form:

```
pathName create handle [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in `itemconfigure` widget commands to change the item's configuration. The following options are supported for handles:

[29] -fill color

(available only for handle, HTML, Polygon, Portal and Rectangle item types)

Fill the background of the item with *color*, which may be specified in any of the forms accepted by Tk_GetColor. If *color* is "none", the background will not be drawn. It defaults to the red.

[30] -pen color

(available only for handle, line, polygon, portal, rectangle, spline, text and textfile item types)

Color specifies a color to use for drawing the item; it may have any of the forms acceptable to Tk_GetColor. It may also be "none", in which case the outline will not be drawn. This option defaults to black.

HTML Items

Items of type html are compound items representing the specified html file. (HTML is HyperText Markup Language. Based on SGML, HTML is most commonly known as the language describing items for the World-Wide Web.) HTML items know about the internet and will automatically fetch a file from a URL (Universal Resource Locator) as well as in-line images. URL's may also specify local files. When the html data is fetched, it is parsed and the HTML item is created which contains a method for rendering the page. HTML anchors are created as separate items which may have events bound to them. HTML items are an extension of `group` items, and thus have several of the same options as groups.

There is a Tcl file (*draw/html.tcl*) which describes default event bindings for html items which follow hyperlinks, and lay them out with scale.

See the end of the description of HTML items for a description of html anchors.

HTML items are created with widget commands of the following form:

```
pathName create html [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for html items:

[31] -border color

(available only for HTML and portal item types)

Color specifies a color to use for drawing the border of the portal; it may have any of the forms accepted by Tk_GetColor. If *color* is "none", the outline will not be drawn. This option defaults to the fill color.

[32] -borderwidth width

(available only for HTML and Portal item types)

Width specifies the width of the border in current units to be drawn around the item. Wide borders will be drawn completely inside the path specified by the points of this object. Note that this is different than pens. If *width* is 0, then the border will always be drawn one pixel wide, independent of the zoom. *Width* defaults to 1 pixel.

[33] -divisible boolean

(available only for grid, group, and HTML item types)

If true, then events go through the HTML object to its anchors. If false, events stop at the HTML object, and never go through to the anchors. Defaults to true.

[34] -donescript script

(available only for HTML item types)

If *script* is specified, it gets evaluated when the html item has completed loading - including all in-line images. *script* is postpended with the id of the html object. This is necessary because the script is typically specified on the create line where the id of the html object is not yet known.

[35] -errorscript script

(available only for HTML item types)

If *script* is specified, it gets evaluated if there is an error creating the html item. An error can occur for many reasons - especially because creating an html typically starts a network communication process for fetching the URL. *script* is postpended with the id of the html object. This is necessary because the script is typically specified on the create line where the id of the html object is not yet known.

[36] -fill color

(available only for handle,HTML, polygon, portal and rectangle item types)

Fill the background of the html item with *color*, which may be specified in any of the forms accepted by Tk_GetColor. If *color* is "none", the background will not be drawn. It defaults to the background of the Pad++ widget it is created on.

[37] -font fontname

(available only for HTML, portal, text and textfile item types)

Specifies the font to be used for rendering text for this item. fontname must specify a filename which contains an Adobe Type 1 font, or the string "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`".

[38] -htmlanchors

(available only for HTML item types)

Returns all the anchors that are part of this HTML item. This is a read-only option, and may not be set.

[39] -members

(available only for group and HTML item types)

Because an HTML item is a group, it may contain other members in addition to its anchors. This allows setting and retrieving of all members that are part of this HTML item.

[40] -updatescript script

(available only for HTML item types)

If *script* is specified, it gets evaluated when the html source has loaded, and then once every time an in-line is loaded. *script* is postpended with the id of the html object. This is necessary because the script is typically specified on the create line where the id of the html object is not yet known.

[41] -url urlname

(available only for HTML item types)

Specifies the URL (Universal Resource Locator, or World-Wide Web address) that this html page should be accessed from. It must be specified with a valid address. Some examples are: "http://www.unm.edu", "http://www.cs.unm.edu/bederson", "file://nfs/u3/bederson/public_html/begin.html", "home-page.html".

[42] -width width

(available only for all item types)

Specifies the width (in the current units) of the html page. The page will be re-laid out according to the new width, and the length of the page may change dependent on the new width.

HTML ANCHORS

The anchors are special Pad++ items of type "htmlanchor". They are automatically grouped with the HTML object. As such, they can not be deleted independently, and are automatically deleted when the html object they are associated with is deleted. Some anchors have multiple components (i.e., and image and some text). In this case, they all have the same URL, and changing the pen color of one component automatically changes the pen color of the other components.

Anchors may be configured with the itemconfigure command. The following options are supported for html anchors:

[43] -html

(available only HTML anchors item types)

Returns the html item this anchor belongs to. This is a read-only option.

[44] -ismap

(available only HTML anchors item types)

Returns true if this anchor is an imagemap. This is a read-only option.

[45] -state state

(available only HTML anchors item types)

Specifies the state of the anchor (which controls its color). There is no direct control over an anchor's color. Rather, it uses the default colors unless the HTML page specifies anchor colors. *State* may be one of "unvisited", "active", "visited", or "notloaded". In-line images that haven't been loaded yet are

"notloaded".

[46] -url

(available only HTML anchors item types)

Returns the URL that this anchor addresses. This is a read-only option.

Image Items

Items of type image appear on the display as color images. Images are created with widget commands of the following form:

```
pathName create image [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for images:

[47] -dither dithermode

(available only for image item types)

Specifies if and when the image is rendered with dithering. Dithering is a rendering technique that allows closer approximation to the actual image colors, even when the requested colors are not available. Rendering images with dithering is much slower than without, so this option allows control as to when (if at all), dithering is used. dithermode may be any of

-
- `nodither`: The image is never rendered with dithering.
- `dither`: The image is always rendered with dithering.
- `refinedither`: The image is initially rendered without dithering, and then refined with dithering.

Defaults to *refinedither* (dither only on refinement).

[48] -image imagetoken

(available only for image item types)

Specifies the image that this item will render. (Also see the `allocimage` and `freeimage` commands.)

KPL Items

Items of type Kpl provide a method for creating an item with a user-described render method. Sometimes the Pad++ items available do not have exactly what you want, or you'd like a complex item consisting of several primitives. Rather than create several different Pad++ items and group them together, a single Kpl item can be created with a kind of display list.

Kpl is a language (designed at New York University by Ken Perlin, et. al.) that is very simple, but extremely fast. It is the best language we found for writing interpreted code for rendering quickly. In fact, Kpl has a byte-compiler which makes it faster. Some simple experiments have shown it to be roughly 15 times slower than C for simple math (compared to tcl which is typically about 1,000 times slower than C). Because Kpl is a general-purpose language, it can be used for on-the-fly calculations as well as render calls. Pad++ supplies several render that available through Kpl that allow a Kpl object to render fairly complex objects.

Kpl is a stack-based post-fix language (much like PostScript). Some basic documentation is available with the Pad++ release in *doc/kpl.troff.* See the section in this document on the KPL-PAD++ INTERFACE for a description of how to access Kpl through Pad++, and what Pad++ routines are available from Kpl.

Kpl items are created with widget commands of the following form:

```
pathName create kpl [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following special options are supported for kpl objects:

[49] -renderscript kpl_script

(available only for all item types)

This is like the standard `-renderscript` option available to all items, but in this case, the string specifies a Kpl script instead of a Tcl script.

[50] -bb boundingboxScript

(available only for KPL and TCL item types)

A Kpl script that will be evaluated to compute the bounding box of this item. It should return two-element vectors that specify (x1, y1), (x2, y2) which are the lower left and upper right corners of this items bounding box.

Note that all coordinates in Kpl are specified in pixels, and not in the current Pad++ units. An example follows that creates a Kpl item that draws a brown triangle. In this case, the Kpl code is stored in the file triangle.kpl.

```
# Tcl code to load Kpl code and to create
# Pad++ Kpl item that draws a brown triangle
```

```
kpl eval 'triangle.kpl source
set pen [.pad alloccolor brown]
.pad create kpl -bb {-10:-10 110:110} -renderscript {draw_triangle}

                                   /* Kpl code (in a separate file)
                                      to draw a brown triangle */
{
        'pen tcl_get -> Pen
        Pen setcolor
        3 setlinewidth
        newpath
                0:0 moveto
                100:0 lineto
                50:100 lineto
                0:0 lineto
        stroke
} -> draw_triangle
```

Line Items

Items of type line appear on the display as one or more connected line segments. Lines are created with widget commands of the following form:

```
pathName create line x1 y1... xn yn [option value option value ...]
```

The arguments x1 through yn give the coordinates for a series of two or more points that describe a series of connected line segments. After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for lines:

[51] -arrow where

(available only for line and spline item types)

Indicates whether or not arrowheads are to be drawn at one or both ends of the line. *where* must have one of the values "none" (for no arrowheads), "first" (for an arrowhead at the first point of the line), "last" (for an arrowhead at the last point of the line), or "both" (for arrowheads at both ends). This option defaults to "none".

[52] -arrowshape shape

(available only for line and spline item types)

This option indicates how to draw arrowheads. The *shape* argument must be a list with three elements, each specifying a distance. The first element of the list gives the distance along the line from the neck of the arrowhead to its tip. The second element gives the distance along the line from the trailing points of the arrowhead to the tip, and the third element gives the distance from the outside edge of the line to the trailing points. If this option isn't specified then Pad++ picks a "reasonable" shape.

[53] -capstyle cap

(available only for line and spline item types)

Specifies how the ends of the line are drawn. *cap* may be one of:

-
- `butt`: The ends are drawn square, at the end point.
- `projecting`: The ends are drawn square, past the endpoint.
- `round`: The ends are rounded.

[54] -joinstyle join

(available only for line, polygon, rectangle and spline item types)

Specifies how the joints at vertices are drawn. *join* may be one of:

-
- `bevel`: The joints are drawn without protruding. They are cut-off and sharp.
- `miter`: The joints are drawn protruding to a point.
- `round`: The joints are rounded.

[55] -noisedata noisedata

(available only for line item types)

Specifies the noise parameters used to make rough-looking lines. `noisedata` is a four element list of numbers of the form:

"Pos Freq Amp Steps"

Rough lines are generated using the Perlin noise function. The Perlin noise function is like a sin function with a very irregular amplitude - like sin, noise has a constant period (one), but no two segments of the noise curve are alike. Noisy lines are generated by adding noise to the tangent direction of a line.

In the current implementation, there are four noise parameters: Pos, Freq, Amp, and Steps. Pos determines what part of the noise curve is sampled for that object. Freq determines the rate of sampling, Amp indicates the level, and Steps indicates how many samples to introduce per line segment. The drawing algorithm is straightforward. For each line segment, coordinates are generated as follows:

```
DrawRoughLine(x1, y1, x2, y2, Pos, Freq, Amp, Steps) :
        step  = 1.0/Steps;
        mag   = length(x1,y1,x2,y2);
        theta = direction(x1,y1,x2,y2);

        xmag = Amp * sin(theta) * mag;
        ymag = Amp * cos(theta) * mag;

        vertex(x1, y1);

        for (a = step; a < steps; a += step) {
                n = noise(Pos);
                vertex(lerp(a,x1,x2) + n*xamp, lerp(a,y1,y2) + n*yamp);
                Pos += Freq;
        }
        vertex(x2, y2);
```

Note that we multiply Amp by mag, the length of the line. This is necessary in Pad++ since the zooming functionality means that lines can be of nearly any size. Making the level of noise proportional to the length of the line keeps the informality uniform at all sizes. (We should probably also modulate the number of points generated by the thickness of the line, so small thin lines are cheap).

Values of 0.3 for Freq, 0.1 for Amp, 10 for Steps produces pleasant-looking lines. Pos can be an arbitrary floating point number - giving different objects unique values for Pos ensures that each object has a different appearance.

[56] -pen color

(available only for handle, line, polygon, portal, rectangle, spline, text and textfile item types)

Color specifies a color to use for drawing the line; it may have any of the forms acceptable to Tk_GetColor. It may also be "none", in which case the line will not be drawn. This option defaults to black.

[57] -penwidth width

(available only for line, polygon, rectangle and spline item types)

Width specifies the width of the pen in current units to be drawn around the item. Wide lines will be drawn centered on the path specified by the points. If *width* is 0.0, then the pen will always be drawn one pixel wide, independent of the zoom. *Width* defaults to 1 pixel.

Pad Items

Each pad widget implicitly defines a special "pad" item which always has the id "1". This is a special item which can get events and has a few itemconfigure options. It may not be explicitly created or deleted. The valid options are:

[58] -visiblelayers layers

(available only for pad and portal item types)

Specifies what layers are visible within this portal. *layers* can be either a list of layers which will specify which items will be displayed within this portal, or take the special form of "all -layer1 -layer2 -layer3 ..." in which case all layers except the ones specified will be displayed. Defaults to "all". (See the *-layer* itemconfigure option that all items have.)

Polygon Items

Items of type polygon appear as polygonal regions on the display. Each polygon may have an outline (pen color), a fill, or both. Polygon are created with widget commands of the following form:

```
pathName create polygon x1 y1... xn yn [option value option value ...]
```

The arguments x1, y1, ..., xn, and yn specify the coordinates of the vertices of the polygon. After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for polygons:

[59] -fill color

(available only for handle, HTML, polygon, portal and rectangle item types)

Fill the area of the polygon with color, which may be specified in any of the forms accepted by Tk_GetColor. If color is "none", (the default), then the polygon will not be filled.

[60] -joinstyle join

(available only for line, polygon, rectangle and spline item types)

Specifies how the joints at vertices are drawn. *join* may be one of:

-
- `bevel`: The joints are drawn without protruding. They are cut-off and sharp.
- `miter`: The joints are drawn protruding to a point.
- `round`: The joints are rounded.

[61] -pen color

(available only for handle, line, polygon, portal, rectangle, spline, text and textfile item types)

Draw an outline around the edge of the polygon in color. Color may have any of the forms accepted by Tk_GetColor. If color is "none", then no outline will be drawn for the rectangle. This option defaults to black.

[62] -penwidth width

(available only for line, polygon, rectangle and spline item types)

Width specifies the width of the pen in current units to be drawn around the item. Wide lines will be drawn centered on the path specified by the points. If *width* is 0.0, then the pen will always be drawn one pixel wide, independent of the zoom. *Width* defaults to 1 pixel.

Portal Items

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, a 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. The surface that the portal is looking onto is called that portal's *lookon*. Portal items are created with widget commands of the following form:

```
pathName create portal x1 y1 x2 y2 ... [option value option value ...]
```

If two points are specified, then the portal will be rectangular where those two points specify the lower left and upper right coordinates of the portal. If more than two points are specified, then the portal will be polygonal shaped by those points. There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for text items:

[63] -border color

```
(available only for HTML and portal item types)
```

Color specifies a color to use for drawing the border of the portal; it may have any of the forms accepted by Tk_GetColor. If *color* is "none", the outline will not be drawn. This option defaults to the fill color.

[64] -borderwidth width

`(available only for HTML and portal item types)`

Width specifies the width of the border in current units to be drawn around the item. Wide borders will be drawn completely inside the path specified by the points of this object. Note that this is different than pens. If *width* is 0, then the border will always be drawn one pixel wide, independent of the zoom. *Width* defaults to 1 pixel.

[65] -fill color

(available only for handle, HTML, polygon, portal and rectangle item types)

Fill the background of the portal with color, which may be specified in any of the forms accepted by Tk_GetColor. If *color* is "none", the background will not be drawn. It defaults to the background of the Pad++ widget it is created on.

[66] -font fontname

(available only for HTML, portal, text and textfile item types)

Specifies the font to be used for rendering text for this item. fontname must specify a filename which contains an Adobe Type 1 font, or the string "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`".

[67] -lookon surface

(available only for portal item types)

Specifies which Pad++ surface this portals looks onto. surface should be the complete pathName of a Pad++ widget. Defaults to the surface the portal was created on.

[68] -pen color

(available only for handle, line, polygon, portal, rectangle, spline, text and textfile image types)

color specifies the text color of the title. If color is "none", then no outline will be drawn for the rectangle. This option defaults to either black or white - whichever contrasts the most with the fill color.

[69] -relief relief

(available only for portal item types)

Specifies the relief to be used by the border of this item. *relief* may be any of: *raised*, *sunken*, *flag*, *ridge*, or *groove*. Defaults to "*ridge*"

[70] -title title

(available only for portal item types)

If *title* is specified, then the portal will be rendered with a titlebar consisting of *title*. Otherwise, no title bar is drawn. Defaults to the empty string.

[71] -view place

(available only for portal item types)

Specifies the (x, y, zoom) location this portal looks onto. Like a Pad++ widget, place specifies the point rendered at the center of the portal and the magnification. Defaults to directly under the location the portal was created at.

[72] -visiblelayers layers

(available only for pad and portal item types)

Specifies what layers are visible within this portal. *layers* can be either a list of layers which will specify which items will be displayed within this portal, or take the special form of "all -layer1 -layer2 -layer3 ..." in which case all layers except the ones specified will be displayed. Defaults to "all". (See the *-layer* itemconfigure option that all items have.)

Rectangle Items

Items of type rectangle appear as rectangular regions on the display. Each rectangle may have an outline (pen color), a fill, or both. Rectangles are created with widget commands of the following form:

```
pathName create rectangle x1 y1 x2 y2 [option value option value ...]
```

The arguments x1, y1, x2, and y2 give the coordinates of two diagonally opposite corners of the rectangle After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for rectangles:

[73] -fill color

(available only for handle, HTML, polygon, portal and rectangle item types)

Fill the area of the rectangle with color, which may be specified in any of the forms accepted by Tk_GetColor. If *color* is "none" (the default), then the rectangle will not be filled.

[74] -joinstyle join

(available only for line, polygon, rectangle and spline item types)

Specifies how the joints at vertices are drawn. *join* may be one of:

-
- `bevel`: The joints are drawn without protruding. They are cut-off and sharp.
- `miter`: The joints are drawn protruding to a point.
- `round`: The joints are rounded.

[75] -pen color

(available only for handle, line, polygon, portal, rectangle, spline, text and textfile item types)

Draw an outline around the edge of the rectangle in color. *Color* may have any of the forms accepted by Tk_GetColor. If *color* is "none", then no outline will be drawn for the rectangle. This option defaults to black.

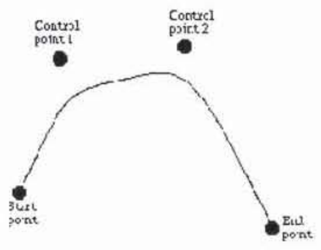
[76] -penwidth width

(available only for line, polygon, rectangle and spline item types)

Width specifies the width of the pen in current units to be drawn around the item. Wide lines will be drawn centered on the path specified by the points. If *width* is 0.0, then the pen will always be drawn one pixel wide, independent of the zoom. *Width* defaults to 1 pixel.

Spline Items

Items of type spline appear on the display as one or more bezier curves joined end to end, so the last point of the one curve is used as the first point of the next. Splines are displayed as smooth curves at any magnification. They are rendered in more detail when they are larger. It is possible to create a fixed approximation to a spline with the `spline2line` command. In addition, it is possible to generate a spline that approximates a multi-segmented line with the `line2spline` command. A bezier curve is defined using four points - the start and end point for the curve, and two control points that indicate the path that the curve follows. For example:



For a spline made from a single bezier segment, the points are given as follows:

<start-x> <start-y> <c1-x> <c1-y> <c2-x> <c2-y> <end-x> <end-y>

That is, first the start point is given, followed by the first control point, followed by the second control point and finishing with the end point for the curve. For example, you can create a simple spline using:

.pad create spline 0 0 10 10 20 10 30 0

here (0, 0) defines the start of the curve. (10, 10) is the first control point, (20, 10) is the second control point, and the curve ends at (30, 0).

Splines are created with widget commands of the following form:

```
pathName create spline x1 y1... xn yn [option value option value ...]
```

The arguments x1 through yn give the coordinates for a series of one or more splines. Each point is specified by two coordinates. When specifying a spline made from two or more bezier curves, the end point of the first curve is used as the start point for the second, so the second curve only requires an additional three points (two control points and an end point). In general a spline of N bezier curves requires 3N+1 points (6N+2 coordinates). This represents a start point and then three points for each curve.

For convenience, if the end point of the last curve segment in a spline is omitted, Pad++ assumes that the curve should be 'closed' - it uses the start point of the first curve as the end point for the last curve, creating a closed shape. For closed shapes, therefore, you should provide 3N points (6N coordinates).

After the coordinates there may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for lines:

[77] -arrowshape shape

(available only for line and spline item types)

This option indicates how to draw arrowheads. The *shape* argument must be a list with three elements, each specifying a distance. The first element of the list gives the distance along the spline from the neck of the arrowhead to its tip. The second element gives the distance along the spline from the trailing points of the arrowhead to the tip, and the third element gives the distance from the outside edge of the spline to the trailing points. If this option isn't specified then Pad++ picks a "reasonable" shape.

[78] -arrow where

(available only for line and spline item types)

Indicates whether or not arrowheads are to be drawn at one or both ends of the spline. *where* must have one of the values "none" (for no arrowheads), "first" (for an arrowhead at the first point of the line), "last" (for an arrowhead at the last point of the line), or "both" (for arrowheads at both ends). This option defaults to "none".

[79] -capstyle cap

(available only for line and spline item types)

Specifies how the ends of the spline are drawn. *cap* may be one of:

-
- `butt`: The ends are drawn square, at the end point.
- `projecting`: The ends are drawn square, past the endpoint.
- `round`: The ends are rounded.

[80] -joinstyle join

(available only for line, polygon, rectangle and spline item types)

Specifies how the joints at vertices are drawn. *join* may be one of:

-
- `bevel`: The joints are drawn without protruding. They are cut-off and sharp.
- `miter`: The joints are drawn protruding to a point.
- `round`: The joints are rounded.

[81] -pen color

(available only for handle, line, polygon, portal, rectangle, spline, text and textfile item types)

Color specifies a color to use for drawing the spline; it may have any of the forms acceptable to Tk_GetColor. It may also be "none", in which case the line will not be drawn. This option defaults to black.

[82] -penwidth width

(available only for line, ploygon, rectangle and spline item types)

Width specifies the width of the pen in current units to be drawn around the item. Wide lines will be drawn centered on the path specified by the points. If *width* is 0.0, then the pen will always be drawn one pixel wide, independent of the zoom. *Width* defaults to 1 pixel.

TCL Items

Items of type tcl are really a simple of way of having user-describable item. A Tcl item really consists of two Tcl scripts to render an item procedurally (one to render, and the other to compute the bounding box.) The render script can render by calling the pad widget with the various drawing routines (see `drawline`, `drawtext`, `setcolor`, `setlinewidth`.) Tcl's are created with widget commands of the following form:

```
pathName create tcl [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for tcl objects:

[83] -bb boundingboxScript

(available only for KPL and TCL item types)

A Tcl script that will be evaluated to compute the bounding box of this item. It should return a 4 element list whose members are "x1 y1 x2 y2" which are the lower left and upper right corners of this items bounding box.

Text Items

A text item displays a string of characters on the screen in one or more lines. There is a single custom "vector" font. Text items are created at a default size of one pixel high. Their size can be changed with the `scale` command or the *-place* itemconfigure option.

INDICES

Many of the commands for text take one or more indices as arguments. An index is a string used to indicate a particular place within a text, such as a place to insert characters or one endpoint of a range of characters to delete. Indices have the syntax:

```
base modifier modifier modifier ...
```

Where *base* gives a starting point and the modifiers adjust the index from the starting point (e.g. move forward or backward one character). Every index must contain a base, but the modifiers are optional.

The base for an index must have one of the following forms:

```
line.char
```

Indicates *char*'th character on line *line*. Lines are numbered from 0. Notice that this is different than the Tk text widget. Within a line, characters are numbered from 0.

```
line.end
```

Indicates the last character on line *line*. Lines are numbered from 0.

```
char
```

Indicates the *char*'th character from the beginning of the file (starting at 0).

```
@x,y
```

Indicates the character that covers the pixel whose *x* and *y* coordinates within the text's window are *x* and *y*.

```
end
```

Indicates the last character in the text.

```
mark
```

Indicates the character just after the mark whose name is *mark*.

If modifiers follow the base index, each one of them must have one of the forms listed below. Keywords such as chars and wordend may be abbreviated as long as the abbreviation is unambiguous. Modifiers must have one of the following forms:

```
+ count chars
```

Adjust the index forward by count characters, moving to later lines in the text if necessary. If there are fewer than count characters in the text after the current index, then set the index to the last character in the text. Spaces on either side of count are optional.

```
- count chars
```

Adjust the index backward by count characters, moving to earlier lines in the text if necessary. If there are fewer than count characters in the text before the current index, then set the index to the first character in the text. Spaces on either side of count are optional.

```
+ count lines
```

Adjust the index forward by count lines, retaining the same character position within the line. If there are fewer than count lines after the line containing the current index, then set the index to refer to the same character position on the last line of the text. Then, if the line is not long enough to contain a character at the indicated character position, adjust the character position to refer to the last character of the line. Spaces on either side of count are optional.

```
- count lines
```

Adjust the index backward by count lines, retaining the same character position within the line. If there are fewer than count lines before the line containing the current index, then set the index to refer to the same character position on the first line of the text. Then, if the line is not long enough to contain a character at the indicated character position, adjust the character position to refer to the last character of the line. Spaces on either side of count are optional.

```
linestart
```

Adjust the index to refer to the first character on the line.

```
lineend
```

Adjust the index to refer to the last character on the line.

```
wordstart
```

Adjust the index to refer to the first character of the word containing the current index. A word consists of any number of adjacent characters that are letters, digits, or underscores, or a single character that is not one of these.

`wordend`

Adjust the index to refer to the character just after the last one of the word containing the current index. If the current index refers to the last character of the text then it is not modified.

If more than one modifier is present then they are applied in left-to-right order. For example, the index "`end - 1 chars`" refers to the next-to-last character in the text and "`insert wordstart - 1 c`" refers to the character just before the first one in the word containing the insertion cursor.

MARKS

The second form of annotation in text widgets is a mark. Marks are used for remembering particular places in a text. They have names and they refer to places in the file, but a mark isn't associated with particular characters. Instead, a mark is associated with the gap between two characters. Only a single position may be associated with a mark at any given time. If the characters around a mark are deleted the mark will still remain; it will just have new neighbor characters. In contrast, if the characters containing a tag are deleted then the tag will no longer have an association with characters in the file. Marks may be manipulated with the `mark` sub-command, and their current locations may be determined by using the mark name as an index in widget commands.

One mark has special significance. The mark *insert* is associated with the insertion cursor. The mark *point* is an synonym for insert. This special mark may not be unset.

USAGE

Text items are supported by the Pad++ text command. Text items are created with widget commands of the following form:

```
pathName create text [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for text items:

[84] -font fontname

(available only for text item types)

Specifies the font to be used for rendering text for this item. fontname must specify a filename which contains an Adobe Type 1 font, or the string "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`".

[85] -pen color

(available ony for handle, line, polygon, portal, rectangle, spline, text and textfile item types)

Color specifies a color to use for drawing the text characters; it may have any of the forms accepted by Tk_GetColor. It may also be "none", in which case the text will be not be drawn. This option defaults to black.

[86] -text string

(available only for text and textfile item types)

String specifies the characters to be displayed in the text item. Newline characters cause line breaks, and tab characters are supported. This option defaults to an empty string.

The `text` command is described above with the other Pad++ commands under WIDGET COMMANDS.

Textfile Items

A textfile item displays a string of characters on the screen in one or more lines as with text items, but the text is loaded in from a file. Textfile items are supported by the Pad++ text command. Textfile items are created with widget commands of the following form:

```
pathName create textfile [option value option value ...]
```

There may be any number of option-value pairs, each of which sets one of the configuration options for the item. These same option-value pairs may be used in itemconfigure widget commands to change the item's configuration. The following options are supported for text items:

[87] -file fileName

(available only for textfile item types)

fileName specifies the filename to read a text file from.

[88] -font fontname

(available only for HTML, portal, text and textfile item types)

Specifies the font to be used for rendering text for this item. fontname must specify a filename which contains an Adobe Type 1 font, or the string "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`".

[89] -pen color

(available only for handle, line, polygon, portal, rectangle, spline, text and textfile item types)

Color specifies a color to use for drawing the text characters; it may have any of the forms accepted by Tk_GetColor. It may also be "none", in which case the text will be not be drawn. This option defaults to black.

[90] -text

```
(available only for textfile item types)
```

Returns the text in this textfile item. This is a read-only option and can not be set.

Default Bindings

In the current implementation, new Pad++ widgets are not given any default behavior: all behavior has to be described explicitly. However, the PadDraw sample application has many event bindings that may be useful.

Global TCL Variables

Pad++ defines several global Tcl variables that are available for use by Tcl applications. They are:

- **Pad_Error** True during Pad++ background errors.

- **Pad_Version** Current version of this Pad++ software
- **Pad_Write** Used in the **<Write>** event for an application to specify if the system should write out a specific object or not. (See the **write** command and the **<Write>** event in the **bind** command.)

KPL-Pad++ Interface

As described in the section above on KPL ITEMS, Kpl is a byte-compiled language that comes with Pad++ that is typically used for creating new objects. It is a general-purpose language, and has the ability to call certain Pad++ rendering routines. Some basic documentation is available with the Pad++ release in *doc/kpl.troff*.

There are two ways to interact with Kpl. The first is to make a Pad++ Kpl item with a Kpl renderscript (described above). In this case, every time the item is rendered, the Kpl script will be executed. The second method is to use the `kpl` command available directly from Tcl. The `kpl` command has the following format:

```
kpl subcommand [args ...]
```

Where *subcommand* must be one of the following:

`eval` string

Byte-compiles and evaluates *string* as a Kpl script.

`push` value

Pushes *value* onto the top of the Kpl stack.

`pop`

Pops the top element off of the Kpl stack and returns it.

`get` name

Returns the current value of the Kpl variable, *name*.

`set` name value

Sets the Kpl variable *name* to *value*.

There are several Kpl commands available for interacting with the Tcl environment, and for rendering directly onto the Pad++ surface (when within a render callback). They are organized into a few groups as follows:

These commands provide a mechanism for accessing Tcl variables from Kpl.

```
tclset name value
```

Sets the global Tcl variable name to *value*.

```
tclset2 array_name element value
```

Sets the global Tcl array *array_name*(*element*) to *value*.

```
tclget name
```

Returns the value of the global Tcl variable *name*.

```
tclget2 array_name element
```

Returns the value of the global Tcl array *array_name*(*element*).

```
tcleval tcl_string
```

Evaluates the Tcl string *tcl_string*.

These commands provide basic drawing capability.

```
drawborder llcorner urcorner width border relief
```

Draws a 3D border within the rectangle specified by *llcorner* and *urcorner* (where each of those are 2D vectors). *Width* specifies the zoomable width of the border. *Border* specifies the border color and must have been previously allocated with the Pad++ `allocborder` command. *Relief* specifies the style of border, and must be one of: "`raised`", "`flat`", "`sunken`", "`groove`", "`ridge`", "`barup`", or "`bardown`".

```
drawline vector
```

Draws a line specified by *vector*. As Kpl vectors may be up to 16-dimensional, this vector can specify up to 8 (x, y) points. This routine will draw a line connecting as many points as are specified within *vector*.

```
drawimage imagetoken x y
```

Draws the image specified by imagetoken at the point (x, y). (Also see `allocimage`, `freeimage`, and `info` commands as well as the description of `image` items). This command can only be called within a render callback.

```
drawpolygon vector
```

Draws a polygon specified by *vector*. As Kpl vectors may be up to 16-dimensional, this vector can specify up to 8 (x, y) points. This routine will draw a closed polygon connecting as many points as are specified within *vector*.

```
drawtext text position
```

Draws text. *Text* specifies the text to be drawn. *Position* specifies the where the text gets drawn. *Position* is a two-dimensional vector specifying the (x, y) position. (Also see the KPL `setcolor`, `setfont`, and `setfontheight` commands.)

```
getlevel
```

Returns the current refinement level.

`getsize`

Returns the current size of the object, where size is the larger of the width and height.

`renderitem tagOrId`

During a render callback triggered by the *-renderscript* option, this function actually renders the object. During a -*renderscript* callback, all the items specified by *tagOrId* are rendered (and the current item is not rendered unless it is in *tagOrId*). This function may only be called during a render callback.

`setabslinewidth width`

Sets the current drawing with to an absolute width. All lines will be drawn with this width. This is an absolute width, so this specifies the width independent of the current view. I.e., the line width will not change as the view changes.

`setcapstyle capstyle`

Sets the capstyle of lines for drawing. *Capstyle* may be any of: "butt", "projecting", or "round".

`setcolor color`

Sets the current drawing color to *color*. Note that *color* must have been previously allocated by the `alloccolor` Pad++ command.

`setfont font`

Specifies the font to be used for rendering text for this item. *Font* must specify a filename which contains an Adobe Type 1 font, or the string "`System`" which causes the Pad++ line-font to be used. Defaults to "`System`". (Also see the `setfontheight` command.)

`setfontheight height`

Sets the height of the font for future drawing with render callbacks. *Height* is specified in pixels. (Also see the `setfont` command).

`setjoinstyle joinstyle`

Sets the joinstyle of lines for drawing. *Joinstyle* may be any of: "bevel", "miter", or "round".

`setlinewidth width`

Sets the current drawing width to a zoomable width. All lines will be drawn with this width. This is a zoomable width, so this specifies the width as it will look when the view has a magnification of 1.0.

These commands provide drawing commands in a style much like postscript.

`closepath`

Specifies the end of a path.

`curveto vector`

Draws a bezier curve. Here, *vector* is a six-dimensional vector. The current point plus these three points specify four points which control the bezier curve.

`fill`

Fills the current path.

`lineto vector`

Specifies a straight line in the current path from the current point to (x, y) specified by *vector*. Makes (x, y) the current point.

`moveto vector`

Moves the current point within the current path to (x, y) specified by *vector*.

`newpath`

Specifies the beginning of a new path.

`stroke`

Draws the current path with an outline only - the path is not filled.

These commands provide control over refinement.

`interrupted`

Returns true (1) if there has been an event during this render to interrupt it. It is up to objects that take very long to render themselves to check this flag during the rendering. If it is true (i.e., the render has been interrupted), then the Kpl render routine should return immediately - without completing the render. Generally, renders at refinement level 0 should always be quite fast, but further refinement levels can take an arbitrarily long time to render as long as they are interruptible.

`refine`

Specifies that this item wants to be refined. Pad++ will schedule a refinement, and at some point in the near future, the item will be re-rendered at the next higher refinement level. An item can use the current level in conjunction with this command to render itself simply at first, and then fill in more and more detail when it is refined.

Here is an example that creates a Kpl item with a renderscript that exercises some of the commands described here.

```
                                # Tcl code to load Kpl code and to create
                                # Pad++ Kpl item.
kpl eval 'triangle.kpl source
set pen [.pad alloccolor brown]
.pad create kpl -bb {-10:-10 110:110} -renderscript {test_drawing}

                                /* Kpl code (in a separate file)
                                   to test the drawing commands */
{
                                        /* Draw a looping bezier curve */
        3 setlinewidth
        'pen1 tclget setcolor
        newpath
                0:0 moveto
                200:75:-100:75:100:0 curveto
        stroke

                                        /* Draw a filled square */
        'pen2 tclget setcolor
        newpath
                0:0 moveto
                50:0 lineto
                50:50 lineto
                0:50 lineto
        fill

                                        /* Draw a square outline */
        'pen3 tclget setcolor
        newpath
                0:0 moveto
                50:0 lineto
                50:50 lineto
                0:50 lineto
                0:0 lineto
        stroke

                                        /* Draw a square outline
                                        with an absolute width */
        1 setabslinewidth
        'pen4 tclget setcolor
        newpath
                0:0 moveto
                50:0 lineto
                50:50 lineto
                0:50 lineto
                0:0 lineto
        stroke

                                        /* Cause one level of refinement.
                                        Notice the bezier curve is rendered
                                        at low-resolution at first,
                                        and then improves with refinement. */
        getlevel => i
        i 1 < {
                refine
        }
} -> test_drawing
```

Executables	Padwish Synopsis	TCL Synopsis	Widget-Specific Options
Widget Commands	Overview of Item Types	All Item Types	Grid Items
Relative Placement	Restrictions on Master Windows	Differences Between Pad++ and TK Grid Commands	Examples
Group Items	Handle Items	HTML Items	HTML ANCHORS
Image Items	KPL Items	Line Items	Pad Items
Polygon Items	Portal Items	Rectangle Items	Spline Items
TCL Items	Text Items	INDICES	MARKS
Textfile Items	Default Bindings	Global TCL Variables	KPL-Pad++ Interface

Reference Guide - 2 OCT 1996

Pad++ Bederson - 1

Pad++: A Zoomable Graphical Interface System

Benjamin B. Bederson and James D. Hollan

Computer Science Department
University of New Mexico
Albuquerque, NM 87131
(505) 277-3112
{bederson,hollan}@cs.unm.edu

ABSTRACT
Large information spaces are often difficult to access efficiently and intuitively. We are exploring Pad++, a graphical interface system based on zooming, as an alternative to traditional window and icon-based approaches. Objects can be placed in the graphical workspace at any size, and zooming is the fundamental navigational technique. The goal is to provide simple methods for visually navigating complex information spaces that ease the burden of locating information while maintaining an intuitive sense of location and of relationship between information objects.

KEYWORDS: Navigation, interactive interfaces, multiscale interfaces, zooming, authoring, information navigation, hypertext, information visualization, multimedia, world wide web.

INTRODUCTION

Over the past several years, a variety of techniques have been introduced for viewing large information spaces, including: SDMS [4], fisheye views [5], information visualizer [3], graphical fisheye views of graphs [10], Pad [9], and Pad++ [1][2][7]. Space-scale diagrams have been used as an analytical tool for some of them [6].

This paper accompanies a CHI'95 demonstration of Pad++, an interface system based on zooming. Pad++ workspaces are large high resolution areas, allowing the viewing of complex collections of information at multiple scales. Zooming and panning are the primary methods of navigation in Pad++. Several efficiency mechanisms [2] are employed to maintain interactive frame rates for animation, even when scenes get complicated.

Pad++ is a general-purpose substrate for exploring user interfaces. It directly supports creation and manipulation of zoomable graphical objects and navigation within a zoomable workspace. Pad++ is built as a new widget for Tk using Tcl, an interpreted scripting language. Increasingly popular, Tcl and Tk [8] combine a scripting language and Motif-like library for creating graphical user interfaces and applications without the need to write C code. The Tcl interface to Pad++ is similar to the interface to the Tk Canvas widget, a surface for drawing structured graphics.

Objects in Pad++ can be implemented so that they change the way they look depending on, among other things, their size, complexity of the current view, characteristics of users' task, lenses positioned over them, or the type of interface physics [2] currently operational. Such changes in view we call *semantic zooming*. It provides, for example, a simple method for representing abstraction. When you zoom out, you see a simplified rendering of the object, and when you zoom in, you see more details. Perlin [9] described a prototype zooming calendar based on this notion.

NAVIGATION

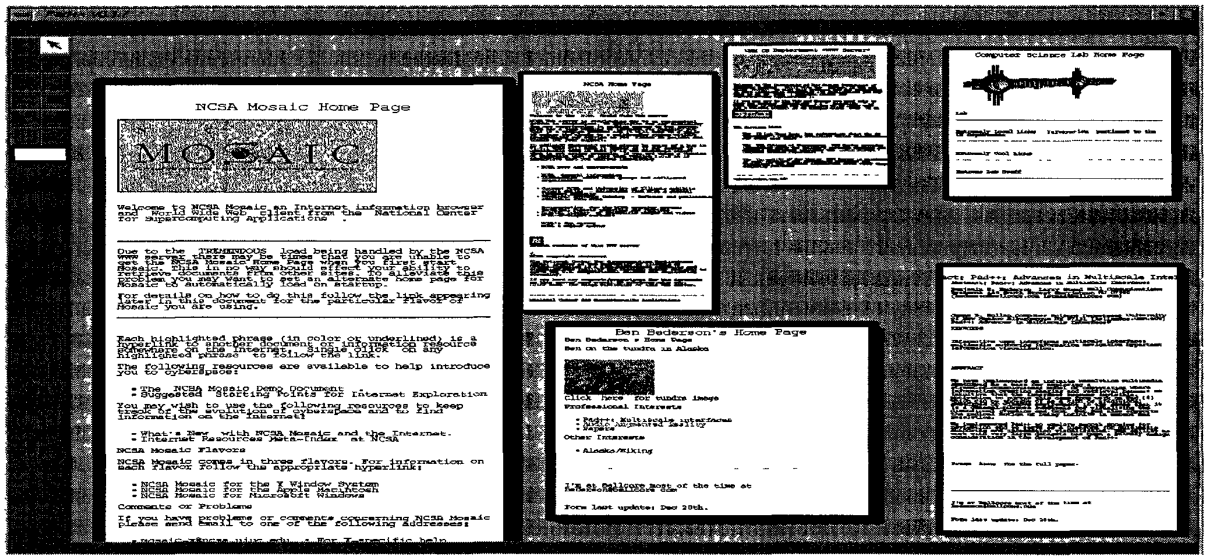
Finding information effectively with a Pad++ interface is important because intuitive navigation through large information spaces is a primary motivation. To accomplish this, Pad++ supports visual searching with zooming in addition to traditional mechanisms, such as content-based search.

Searching in Pad++ produces smooth animations to the desired objects. Animations interpolate in pan and zoom to bring the view to the specified location. If the end point, however, is more than one screen width away from the starting point, the animation zooms out to a point midway between the starting and ending points, far enough out so that both points are visible. The animation then smoothly zooms in to the destination. This gives a sense of context to the viewer and helps maintain object constancy. In addition it speeds up the animation since most of the panning is performed when zoomed out and thus covers more distance than panning while zoomed in. We use space-scale diagrams [6] to help analyze and construct these trajectories.

SAMPLE APPLICATION - HYPERTEXT

Hypertext systems confront the problem of how to give users an intuitive sense of location as they navigate through large information spaces. An example is NCSA's Mosaic system. It allows traversal of a vast information space across the internet via hyperlinks. In Mosaic, as with many other window-based hypertext systems, following a link replaces the contents of the window with the linked data, or sometimes brings up a new window. However, there is no graphical depiction of the relationship among windows - even when there is a strong semantic relationship. Thus, it is quite common to hear users complain of losing a sense of relationship between where they are and where they've been.

This work was supported in part by ARPA contract #N66001-94-C-6039 to the University of New Mexico. For more information about this project see http://www.cs.unm.edu/pad++.



NCSA Mosaic Home Page

Welcome to NCSA Mosaic, an Internet information browser and World Wide Web client from the National Center for Supercomputing Applications .

Due to the TREMENDOUS load being handled by the NCSA WWW server there may be times that you are unable to get the NCSA Mosaic Home Page when you first start Mosaic. This in no way should effect your ability to retrieve documents from other sites. To alleviate this problem you may want to set an alternate home page for Mosaic to automatically load on startup.

For details on how to do this follow the link appearing later in this document for the particular flavor of Mosaic you are using.

Each highlighted phrase (in color or underlined) is a hyperlink to another document or information resource somewhere on the Internet. Single click on any highlighted phrase to follow the link.

The following resources are available to help introduce you to cyberspace:

- The NCSA Mosaic Demo Document .
- Suggested Starting Points for Internet Exploration

You may wish to use the following resources to keep track of the evolution of cyberspace and to find information on the Internet:

- What's New with NCSA Mosaic and the Internet.
- Internet Resources Meta-Index at NCSA

NCSA Mosaic Flavors

NCSA Mosaic comes in three flavors. For information on each flavor follow the appropriate hyperlink:

- NCSA Mosaic for the X Window System
- NCSA Mosaic for the Apple Macintosh
- NCSA Mosaic for Microsoft Windows

Comments or Problems

If you have problems or comments concerning NCSA Mosaic please send email to one of the following addresses:

- mosaic-x@ncsa.uiuc.edu - For X-specific help

Ben Bederson's Home Page

Ben Bederson's Home Page

Ben on the tundra in Alaska

Click here for tundra image

Professional Interests

- Pad++: Multiscale interfaces
- Audio Augmented Reality
- Papers

Other Interests

- Alaska/Hiking

I'm at Bellcore most of the time at bederson@bellcore.com

Form last update: Dec 20th.

Computer Science Lab Home Page

Figure 1. A screen dump showing a series of World Wide Web home pages loaded into Pad++.

Pad++ attempts to address this problem by using a very high resolution zoomable surface to graphically layout the links representing traversals. When a hyperlink is selected, the linked data is loaded to the side and made smaller while the user's view is animated to center the new information. The nodes are layed out in such a way that no traversal of links can cause overlapping. Pad++ can read in hypertext files written in Hypertext Markup Language (HTML). Figure 1 shows a snapshot with a number of home pages loaded and several links followed.

CONCLUSION

Pad++ is a tool for exploring interfaces and visualizations based on zooming. We believe that multiscale interfaces provide effective alternative mechanisms for addressing problems associate with navigation of very large information collections. Our goal is to provide simple methods for visually navigating that ease the burden of locating information while maintaining an intuitive sense of location and of relationships between information objects.

We are currently, in collaboration with NYU and Bellcore, continuing development of the Pad++ substrate as well as starting work in several application domains.

ACKNOWLEDGMENTS

We acknowledge Ken Perlin and his students and staff, David Fox, Matthew Fuchs, Jon Meyer and David Bacon, at NYU for many enjoyable discussions and for initiating our interest in zooming interfaces. We thank Mike Lesk, George Furnas, and Al Aho for releasing an early version of Pad++ we developed at Bellcore. We especially appreciate Craig Wier's support as part of ARPA's new HCI Initiative.

REFERENCES

[1] Bederson, B. B., Stead, L., and Hollan, J. D. Pad++: Advances in Multiscale Interfaces, *Proceedings of ACM Human Factors in Computing Systems Conference Companion (CHI'94)*, 315-316.

[2] Bederson, B. B. and Hollan, J. D. Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics, *Proceedings of ACM User Interface Software and Technology Conference (UIST'94)*, 17-26.

[3] Card, S. K., Robertson, G. G., and Mackinlay, J. D. The Information Visualizer, an Information Workspace, *Proceedings of ACM Human Factors in Computing Systems Conference (CHI '91)*, 181-188.

[4] Donelson, W. C. Spatial Management of Information, *Proceedings of 1978 ACM SIGGRAPH Conference*, 203-209.

[5] Furnas, G. W. Generalized Fisheye Views, *Proceedings of 1986 ACM SIGCHI Conference*, 16-23.

[6] Furnas, G. W. and Bederson, B. B. Space-Scale Diagrams: Understanding Multiscale Interfaces, *Proceedings of ACM Human Factors in Computing Systems Conference (CHI'95)*, In Press.

[7] Meyer, J., Perlin, K., Bederson, B. B., and Hollan, J. D. Two Document Visualization Techniques for Zoomable Interfaces, *Proceedings of ACM Human Factors in Computing Systems Conference Companion (CHI'95)*, submitted.

[8] Ousterhout, J. K. Tcl and the Tk Toolkit, Addison Wesley, 1994.

[9] Perlin, K. and Fox, D. Pad: An Alternative Approach to the Computer Interface, *Proceedings of 1993 ACM SIGGRAPH Conference*, 57-64.

[10] Sarkar, M., Snibbe, S. S., Tversky, O. J., and Reiss, S. P. Stretching the Rubber Sheet: A Technique for Viewing Large Layouts on Small Screens, *Proceedings of ACM User Interface Software and Technology Conference (UIST'93)*, 81-91.

Pad++ Bederson - 2

Space-Scale Diagrams: Understanding Multiscale Interfaces

George W. Furnas and Benjamin B. Bederson (t)

Bellcore
445 South Street
Morristown, NJ 07960-6438
(201) 829-4289
gwf@bellcore.com
bederson@bellcore.com

(t)
Current address: bederson@cs.unm.edu, Computer Science Department, University of New Mexico, Albuquerque, NM 87131

Abstract

Big information worlds cause big problems for interfaces. There is too much to see. They are hard to navigate. An armada of techniques has been proposed to present the many scales of information needed. Space-scale diagrams provide an analytic framework for much of this work. By representing both a spatial world and its different magnifications explicitly, the diagrams allow the direct visualization and analysis of important scale related issues for interfaces.

Keywords:

Zoom views, multiscale interfaces, fisheye views, information visualization, GIS; visualization, user interface components; formal methods, design rationale.

Introduction

For more than a decade there have been efforts to devise satisfactory techniques for viewing very large information worlds. (See, for example, [6] and [9] for recent reviews and analyses). The list of techniques for viewing 2D layouts alone is quite long: the Spatial Data Management System [3], Bifocal Display[1], Fisheye Views [4][12], Perspective Wall [8], the Document Lens [11], Pad [10], and Pad++ [2], the MacroScope [7], and many others.

Central to most of these 2D techniques is a notion of what might be called multiscale viewing. An interface is devised that allows information objects and the structure embedding them to be displayed at many different magnifications, or scales. Users can manipulate which objects, or which part of the whole structure, will be shown at what scale. The scale may be constant and manipulated over time as with a zoom metaphor, or varying over a single view as in the distortion techniques (e.g., fisheye or bifocal metaphor). In either case, the basic assumption is that by moving through space and changing scale the users can get an integrated notion of a very large structure and its contents, navigating through it in ways effective for their tasks.

This paper introduces space-scale diagrams as a technique for understanding such multiscale interfaces. These diagrams make scale an explicit dimension of the representation, so that its place in the interface and interactions can be visualized, and better analyzed. We are finding the diagrams useful for understanding such interfaces geometrically, for guiding the design of code, and as interfaces to authoring systems for multiscale information.

This paper will first present the necessary material for understanding the basic diagram and its properties. Subsequent sections will then use that material to show several examples of their uses.

THE SPACE-SCALE DIAGRAM

The basic diagram concepts

The basic idea of a space-scale diagram is quite simple. Consider, for example, a square 2D picture (Figure 1a).

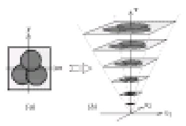


Figure 1. The basic construction of a Space-Scale diagram from a 2D picture.

The space-scale diagram for this picture would be obtained by creating many copies of the original 2D picture, one at each possible magnification, and stacking them up to form an inverted pyramid (Figure 1b). While the horizontal axes represent the original spatial dimensions, the vertical axis represents scale, i.e., the magnification of the picture at that level. In theory, this representation is continuous and infinite: all magnifications appear from 0 to infinity, and the "picture" may be a whole 2D plane if needed.

Before we can discuss the various uses of these diagrams, three basic properties must be described. Note first that a user's viewing window can be represented as a fixed-size horizontal rectangle which, when moved through the 3D space-scale diagram, yields exactly all the possible pan and zoom views of the original 2D surface (Figure 2).

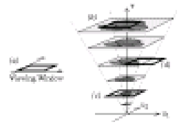


Figure 2. The viewing window is shifted rigidly around the 3D diagram to obtain all possible pan/ zoom views of the original 2D surface, e.g., (b) a zoomed in view of the circle overlap, (c) a zoomed out view including the entire original picture, and (d) a shifted view of a part of the picture.

This property is useful for studying pan and zoom interactions in continuously zoomable interfaces like Pad and Pad++ [2][10].

Secondly, note that a point in the original picture becomes a ray in this space-scale diagram. The ray starts at the origin and goes through the corresponding point in the continuous set of all possible magnifications of the picture (Figure 3).

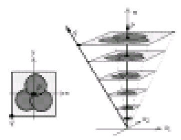


Figure 3. Points like p and q in the original 2D surface become corresponding "great rays" p and q in the space-scale diagram. (The circles in the picture therefore become cones in the diagram, etc.)

We call these the *great rays* of the diagram. As a result, regions of the 2D picture become generalized cones in the diagram. For example, circles become circular cones and squares become square cones.

A third property follows from the fact that typically the properties of the original 2D picture (e.g., its geometry) are considered invariant under moving the origin of the 2D coordinate system. In the space-scale diagrams, such a change of origin corresponds to a "shear" (Figure 4), i.e., sliding all the horizontal layers linearly so as to make a different great ray become vertical.

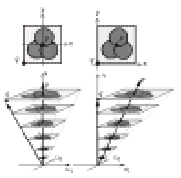


Figure 4. Shear invariance. Shifting the origin in the 2D picture from *p* to *q* corresponds to shearing the layers of the diagram so the *q* line becomes vertical. Each layer is unchanged, and great rays remain straight. Only those conclusions which remain true under all such shears are valid.

Thus, if one only wants to consider properties of the original diagram that are invariant under change of origin, the only meaningful properties of the space-scale diagram are those invariant under such a shear. For example, the absolute angles between great rays change with shear, and so should be given no special meaning.

Now that the basic concepts and properties of space-scale diagrams have been introduced by the detailed Figures 1-4, we can make a simplification. Those figures have been three dimensional, comprising two dimensions of space and one of scale ("2+1D"). Substantial understanding may be gained, however, from the much simpler two-dimensional versions, comprising one dimension of space and one dimension of scale ("1+1D"). It could, for example be a vertical slice from, or an edge on view of, the 2+1D version, or just a space-scale view of a truly 1D world (e.g., a time line). In the 1+1D diagram, since the spatial world is 1D, a viewing window is a line segment that can be moved around the diagram to represent different pan and zoom positions. It is convenient to show the window as a narrow slit, so that looking through it shows the corresponding 1D view. Figure 5 shows one such diagram illustrating a sequence of three zoomed views.

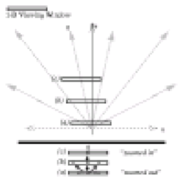


Figure 5. A "1+1D" space-scale diagram has one spatial dimension, u, and one scale dimension, v. The six great rays here correspond to six points in a 1D spatial world, put together at all magnifications. The viewing window, like the space itself, is one dimensional, and is shown as a narrow slit with the

corresponding 1-D window view being visible through the slit. Thus the sequence of views (a), (b), (c) begins with a view of all six points, and then zooms in on the point q. The views, (a), (b), (c) are redrawn at bottom to show the image at those points.

The basic math.

It is helpful to characterize these diagrams mathematically. This will allow us to use analytic geometry along with the diagrams to analyze multiscale interfaces, and also will allow us to map conclusions back into the computer programs that implement them.

The mathematical characterization is simple. Let the pair *(x,z)* denote the point *x* in the original picture considered magnified by the multiplicative scale factor *z*. We define any such *(x,z)* to correspond to the point *(u,v)* in the space-scale diagram where *u=xz* and *v=z*. This second trivial equation is needed to make the space-scale coordinates distinct, and because there are other versions of space-scale diagrams, e.g., where *v=log(z)*. Conversely, of course, a point *(u,v)* in the space-scale diagram corresponds to *(x,z)*, i.e., a point *x* in the original diagram magnified by a factor *z*, where *x=u/v*, and *z=v*. The notation is a bit informal, in that *x* and *u* are single coordinates in the 1+1D version of the diagrams, but a sequence of two coordinates in the 2+1D version.

A few words are in order about the XZ vs. UV characterizations. The *(x,z)* notation can be considered a world-based coordinate system. It is important in interface implementation because typically a world being rendered in a multiscale viewer is stored internally in some fixed canonical coordinate system (denoted with *x*'s). The magnification parameter, *z*, is used in the rendering process. Technically one could define a type of space-scale diagram that plots the set of all *(x,z)* pairs directly. This "XZ" diagram would stack up many copies of the original diagram, all of the same size, i.e., without rescaling them. In this representation, while the picture is always constant size, the viewing window must grow and shrink as it moves up and down in *z*, indicating its changing scope as it zooms. Thus while the world representation is simple, the viewer behavior is complex. In contrast, the "UV" representation of the space-scale diagrams focused on in this paper can be considered view-based. Conceptually, the world is statically prescaled, and the window is rigidly moved about. The UV representation is thus very useful in discussing how the views should behave. The coordinate transform formulas allow problems stated and solved in terms of view behavior, i.e., in the UV domain, to have their solutions transformed back into XZ for implementation.

EXAMPLE USES OF SPACE-SCALE DIAGRAMS

With these preliminaries, we are prepared to consider various uses of space-scale diagrams. We begin with a few examples involving navigation in zoomable interfaces, then consider how the diagrams can help visualize multiscale objects, and finish by showing how other, non-zoom multiscale views can be characterized.

Pan-zoom trajectories

One of the dominant interface modes for looking at a large 2D world is to provide an undistorted window onto the world and allow the user to pan and zoom. This method is used in [2][3][7][10], as well as essentially all map viewers in GISs (geographic information systems). Space-scale diagrams are a very useful way for researchers studying interfaces to visualize such interactions, since moving a viewing window around via pans and zooms corresponds to taking it on a trajectory through scale-space. If we represent the window by its midpoint, the trajectories become curves and are easily visualized in the space-scale diagram. In this section, we first show how easily space-scale diagrams represent pan/zoom sequences. Then we show how they can be used to solve a very concrete interface problem. Finally we analyze a more sophisticated pan/zoom problem, with a rather surprising information theoretic twist.

Basic trajectories.

Figure 6 shows how the basic pan-zoom trajectories can be visualized.

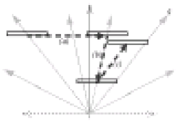


Figure 6. Basic Pan-Zoom trajectories are shown in the heavy dashed lines:. (a) Is a pure Pan,. (b) is a pure Zoom (out), (c) is a "Zoom-around" the point q.

In a simple pan (a), the window' s center traces out a horizontal line as it slides through space at a fixed scale. A pure zoom around the center of the window follows a great ray (b), as the window's viewing scale changes but its position is constant. In a "zoom-around" the zoom is centered around some fixed point other than the center of the window, e.g., q at the right hand edge of the window. Then the trajectory is a straight line parallel to the great ray of that fixed point. This moves the window so that the fixed point stays put in the view. In the figure, for example, the point, q, always intersects the windows on trajectory (c) at the far right edge, meaning that the point, q, is always at that position in the view. If as in this case the fixed point is itself within the window, we call it a zoom-around-within-window or *zaww*. Other sorts of pan-zoom trajectories have their characteristic shapes as well and are hence easily visualized with space-scale diagrams.

The joint pan-zoom problem.

There are times when the system must automatically pan and zoom from one place to another, e.g., moving the view to show the result of a search. Making a reasonable joint pan and zoom is not entirely trivial. The problem arises because in typical implementations, pan is linear at any given scale, but zoom is logarithmic, changing magnification by a constant factor in a constant time. These two effects interact. For example, suppose the system needs to move the view from some first point *(x1 , z1)* to a second point *(x2 , z2)*. For example, a GIS might want to shift a view of a map from showing the state of Kansas, to showing a zoomed in view of the city of Chicago, some thousand miles away. A naive implementation might compute the linear pans and log-linear zooms separately and execute them in parallel. The problem is that when zooming in, the world view expands exponentially fast, and the target point *x2* runs away faster than the pan can keep up with it. The net result is that the target is approached non-monotonically: it first moves away as the zoom dominates and only later comes back to the center of the view. Various seat-of-the pants guesses (taking logs of things here and there) do not work either.

What is needed is a way to express the desired monotonicity of the view's movement in both space and scale. This viewbased constraint is quite naturally expressed in the UV spacescale diagram as a bounding parallelogram (Figure 7).

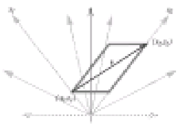


Figure 7. Solution to the simple joint pan-zoom problem. The trajectory s monotonically approaches point 2 in both pan and zoom.

Three sides of the parallelogram are simple to understand. Since moving up in the diagram corresponds to increasing magnification, any trajectory which exits the top of the parallelogram would have overshot the zoom-in. A trajectory exiting the bottom would have zoomed out when it should have been zooming in. One exiting the right side would have overshot the target in space. The fourth side, on the left, is the most interesting. Any point to the left of that line corresponds to a view in which the target *x2* is further away from the center of the window than where it started, i.e., violating the nonmonotonic approach. Thus any admissible trajectory must stay within this parallelogram, and in general must never move back closer to this left side once it has moved right. The simplest such trajectory in UV space is the diagonal of the parallelogram. Calculating it is simple analytic geometry. The coordinates of points 1 and 2 would typically come from the implementation in terms of XZ. These would first be transformed to UV. The linear interpolation is done trivially there, and the resulting equation transformed back to XZ for use in the implementation. If one composes all these algebraic steps into one formula, the trajectory in XZ for this 1-D case is:

Thus to get a monotonic approach, the scale factor, *z*, must change hyperbolically with the panning of *x*. This mathematical relationship is not easily guessed but falls directly out of the analysis of the space-scale diagram. We implemented the 2D analog in Pad++ and found the net effect is visually much more pleasing than our naive attempts, and count this as a success of space-scale diagrams.

Optimal pan-zooms and shortest paths in scale-space

Since panning and zooming are the dominant navigational motion of these undistorted multiscale interfaces, finding "good" versions of such motions is important.The previous example concerned finding a trajectory where "good" was defined by monotonicity properties. Here we explore another notion of a "good" trajectory, where "good" means "short".

Paradoxically, in scale-space the shortest path between two points is usually not a straight line. This is in fact one of the great advantages of zoomable interfaces for navigation and results from the fact that zoom provides a kind of exponential accelerator for moving around a very large space. A vast distance may be traversed by first zooming out to a scale where the old position and new target destination are close together, then making a small pan from one to the other, and finally zooming back in (see Figure 8).

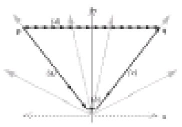


Figure 8. The shortest path between two points is often not a straight line. Here each arrow represents one unit of cost. Because zoom is logarithmic, it is often "shorter" to zoom out (a), make a small pan (b), and zoom back in (c), than to make a large pan directly (d).

Since zoom is naturally logarithmic, the vast separation can be shrunk much faster than it can be directly traversed, with exponential savings in the limit. Such insights raise the question of what is really the optimal shortest path in scale-space between two points.

When we began pondering this question, we noted a few important but seemingly unrelated pieces of the puzzle. First, one naive intuition about how to pan and zoom to cross large distances says to zoom out until both the old and new location are in the view, then zoom back into the new one. Is this related at all to any notion of a shortest path? Second, window size matters in this intuitive strategy: if the window is bigger, then you do not have to zoom out as far to include both the old and new points. A third piece of the puzzle arises when we note that the "cost" of various pan and zoom operations must be specified formally before we can try to solve the shortest path question. While it seems intuitive that the cost of a pure pan should be linear in the distance panned, and the cost of a pure zoom should be logarithmic with change of scale, there would seem to be a puzzling free parameter relating these two, i.e., telling how much pan is worth how much zoom.

Surprisingly, there turns out to be a very natural information metric on pan/zoom costs which fits these pieces together. It not only yields the linear pan and log zoom costs, but also defines the constant relating them and is sensitive to window size. The metric is motivated by a notion of visual informational complexity: the number of bits it would take to efficiently transmit a movie of a window following the trajectory.

Consider a digital movie made of a pan/zoom sequence over some 2D world. Successive frames differ from one another only slightly, so that a much more efficient encoding is possible. For example, if successive frames are related by a small pan operation, it is necessary only to transmit the bits corresponding to the new pixels appearing at the leading edge of the panning window. The bits at the trailing edge are thrown away. The 1D version is shown in Figure 9a.

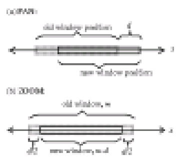


Figure 9. Information metric on pan and zoom operations on a 1D world. (a) Shifting a window by *d* requires *db* new bits. (b) Zooming in by a factor of *(w-d)/w*, throws away *db* bits, which must be replaced with just that amount of diffuse, higher resolution information when the window is magnified and brought back to full resolution.

If the bit density is *b* (i.e., bits per centimeter of window real estate), then the number of bits to transmit a pan of size *d* is *db*.

Similarly, consider when successive frames are related by a small pure zoom-in operation (Figure 9b), say where a window is going to magnify a portion covering only *(w-d)/w* of what it used to cover (where *w* is the window size). Then too, *db* bits are involved. These are the bits thrown away at the edges of the window as the zoom-in narrows its scope. Since this new smaller area is to be shown magnified, i.e., with higher resolution, it is exactly this number of bits, *db*, of high resolution information that must be transmitted to augment the lower resolution information that was already available.

The actual calculation of information cost for zooms requires a little more effort, since the amount of information required to make a total zoom by a factor *r* depends on the number and size of the intermediate steps. For example, two discrete step zooms by a factor of 2 magnification require more bits than a single step zoom by a factor of 4. (Intuitively, this is because showing the intermediate step requires temporarily having some new high resolution information at the edges of the window that is then thrown away in the final scope of the zoomed-in window.) Thus the natural case to consider is the continuous limit, where the step-size goes to zero. The resulting formula says that transmitting a zoom-in (or out) operation for a total magnification change of a factor *r* requires *bw*log*(r)* bits.

Thus the information metric, based on a notion of bits required to encode a movie efficiently, yields exactly what was promised: linear cost of pans (*db*), log costs of zooms (*bw*log*(r)*), and a constant (*w*) relating them that is exactly the window size. Similar analyses give the costs for other elementary motions. For example, a zoom around any other point within the window (a *zaww*) always turns out to have the same cost as a pure (centered) zoom. Other arbitrary zoom-arounds are somewhat more complicated.

From these components it is possible to compute the costs of arbitrary trajectories, and therefore in principle to find minimal ones. Unfortunately, the truly optimal ones will have a complicated curved shape, and finding it is a complicated calculus- of-variations problem. We have limited our work so far to finding the shortest paths within certain parameterized families of trajectories, all of which are piecewise pure pans, pure zooms or pure *zaww*'s. We sketch typical members of the families on a space-scale diagram, pick parameterizations of them and apply simple calculus to get the minimal cases. There is not room here to go through these in detail, but we give an overview of the results.

Before doing so, however, it should be mentioned that, despite all this formal work, the real interface issue of what constitutes a "good" pan/zoom trajectory is an empirical/cognitive one. The goal here is to develop a candidate theory for suggesting trajectories, and possibly for modelling and understanding future empirical work. The suitability of the information-based approach followed here hinges on an implicit cognitive theory that humans watching a pan/zoom sequence have somehow to take in, i.e., encode or understand, the sequence of views that is going by. They need to do this to interpret the meaning of specific things they are seeing, understand where they are moving to, how to get back, etc. It is assumed that, other things being equal, "short" movies are somehow easier, taking fewer cognitive resources (processing, memory, etc.) than longer ones. It is also assumed that human viewers do not encode successive frames of the movie but that a small pan or small zoom can be encoded as such, with only the deltas, i.e., the new information, encoded. Thus to some approximation, movies with shorter encoded lengths will be better. (We are also at this point ignoring the content of the world, assuming that no special content-based encoding is practical or at least that information density at all places and scales is sufficiently uniform that its encoding would not change the relative costs.)

To get some empirical idea of whether this information-theoretic approach to "goodness" of pan-zoom trajectories matches human judgment, we implemented some simple testing facilities. The testing interface allows us to animate between two specified points (and zooms) with various trajectories, trajectories that were analyzed and programmed using space-scale diagrams. We did informal testing among a few people in our lab to see if there was an obvious preference between trajectories and compared these to the theory.

For large separations, pure pan is very bad. There is strong agreement between theory and subjects' experience. Theory says the information description of a pure pan movie should be exponentially longer than one using a substantial amount of zoom. Empirically, users universally disliked these big pans. They found it difficult to maintain context as the animation flew across a large scene. Further, when the distance to be travelled was quite large and the animation was fast, it was hard to see what was happening; if the animation was too slow, it took too long to get there.

At the other extreme, for small separations viewers preferred a short pure pan to strategies that zoomed out and in. It turns out that this is also predicted by the theory for the family piecewise pan/zoom/*zaww* trajectories we considered here. Depending on exactly which types of motions are allowed, the theory predicts that to traverse separations of less than 1 to 3 window widths, the corresponding movie is informationally shorter if it is just a pan.

Does the naively proposed navigation strategy ("zoom out until the starting and ending points are close, then pan in") ever arise in this analysis? At this high level of description, the answer is definitely "yes." The fine points, however, are more subtle. If only *zaww*'s are allowed, the shortest path indeed involves zooming out until both are visible, then zooming in (Figure 10).

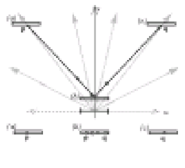


Figure 10. The shortest zaww path between p (a) and q zooms out till both are within the window (b), then zooms in (c). The corresponding views are shown below the diagram.

For users this was quite a well-liked trajectory. If pans are allowed, however, the information metric disagrees slightly with the naive intuition. It says instead to stop the zoom just before both are in view, then make a pan of 1-3 screen separations (just as described for short pans), then finally zoom in. The information difference between this optimal strategy and the naive one is small, and our users similarly found small differences in the acceptability. It will be interesting to examine these variants more systematically.

Our overall conclusion is that the information metric, based on analyses of space-scale diagrams, is quite a reasonable way to determine "good" pan/zoom trajectories.

Showing semantic zooming

Another whole class of uses for space-scale diagrams is for the representation of semantic zooming[10]. In contrast to geometric zooming, where objects change only their size and not their shape when magnified, semantic zooming allows objects to change their appearance as the amount of real estate available to them changes. For example, an object could just appear as a point when small. As it grows, it could then in turn appear as a solid rectangle, then a labeled rectangle, then a page of text, etc.

Figure 11 shows how geometric zooming and semantic zooming appear in a space-scale diagram.

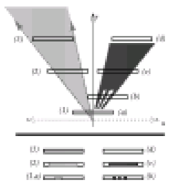


Figure 11. Semantic Zooming. Bottom slices show views at different points.

The object on the left, shown as an infinitely extending triangle, corresponds to a 1D gray line segment, which just appears larger as one zooms in (upward: 1,2,3). On the right is an object that changes its appearance as one zooms in. If one zooms out too far (a), it is not visible. At some transition point in scale, it suddenly appears as a three segment dashed line (b), then as a solid line (c), and then when it would be bigger than the window (d), it disappears again.

The importance of such a diagram is that it allow one to see several critical aspects of semantic objects that are not otherwise easily seen. The transition points, i.e., when the object changes representation as a function of scale, is readily apparent. Also the nature of the changing representations, what it looks like before and after the change, can be made clear. The diagram also allows one to compare the transition points and representations of the different objects inhabiting a multiscale world.

We are exploring direct manipulation in space-scale diagrams as an interface for multi-scale authoring of semantically zoomable objects. For example, by grabbing and manipulating transition boundaries, one can change when an object will zoom semantically. Similarly, suites of objects can have their transitions coordinated by operations analogous to the snap, align, and distribute operators familiar to drawing programs, but applied in the space-scale representation.

As another example of semantic zooming, we have also used space-scale diagrams to implement a "fractal grid." Since grids are useful for aiding authoring and navigation, we wanted to design one that worked at all scales -- a kind of virtual graph paper over the world, where an ever finer mesh of squares appears as you zoom in. We devised the implementation by first designing the 1D version using the space-scale diagram of Figure 12.

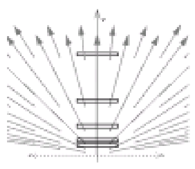


Figure 12. Fractal grid in 1D. As the window moves up by a factor of 2 magnification, new gridpoints appear to subdivide the world appropriately at that scale. The view of the grid is the same in all five windows.

This is the analog of a ruler where ever finer subdivisions appear, but by design here they appear only when you zoom in (move upward in the figure). There are nicely spaced gridpoints in the window at all five zooms of the figure. Without this fractal property, at some magnification the grid points would disappear from most views.

Warps and fisheye views

Space-scale diagrams can also be used to produce many kinds of image warpings. We have characterized the spacescale diagram as a stack of image snapshots at different zooms. So far in this paper, we have always taken each image as a horizontal slice through scale space. Now, instead imagine taking a cut of arbitrary shape through scale space and projecting down to the u axis. Figure 13 shows a step-up-step-down cut that produces a mapping with two levels of magnification and a sharp transition between them.

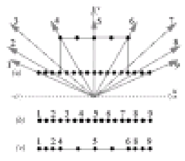


Figure 13. Warp with two levels of magnification and an abrupt transition between them. (a) shows the trajectory through scale-space, (b) shows the unwarped view, and (c) shows the warped view (notice rays 3 and 7 don't appear).

Here, (a) shows the trajectory through scale space, (b) shows the result that would obtain if the cut was purely flat at the initial level, and (c) shows the warped result following.

Different curves can produce many different kinds of mappings. For example, Figure 14 shows how we can create a fisheye view.

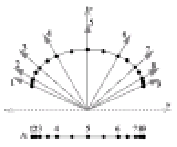


Figure 14. Fisheye view.

By taking a curved trajectory through scalespace, we get a smooth distortion that is magnified in the center and compressed in the periphery. Other cuts can create bifocal [1] and perspective wall [8].

For cuts as in Figure 13, which are piece-wise horizontal, the magnification of the mapping comes directly from the height of the slice. When the cuts are curved and slanted, the geometry is more complicated, but the magnification can always be determined by looking at the projection as in Figure 14.

CONCLUSION

This paper introduces space-scale diagrams as a new technique for understanding multiscale interfaces. Their defining characteristic and principal virtue is that they represent scale explicitly. We showed how they can aid the analysis of pans and zooms because they take a temporal structure and turn it into a static one: a sequence of views becomes a curve in scale-space. This has already helped in the design of good pan/zoom trajectories for Pad++. We showed how the diagrams can help visualization of semantic zooming by showing an object in all its scale-dependent versions simultaneously. We expect to use this as an interface for designing semantically zoomable objects. We also suggested that diagrams may be useful for examining other non-flat multiscale representation, such as fisheye views.

Space-scale diagrams, therefore, are important for visualizing various problems of scale, for aiding formal analyses of those problems, and finally, for implementing various solutions to them.

Acknowledgments

This work was supported in part by ARPA grant N66001-94-C-6039. The authors would like to thank Maria Slowiaczek for her very helpful comments on drafts of this paper.

References

1. Apperley, M.D., Tzavaras, I. and Spence, R, A bifocal display technique for data presentation, Proceedings of Eurographics `82, pp. 27-43.

2. Bederson, B. B. and Hollan, J.D., Pad++: A zooming graph- ical interface for exploring alternate interface physics. In Proceedings of ACM UIST'94, (1994, Marina Del Ray, CA), ACM Press, pp 17-26.

3. Donelson, W., Spatial management of information. In Pro- ceedings of ACM SigGraph'78 (Atlanta, GA), ACM press, pp. 203-209.

4. Furnas, G.W., Generalized fisheye views. In Proceedings of CHI'86 Human Factors in Computing Systems (Boston, MA, April 1986), ACM press, pp. 16-23.

5. Furnas, G. W., The FISHEYE view: A new look at struc- tured files. Bell Laboratories Technical Memorandum, #82- 11221-22, Oct 18, 1982. 22pps.

6. Leung, Y.K. and Apperley, M.D., A unified theory of distor- tion-oriented presentation techniques. In press, TOCHI.

7. Lieberman, H., Powers of ten thousand: navigating in large information spaces. Short paper in Proceedings of ACM UIST'94, (1994, Marina Del Ray, CA), ACM Press, pp. 15- 16.

8. Mackinlay, J.D., Robertson, G.G. and Card, S.K., The per- spective wall: detail and context smoothly integrated. In Proceedings of CHI'91 Human Factors in Computing Sys- tems, ACM press, pp. 173-179.

9. Noik, E.G., A space of presentation emphasis techniques for visualizing graphs. In Proceedings of GI `94: Graphics Interface 1994, (Banff, Alberta, Canada, May 16-20, 1994), pp. 225-234.

10. Perlin, K. and Fox, D., Pad: An Alternative Approach to the Computer Interface. In Proceedings of ACM SigGraph `93 (Anaheim, CA) pp. 57-64.

11. Robertson, George. G. and Mackinlay, Jock, The Document Lens. In Proceedings of ACM UIST'93 (Atlanta, GA), ACM press, pp. 101-108.

12. Sarkar, M. and Brown, M.H., Graphical fisheye views of graphs. In Proceedings of ACM CHI'92 (Monterey, CA, May, 1992), ACM Press, pp. 83-91.

13. Sarkar, M., Snibbe, S.S., Tversky, O.J., and Reiss, S.P., Stretching the rubber sheet: a metaphor for visualizing large structures on small screens. In Proceedings of ACM UIST `93 (November 1993), ACM press, pp. 81-91.

Pad++ Bederson - 3

A Zooming Web Browser

Benjamin B. Bederson, James D. Hollan,
Jason Stewart, David Rogers, Allison Druin, David Vick
Computer Science Department
University of New Mexico
Albuquerque, NM 87131
{bederson, hollan, jasons, drogers, allisond, dvick}@cs.unm.edu
http://www.cs.unm.edu/pad++

ABSTRACT

The World Wide Web (WWW) is becoming increasingly important for business, education, and entertainment. Popular web browsers make access to Internet information resources relatively easy for novice users. Simply by clicking on a link, a new page of information replaces the current one on the screen. Unfortunately however, after following a number of links, people can have difficulty remembering where they've been and navigating links they have followed. As one's collection of web pages grows and as more information of interest populates the web, effective navigation becomes an issue of fundamental importance.

We are developing a prototype zooming browser to explore alternative mechanisms for navigating the WWW. Instead of having a single page visible at a time, multiple pages and the links between them are depicted on a large zoomable information surface. Pages are scaled so that the page in focus is clearly readable with connected pages shown at smaller scales to provide context. As a link is followed the new page becomes the focus and existing pages are dynamically repositioned and scaled. Layout changes are animated so that the focus page moves smoothly to the center of the display surface while contextual information provided by linked pages scales down.

While our browser supports multiscale representations of existing HTML pages, we have also extended HTML to support multiscale layout *within* a page. This extension, *Multi-Scale Markup Language* (MSML), is at an early stage of development. It currently supports inclusion within a page of variable-sized dynamic objects, graphics, and other interface mechanisms from our underlying Pad++ substrate. This provides sophisticated client-side interactions, permits annotations to be added to pages, and allows page constituents to be used as independent graphical objects.

In this paper, we describe our prototype web browser and authoring facilities. We show how simple extensions to HTML can support sophisticated client-side interactions. Finally, we discuss the results of preliminary user-interface testing and evaluation.

Keywords: world-wide web, browser, information navigation, zooming, information visualization, multiscale information, animated user interface, Pad++.

1. INTRODUCTION

In 1945 Vannevar Bush [8] envisioned "a future device for individual use, which is a sort of mechanized private file and library." He termed this device a *memex* and proposed a form of associative indexing in which arbitrary pieces of information could be linked together such that "when one of these items is in view, the other can be instantly recalled by tapping a button." He further conjectured that "wholly new forms of encyclopedias will appear, ready made with a mesh of associative trails running through them, ready to be dropped into the memex and there amplified." Today, fifty years later, we have the World Wide Web and a memex in the form of web browsers. See [4] for an overview of the WWW.

The increasing number of users and the ever-growing quantity of information available on the web present challenging interface and navigation problems. There are a variety of human factors [19] issues that need to be addressed. A larger number of users means that people with diverse talents, interests, and experiences will be on-line via the web. Many will be novices with little prior experience with computers. A simple click of the mouse can bring a user from their friend's home page to unknown destinations across the world. Traditionally, following a cross reference meant shuffling across the library to find another volume. While time-consuming, this reinforced the transition that was taking place. The difficulties that novice users confront can be

0-8194-2041-7/96/$6.00

instructional to developers. While experts may not have as much difficulty, they experience the same cognitive burdens, and may just have a higher threshold before they experience similar difficulties.

While the immediacy of traversing information links offers many advantages, it can also make it difficult to maintain an intuitive sense of where one is, and how one got there - leading to the frequently described sense of being *lost*. This is a classic problem of hypertext systems. Part of the problem can be attributed to windows-based interfaces. Current window systems don't readily support showing more than a few pages at a time. In addition, each page is usually in a separate window with no depiction of relationships to other windows. Popular WWW browsers, like other applications built according to current tiled or overlapping windows philosophies, also have this same problem, although they do offer limited methods to aid navigation by keeping track of interesting sites - usually in hierarchical sets of *hotlists* or *bookmarks*.

Several groups have proposed alternatives and extensions to browsers to address some aspects of this problem. Oostendorp describes the PAINT system (Personalized Adaptive Internet Navigation Tool) [25]. It provides an interface for accessing hierarchies of bookmarks in a style similar to the NEXTStep interface. WebMap is a browser extension that shows a graphical relationship between web pages [11]. Each page is represented by a small circle that can be selected to display the actual page. The links between pages are colored to indicate information about the links, such as whether it is a link to a different server or whether the destination page has already been read. These graphs may be saved and used by others.

While the web is inherently cyclic, it is easier to visualize hierarchies, and so many web visualizations are based on hierarchies extracted from the graph of the web. Some interesting work focuses on alternative visualizations [24]. Furnas [14] shows how *multitrees* can be used to represent a collection of hierarchies sharing parts of the underlying data. One application of multitrees is visualization of bookmarks from multiple individuals[34]. Furnas [16] also describes a framework for characterizing how different structures influence effective view traversal, the mechanical process of moving between information items, and view navigation, finding good paths to information items.

Another approach to visualizing large information spaces that can be applied to web browsing and navigation involves techniques to show detail at particular nodes while maintaining context. One general approach, fisheye views [13], has been extended with graphics [30], three dimensions [9][22], hyperbolic representations[20], animation [10], and zooming [2][3][28]. Other techniques include exploiting a large virtual space [12], using lenses or filters [5][23][31], and visualizing two dimensional layouts [1][21].

In addition to the difficulty of finding information, it becomes ever more important to tailor information for one's own needs. Also rather than searching oneself can be sensible to go by other's recommendations. This is the basis for commercial services such as Yahoo [36], and follows the often effective strategy of exploiting recommendations from those one knows and trusts [35].

Annotations are another important information tailoring facility. Annotations are personal markings that can be used to highlight and comment on information for oneself and others. One interesting approach to annotation on the web separates the annotations from the original documents and stores them in a special annotation server [29]. Used with an enhanced browser, displaying a new page automatically brings in the annotations of others and integrates them into the page.

In the sections that follow, we describe our zooming web browser and the attempt to use animation and multiscale representation of context to support more effective web navigation. In addition to visualization of standard HTML pages, we introduce extensions to HTML that allow more sophisticated presentations and client-side interactions. We demonstrate the beginnings of direct manipulation graphical authoring tools and show how annotation can be supported as a form of authoring. Finally, we present the results of initial user testing and envision a scenario for future web use in the classroom.

2. A ZOOMING WEB BROWSER

Navigating the WWW presents a struggle between focus and context. As one browses or searches the web the need for detailed views of specific items conflicts with the need to maintain a global view of context and history of traversal. This struggle is made more difficult by the haphazard organization of the WWW. Information items closely related by links are not necessarily closely related by content nor in terms of the user's information needs. At times, one seems more likely to find something of interest when not looking for it than when specifically searching. This serendipitous contact with information, though at times frustrating, can also be an advantage. The challenge is how to best support both incidental and intentional access while organizing useful information so that it can be effectively retrieved again in the future.

We are exploring dynamic multiscale techniques to support focus and context during navigation of large information spaces. To accomplish this we are building a zoomable web browser using Pad++, a substrate for building multiscale dynamic user interfaces [2][3][27][28]. Pad++ provides an extensive graphical workspace where dynamic objects can be placed at any position and at any scale. Pad++ supports panning and zooming. Zooming can involve simple geometric scaling or what we term *semantic zooming*, in which rendering of objects can vary based on factors in addition to scale, such as context of the task or complexity of the information being displayed. Pad++ is built as a widget for Tcl/Tk, a scripting language and user-interface library [26][33].

Pad++ allows WWW pages to remain visible at varying scales while they are not specifically being visited, so the viewer may examine many pages at once. In addition, Pad++ allows the user to zoom in and out of pages, enabling explicit control of how much context is viewed at any time. To orient themselves, users can simply zoom back to view a number of web pages. To get more detailed views of a particular page they can zoom in. We think this variable scale contextual display of web pages can provide important support for navigation. We are currently exploring a tree layout system that permits users to dynamically add to and reorganize a tree of web pages. Using our Pad++ web browser, users navigate a space filled with familiar objects, not iconified representations of those objects.

Our dynamic Pad++ tree browser combines a basic focus-driven layout with automatic zooming and panning to support navigation. The software allows the user to select a focus page. That selection animates the page to occupy a larger section of the display. Pages farther from the focus page get increasingly smaller, resulting in a graphical fisheye view [30]. See Figures 1 and 2 for snapshots of the Pad++ web browser during reorganization.

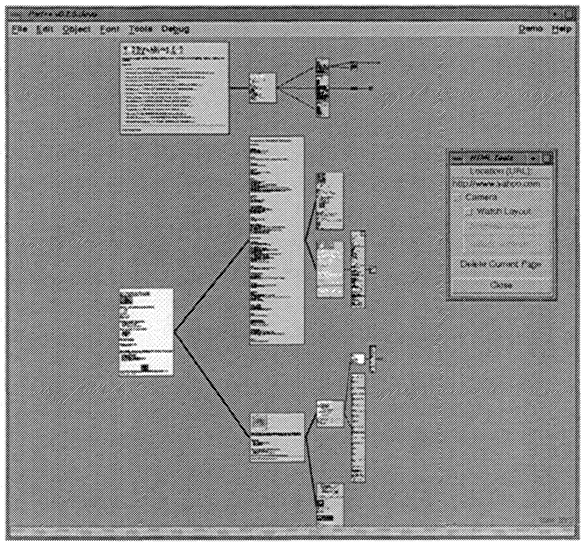


Figure 1: Snapshot of Pad++ Web Browser.

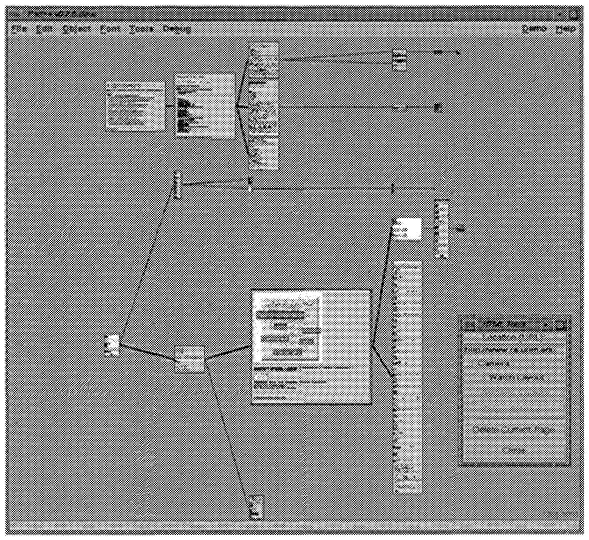


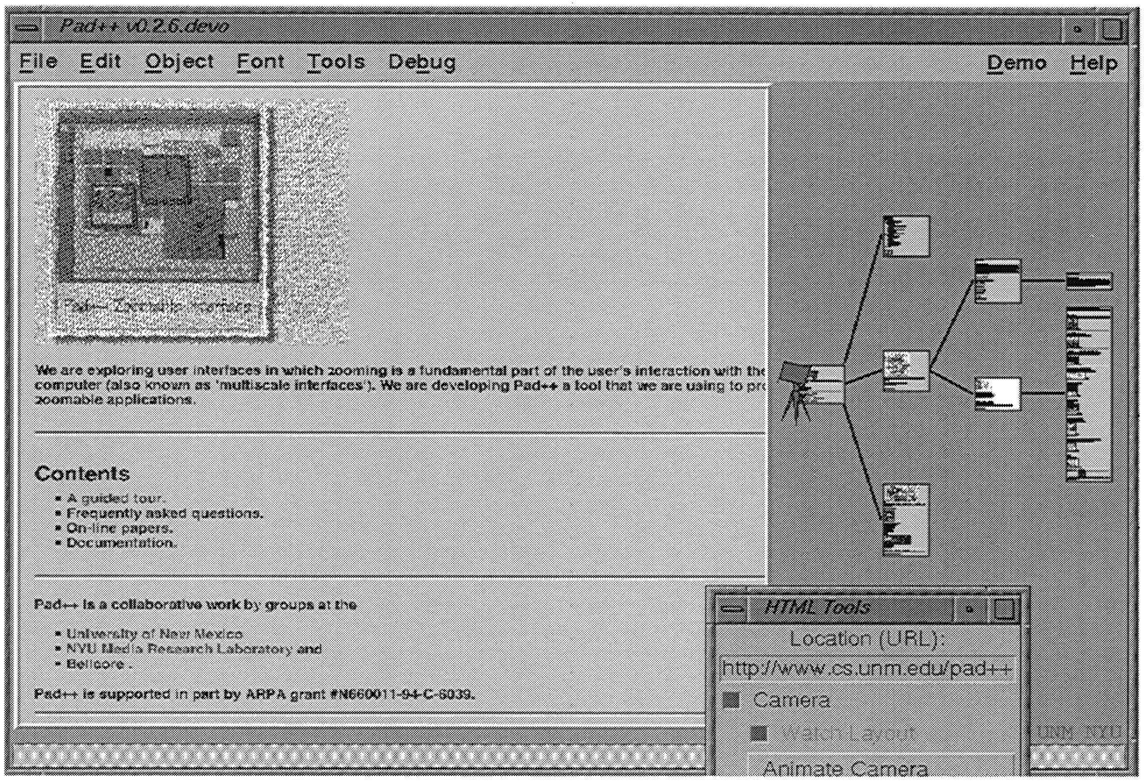
Figure 2: Another view of same web pages.

The Pad++ WWW browser combines Pad++'s interactive multiscale display with dynamic objects that can restructure themselves in response to user actions. Clicking on a link brings up a new page, adds it to the tree of pages, and causes the tree to restructure itself. Unlike other web browsers that immediately replace the current page with a new page, the restructuring process is animated so that users can understand how the tree is being reorganized. The animation helps maintain object constancy and the graphical depiction of links highlights relationships between pages. The new page becomes the current focus and is moved to the center of the screen, at a size suited for viewing. The user may designate any existing page to be the current focus by clicking on it.

As in earlier fisheye displays, our basic layout function assigns a *degree-of-interest value* to each node in the tree based on its distance from the focus page. We define the distance to be the shortest path between two pages[13]. This value is then used to determine the size of each node. See [11] for a description of other hierarchical layout techniques not based on fisheye views.

The layout described above provides a sense of context while following links. We have also implemented an alternative *camera mode* of navigation. It shows the web of links on one side of the screen with a zoomed in view of the focus page on the other side of the screen. A camera is depicted along with the web of pages. The camera can be dragged around or automatically animated through the web. The zoomed in view shows the page the camera is currently looking at (Figure 3). This mode also supports automated tours. For example, one type of camera can take you on a tour of all the parts of your saved web pages that have changed since you last looked at them.

We are currently experimenting with more flexible mechanisms for dynamic tree layout and interaction. These include exploring alternative visualizations and better methods for managing and interacting with large dynamic trees. New tree layout methods will work with any kind of item on the Pad++ surface. Thus, in addition to HTML pages, users will be able to create spaces using any Pad++ object, including drawings, interactive maps, and text.



We are exploring user interfaces in which zooming is a fundamental part of the user's interaction with the computer (also known as 'multiscale interfaces'). We are developing Pad++ a tool that we are using to pr zoomable applications.

Contents

- A guided tour.
- Frequently asked questions.
- On-line papers.
- Documentation.

Pad++ is a collaborative work by groups at the

- University of New Mexico
- NYU Media Research Laboratory and
- Bellcore .

Pad++ is supported in part by ARPA grant #N660011-94-C-6039.

Figure 3: Camera view of web pages.

The new layout code is designed to be hierarchical, so that users may designate subtrees to have different layouts. This allows greater freedom in grouping and display. For example, a certain information tree may contain nodes with subtrees consisting of hundreds or thousands of nodes each. These nodes could exploit a hyperbolic layout to compress the information and the hyperbolic nodes themselves might be layed out radially [20].

Another topic we are exploring involves tradeoffs between maintaining pointers to information on the Web and making copies of the information locally. For example, a user might want to copy items from a remote page to prevent that information from being lost. At other times users may wish to maintain only pointers to information since it is being maintained elsewhere. There are interesting related issues of annotation, that we discuss later, as well as issues of maintaining annotation placement as pages change.

3. AUTHORING WEB PAGES

Thus far, we have discussed visualization and navigation of WWW pages written using standard HTML. We are also exploring extensions to HTML to allow web page authors access to Pad++ multiscale visualization and layout facilities. This extension, *Multi-Scale Markup Language* (MSML), enables users to include arbitrary Pad++ objects in web pages. We have implemented MSML using the HTML *<Meta> tag* and thus added features are invisible to HTML browsers.

Important motivations for MSML are to make WWW pages and their components first-class Pad++ objects and allow authors using MSML to exploit all Pad++ facilities. In addition to adding zooming and other dynamic features to web pages, the goal is permit authors to manipulate and interact with any web page element. Making elements first-class Pad++ objects will result in authors being able to move, scale, delete, add, or modify them. Our approach is similar in spirit to the SELF project [32]. While much remains to be accomplished to support the full informational physics [3] we envision, all of the examples detailed below work in the current Pad++ Web browser.

HTML provides very few basic types. Examples include text, images, bullets items, and horizontal rules. MSML in contrast, supports not only HTML types but also provides access to all the graphical features of Pad++. This gives users a richer toolkit of objects to use when creating documents: text (Postscript Type 1 fonts), lines, rectangles, ovals, lenses (providing filtering and alternative representation), portals (furnishing additional zoomable views of the Pad++ surface), compound objects created from these basic elements, and access to Pad++'s dynamic zooming and panning facilities.

Zooming and Scale

HTML provides header tags, <H1>, <H2>, etc., to indicate degree of importance of sections in WWW documents. However, this mechanism only works with text, not with other media such as images. MSML introduces a method to control the size of all types of objects, including images and graphics. We have used this, for example, to create a multiscale hierarchical outline (see Figure 4). All MSML extensions are written using special Meta-tag keys. This approach makes it clear that the included code is an extension and allows it to simply be ignored by standard HTML-based web browsers.

A portion of the MSML required to create the outline depicted in Figure 4 is given below. The *pad_scale* key takes a single argument that multiplies the current scale and affects the size of all future objects until a */pad_scale* key is seen.

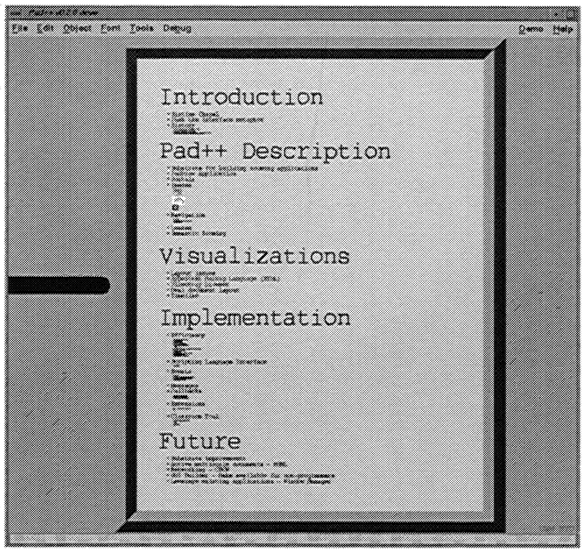
```
<html>
Introduction

<meta pad_scale=0.25>
<ul>
<li>Sistine Chapel
<li>Push the interface metaphor
<li>History

<meta pad_scale=0.25>
<ul>
<li>Ivan Sutherland, SketchPad, 1963
<li>William Donelson, MIT, 1978
<li>George Furnas, Fisheye Views - Bellcore, 1986
<li>Ken Perlin, David Fox, PAD - NYU, 1993
</ul>
<meta /pad_scale>

</ul>
<meta /pad_scale>
...
```

Another example use of MSML involves inclusion of a multiscale state map on a web home page. As the view is zoomed in, first counties, then cities, and then street maps of cities are shown. Finally, even the location of one's home or work could be indicated on the street map. It would even be possible to continue zooming until a floor plan of home or work location becomes visible. See Figures 5-7 for a sequence of snapshots as we zoom into New Mexico. We first see county names and ultimately an Albuquerque street map.

Below is the MSML code that produced the examples in Figure 5-7:

```
<html>
This is the New Mexico Map page
<hr>
<meta pad_tcl={msml_load_tcl http://www.cs.unm.edu/~bederson/pad/county.tcl county}>
```

The map data is stored in a separate code file and is loaded using the `pad_tcl` tag. It passes a Tcl script that uses the MSML library function *msml_load_tcl*. This function takes two arguments: a URL to a Pad++ Tcl script and a tag name to be associated with every object the script creates. It is via this tag that objects are associated with the HTML page.



Introduction

Pad++ Description

Visualizations

Implementation

Future

Figure 4: Outline using MSML Scale Tag.

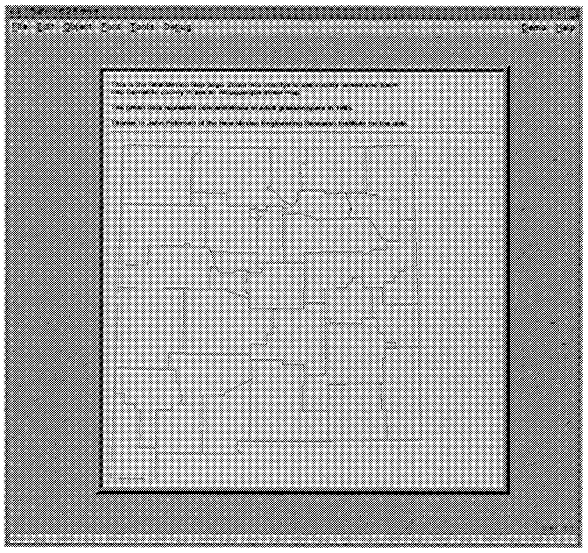


Figure 5: New Mexico county data.

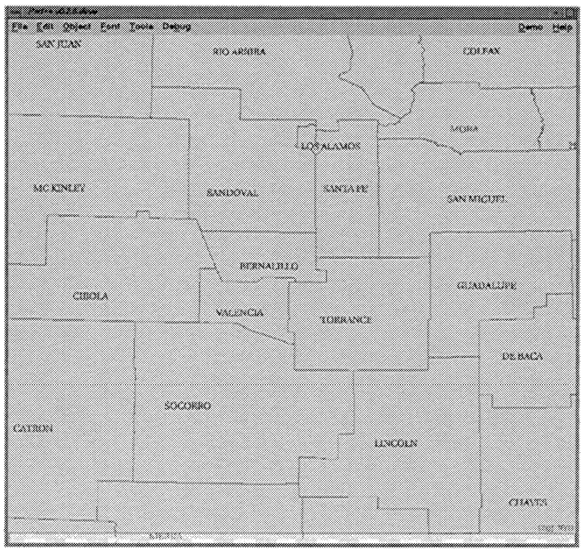


Figure 6: NM zoomed into county names.

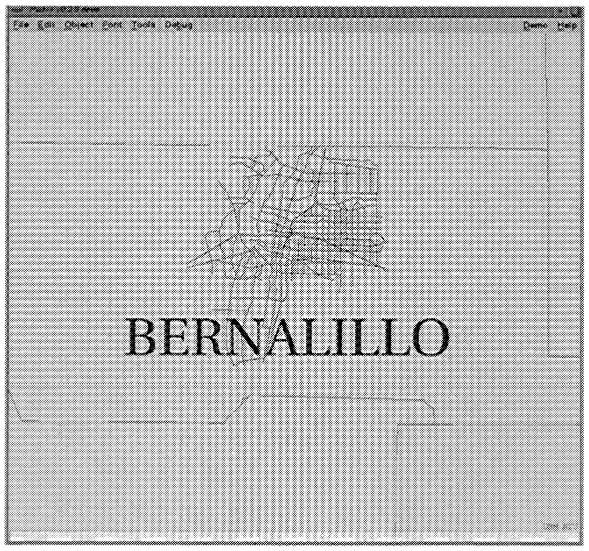


Figure 7: NM zoomed into Albuquerque.

The URL specifying `county.tcl` contains Tcl code that creates the county data. The `msml_load_tcl` function inserts it into the Web page and properly scales it. One advantage of MSML is that almost any object on the Pad++ surface can be used as an anchor, not just text and images. So once a user has zoomed into the New Mexico state map and located Albuquerque, the homes of the members of the Pad++ group as well as the Computer Science department itself could be anchors to other WWW pages, or to other points on the map, such as those of collaborators from other parts of the world.

Interaction

Traditionally, browsers come with pre-defined functions and all interactions with a web document are constrained to those functional abilities. Limited animation is possible through techniques such as *server-push* and *client-pull*. Forms, a simple interface built into most clients to collect information and send it to the server for processing, provide constrained GUI-like interactions. However, until recently, nothing supporting more interactive and flexible interfaces has been available.

Currently there are a number of efforts to create more interactive WWW documents. The primary approach is to write code in a programming language, instead of HTML, that can be downloaded into a browser equipped to interpret the language. Sun's HotJava project uses the Java language [18], Cygnus Support's GNU Remote Operations Web (GROW) proposes to use GNU's Guile extension language [17], and Microsoft's Blackbird will use dynamically loadable object files [6]. By providing the ability to download and run code locally allows complicated animations, for example, to be encoded in a concise and network efficient manner.

The Pad++ WWW browser contains a full Tcl interpreter. MSML provides mechanisms to include Tcl code within the page, instruct the browser to download either scripts or saved Pad++ data files over the WWW, and pass the code to the Tcl interpreter. This, of course, carries with it enormous security risks that we have not yet addressed. In the future we may restrict ourselves to a safe subset of the language as with Java or GROW. In our case this will initially be SafeTcl [7] but may also include support for other languages (e.g. Java).

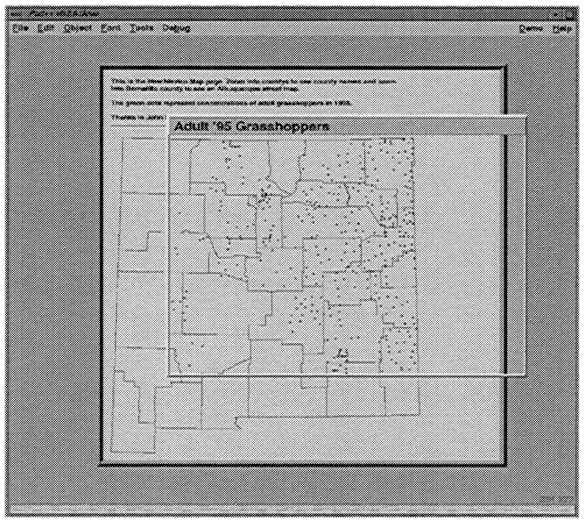


Figure 8: The grasshopper filter applied to the New Mexico County data.

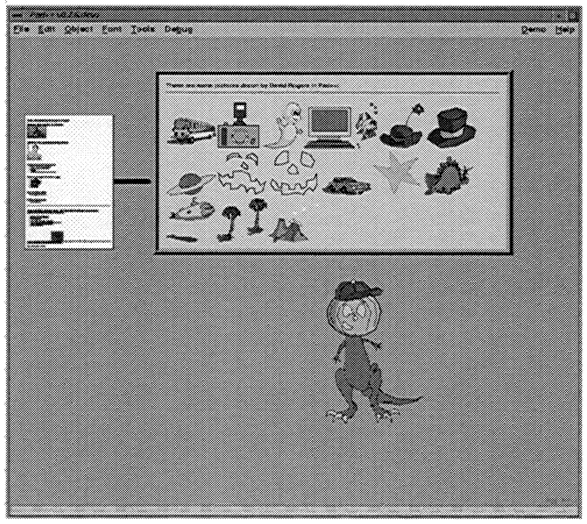


Figure 9: A WWW page containing Pad++ data files with some elements dragged out.

Lenses

Pad++ lenses can change the way objects look on the Pad++ surface [2][5][31]. In MSML, they can be particularly useful. For example, the data used to create the map of New Mexico consists of about 50K bytes of vertex data just to define the county boundaries and the city locations. There are many different kinds of data that one could be interested in displaying in relationship to a map of New Mexico (e.g., demographics of the local populations, geographical features of interest, etc.) If one were interested in not only providing an accurate street map to the Computer Science Department but also, say, an accurate count of the 1995 grasshopper population across the state, it would be pointless to include the geographical vertices twice. Instead, the same map can be used, and lenses supplied such that when viewed through the appropriate lens only the information of interest is visible. Figure 8 demonstrates the use of such a lens.

Annotations and Authoring

Because MSML allows users to interact with all elements of a WWW page, it provides a unique opportunity to explore annotating and authoring of WWW pages. Based on experiences working with children, we have come to appreciate the need for simple and intuitive ways to author WWW pages. One very natural authoring mechanism is to directly use and modify existing components of others peoples' web pages. Every web page then becomes a potential supplier of components.

We have begun to implement a drag-and-drop interface for authoring WWW documents. Figure 9 shows a page containing several Pad++ drawings. Just under that page, a composite figure was created by dragging elements out of the top web page (a kit of components constructed for young users to author zoomable hypertext stories) and putting them together. Our approach is to allow creation of web pages using the same direct manipulation techniques. In addition, we expect to provide layout support to facilitate creation of aesthetically effective web pages.

A direct extension of this authoring technique allows users to maintain local copies of WWW pages with added annotations. A goal, much in keeping with the kind of personal information environment Vannevar Bush envisioned, is to enable users to create, save, and share annotated databases of WWW pages. This is similar to graphical hotlists but with the important difference that with MSML, users can maintain not only the HTML data for a particular page, but also associate arbitrary Pad++-based annotations. Pad++ supports a variety of annotations: not only can text be added, but entire WWW pages can be added to other WWW pages creating hierarchical meta-pages; graphics can be added, for example, to indicate an interesting section of a particular page, even when that page is scaled very small. Our goal is to provide rich annotation facilities by combining Pad++'s dynamic multiscale annotation ability with a control system that supports creation of virtual documents in which documents and annotations are separately controlled and maintained. See [29] for an example of one such control mechanism.

To support effective views of collected web pages it is important to be able to show modifications over time. If one cares about a particular set of pages enough to include them in a personal collection it is likely that notifications of modifications will also be of value. We are exploring mechanisms to highlight changes. One technique highlights, by changing color or bounding with a rectangle, sections changed since some specific date (for example, the last time you viewed the document). Another uses scaling to show the history of changes. Older versions are shown at increasingly smaller scale. Marking changes is similar in spirit to efforts conducted by [35] using the WebWatch program. Unlike that effort we conjecture that one might not merely want to know that a document has been altered but also the details of the changes.

4. USER TESTING RESULTS

In November of 1995, we completed a pilot test of our zooming web browser. There were 14 test participants from the University of New Mexico community, equally split between the College of Education and the Computer Science Department. A majority used email regularly, but almost no one used the WWW as frequently. Over a third of the participants from the College of Education had never used the WWW. All of the Computer Science participants had used the WWW. Before the pilot test began, a 5-minute demonstration of our browser was shown to participants. Participants were then asked to use the browser for 30-60 minutes and subsequently completed a survey about their initial impressions.

We found users to be overwhelmingly positive about using the zooming web browser. We received such comments as:

> *"I think it's different than any other browser, and a lot more interesting to use."*
>
> *"It will take some getting used to, but to have the ability to create a tree of where you were is a great advantage."*
>
> *"It's easier and more friendly than Netscape. It is a little slow and jerky too."*
>
> *"I found it a very useful browser, and liked the hierarchical tree structures that are created. There is no need to click 'back' or 'forward' like you do for Mosaic or Netscape."*
>
> *"Easy access to previously viewed pages-- excellent way to view pages larger than the screen."*

The survey also asked users to select one or more entries from a list of possible short descriptions of their experience. The results are given in Table 1:

SHORT DESCRIPTION OF ZOOMING WEB BROWSER	# OF PARTICIPANTS WHO SELECTED IT (out of 14)
Interesting	12
Would try using it again	11
Enjoyable	7
Useful	7
Exceptional	4
Frustrating	3
Confusing	2
Would never use it again	0

Table 1: User Study results

When test participants were asked to describe what they most liked about using the zooming web browser, over 70% of the participants said they liked seeing the tree of where they were in the WWW and navigating by zooming. When participants were asked to describe what they liked the least, they commonly mentioned the speed of interaction. Users would like to have faster browsing tools, as well as the ability to delete a page from the tree structure. Based on this feedback, we have since modified the browser. In the newest release, WWW pages are displayed several times faster than during the pilot test and users now have the option of deleting any viewable page on the tree.

In the pilot test survey, participants were also asked how they thought the zooming web browser compared with other WWW browsers. Of the 14 test participants, a little over half felt qualified to respond. (A handful of the College of Education participants had never used another browser and therefore could not make a comparison.) The participants who responded were overwhelmingly more positive about the zooming web browser than Netscape or Mosaic. They felt that the browser had a better visual layout and was generally easier to use than other browsers. Surprisingly, only one participant felt that Netscape was easier and less buggy.

In summary, the results of our pilot test offer positive support for and constructive feedback about the use of zooming in a web browser. This feedback continues to inform our browser development efforts. We expect to continue testing with a more diverse population of users to better understand the problems and advantages issues associated with zooming.

5. A USER SCENARIO FOR THE FUTURE

With the technologies we are currently creating, we can foresee a time in the future when the following scenario will come to pass:

> It is morning. David Brooks enters his classroom. In an hour his fifth grade students will join him, but until then, David sits down at his computer, coffee in hand, to scan his favorite web pages. David begins by wandering the local museums' home pages. He knows that today he and his students will begin a thematic unit on dinosaurs. Before they arrive, he quickly drags various dinosaur images and text from different web pages, and creates a new student page. He decides that his page looks more like a wall of graffiti than a planned document. He wishes he had more time to animate the dinosaurs, design information lenses, and establish zooming links to other home pages. Suddenly it dawns on him, those would be great things for his students to do! He breaths a sigh of relief, sips his coffee, and waits for his students to arrive.
>
> Once all 21 students have been welcomed, David explains that thanks to their insistence, they will now turn their energies to learning about dinosaurs. The students clap and cheer. Once they settle down, David splits them up into design teams

of 3, and asks them to move to their computers. On each student team's screen is an image of David's dinosaur page. David explains how bad this page is, and asks his students to help him redesign it. He asks them to create animations, lenses, and links to other pages. The students eagerly work on their projects.

A week later, the student teams present their work to the class. The first group to go presents a page with four simply drawn dinosaurs. When a student zooms into the web page, the dinosaurs begin to move about. One of the students points out that they found information on the web that described how these dinosaurs moved, so they designed their beasts with this in mind. The team zooms in on one dinosaur, a tyrannosaurus rex. As they zoom in, the picture of the dinosaur disappears and text information appears. The student team explains that by selecting the highlighted word in the text it will take you back to the original WWW page that the text came from. As they explain, the text zooms out, a new tree link is formed, and a new page is zoomed in on the screen.

After much applause the next student team presents their work. They zoom in on their WWW page to display one large dinosaur. They explain that their project gives you lots of information on just one dinosaur. They begin by dragging various lenses from the side of their page. Each lens is shaped like the information it displays. The large tree-shaped lens when dragged over the dinosaur shows the types of vegetation the dinosaur eats. The sun-shaped lens shows the kinds of climates this dinosaur likes to live in. The team explains that by zooming in with any of the lenses it will take you to a WWW page with more information. They demonstrate this and start to uncover a tree of information.

David is impressed with his class's work. As each team presents, they offer more creative solutions than he thought possible. At the end of class he decides to ask the school principal if he can publish his students' research projects on the WWW. Not only does she agree, but she explains that the local museum has been looking for WWW pages created by students. She points out that this would be just the thing for their WWW section entitled: "A Kid's Tree of Knowledge" that is being designed to commemorate the fiftieth anniversary of Vannevar Bush's *As We May Think* article.

6. CONCLUSION

The World-Wide Web has become an important and widely used resource. Because of this, it is crucially important to address its usability. We have shown one promising technique based on zooming to better support web navigation. This technique was used to implement a prototype web browser that was found to be appealing to users. We illustrated extensions to the WWW authoring language to enable creation of more dynamic and interactive multiscale documents. Finally, we demonstrated how a direct manipulation authoring environment allows users to construct and modify web pages using items from existing pages.

Pad++ and the zooming Web browser will be made generally available in the near future. To find current information, send mail to pad-info@cs.unm.edu, or look at <URL: http://www.cs.unm.edu/pad++>.

7. ACKNOWLEDGEMNTS

We acknowledge generous support from ARPA's Human-Computer Interaction Initiative (Contract #N66001-94-C-6039). The test subjects from UNM's College of Education and Computer Science Department gave us valuable feedback. We appreciate the work of our collaborators at the New York University Media Research Lab, especially that of Jon Meyer, in helping us develop Pad++.

8. REFERENCES

[1] M.D. Apperley, I. Tzavaras, and R. Spence. "A Bifocal Display Technique for Data Presentation", *Proceedings of Eurographics '82*, 27-43.

[2] Benjamin B. Bederson and James D. Hollan. "Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics", *Proceedings of ACM Symposium on User Interface Software and Technology (UIST'94)*, 17-26.

[3] Benjamin B. Bederson, James D. Hollan, Ken Perlin, Jon Meyer, David Bacon, and George Furnas. "Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics", *Journal of Visual Languages and Computing* (in press).

[4] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. "The World-Wide Web", *Communications of the ACM*, August 1994, 37 (8), 76-82.

[5] Eric A. Bier, Maureen C. Stone, Ken Pier, William Buxton, and Tony D. DeRose. "Toolglass and Magic Lenses: The See-Through Interface", *Proceedings of ACM SIGGRAPH Conference (Sigraph'93)*, 73-80.

[6] "Blackbird", <URL: http://www.microsoft.com/developer/intdev/bbdsht.html>.

[7] Nathaniel S. Borenstein. "EMail With A Mind of Its Own: The Safe-Tcl Language for Enabled Mail", *ULPAA '94, Barcelona, 1994.* <URL: http://minsky.med.virginia.edu/sdm7g/Projects/Python/safe-tcl/ulpaa-94.txt>

[8] Vannevar Bush. "As We May Think", *The Atlantic Monthly*, July 1945, 101-108.

[9] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. "The Information Visualizer, an Information Workspace", *Proceedings of ACM Human Factors in Computing Systems Conference (CHI'91)*, 181-188.

[10] Bay-Wei Chang and David Ungar. "Animation: From Cartoons to the User Interface", *Proceedings of ACM Symposium on User Interface Software and technology (UIST'93)*, 45-55.

[11] Peter Doemel, "WebMap - A Graphical Hypertext Navigation Tool", *2nd International Conference on the World-Wide Web*, Chicago, IL, 1994, 785-789.

[12] William C. Donelson. "Spatial Management of Information", *Proceedings of 1978 ACM SIGGRAPH Conference*, 203-209.

[13] George W. Furnas. "Generalized Fisheye Views", *Proceedings of 1986 ACM SIGCHI Conference*, 16-23.

[14] George W. Furnas and Jeff Zacks. "Multitrees: Enriching and Reusing Hierarchical Structure", *Proceedings of ACM SIGCHI'94*, 330-336.

[15] George W. Furnas and Benjamin B. Bederson. "Space-Scale Diagrams: Understanding Multiscale Interfaces", *Proceedings of ACM SIGCHI'95*, 234-241.

[16] George W. Furnas. "Effectively View-Navigable Structures", Presented at *HCIC '95 Workshop*, Snow Mountain Ranch, Colorado: Human Computer Interaction Consortium Workshop, February, 1995.

[17] "Grow - The GNU Remote Operations Web", <URL: http://www.cygnus.com/tiemann/grow>.

[18] "Java: Programming for the Internet", <URL: http://www.javasoft.com/>.

[19] Wendy A. Kellogg and John T. Richards. "The Human Factors of Information on the Internet", *in Advances in Human Computer Interaction, Volume 5*, ed. J. Nielsen, Ablex Press, 1-36.

[20] John Lamping, Ramana Rao, and Peter Pirolli. "A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies", *Proceedings of CHI'95 Human Factors in Computing Systems, ACM press*, 401-408.

[21] Henry Lieberman. "Powers of Ten Thousand: Navigating in Large Information Spaces", *Proceedings of the ACM User Interface and Software Technology conference (UIST'94)*, (short paper), 15-16.

[22] Jock D. Mackinlay, George G. Robertson, and Stu K. Card. "The Perspective Wall: Detail and Context Smoothly Integrated", *Proceedings of CHI'91 Human Factors in Computing Systems, ACM press*, 173-179.

[23] Jock D. Mackinlay and George G. Robertson. "The Document Lens", *Proceedings of the ACM User Interface and Software Technology conference (UIST'93)*, 101-108.

[24] Sougata Mukherjea, James D. Foley, and Scott Hudson. "Visualizing Complex Hypermedia Networks through Multiple Hierarchical Views", *Proceedings of CHI'95 Human Factors in Computing Systems, ACM press*, 331-337.

[25] K. A. Oostendorp, W.F. Punch, and R.W. Wiggings, "A Tool for Individualizing the Web", *2nd International Conference on the World-Wide Web*, Chicago, IL, 1994, 49-57.

[26] John K. Ousterhout. *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[27] "Pad++", <URL: http://www.cs.unm.edu/pad++>.

[28] Ken Perlin and David Fox. “Pad: An Alternative Approach to the Computer Interface”, *Proceedings of 1993 ACM SIGGRAPH Conference*, 57-64.

[29] Martin Roscheisen, Christian Mogensen, and Terry Winograd. “Beyond Browsing: Shared Comments, SOAPS, Trails, and On-line Communities”, Unpublished paper, Computer Science Department, Stanford University, Stanford, CA, USA.

[30] Monojit Sarkar and Marc H. Brown. “Graphical Fisheye Views”, *Communications of the ACM*, 37 (12), December, 1994.

[31] Maureen C. Stone, Ken Fishkin, and Eric A. Bier. “The Movable Filter as a User Interface Tool”, *Proceedings of ACM SIGCHI’94*.

[32] David Ungar and Randy B. Smith. “Self: The Power of Simplicity”, *Proceedings of OOPSLA* ‘87, 227-241.

[33] “The Tcl/Tk Project at Sun Microsystems Laboratories”, <URL: http://www.sunlabs.com/research/tcl/>.

[34] “VRML - Virtual Reality Modeling Language”, <URL: http://www.vrml.org/>.

[35] Kent Wittenburg, Duco Das, Will Hill, and Larry Stead. “Group Asynchronous Browsing on the World Wide Web”, *Proceedings of the 4th International World Wide Web Conference*, Boston, MA, International WWW Conference.

[36] “Yahoo”, <URL: http://www.yahoo.com/>.

Pad++ Bederson - 4

SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 28(10), 1101–1135 (AUGUST 1998)

Implementing a Zooming User Interface: Experience Building Pad++

BEN BEDERSON,*[1] AND JON MEYER[2]
[1]*Computer Science Department, University of Maryland, Human-Computer Interaction Lab, 3171 A.V. Williams Building, College Park, MD 20742, USA*
(email: bederson@cs.umd.edu)
[2]*Computer Science Department, New York University, Media Research Lab, 719 Broadway, 12th Floor, New York, NY 10003, USA*
(email: meyer@cs.nyu.edu)

SUMMARY

We are investigating a novel user interface paradigm based on zooming, in which users are presented with a zooming view of a huge planar information surface. We have developed a system called Pad++ to explore this approach.† The implementation of Pad++ is related to real-time 3D graphics systems and to 2D windowing systems. However, the zooming nature of Pad++ requires new approaches to rendering, screen management, and spatial indexing. In this paper, we describe the design and implementation of the Pad++ engine, focusing in particular on rendering and data structure issues. Our goal is to present useful techniques that can be adopted in other real-time graphical systems, and also to discuss how 2D zooming systems differ from other graphical systems. © 1998 John Wiley & Sons, Ltd.

KEY WORDS: Zooming User Interfaces (ZUIs); real-time computer graphics; animation; 3D graphics; windowing systems; User Interface Management Systems (UIMS); Pad++

INTRODUCTION

For several years, we have been investigating an alternative user interface paradigm based on zooming. In our approach, users navigate over a single large information surface. Documents can be placed on the surface at any position, and also scaled to any size. Navigations (including pans, zooms and hyperlinks) smoothly animate the view so the requested document appears at the right position and size.

Zooming User Interfaces (ZUIs) are exciting to us because they present a possible solution to problems which plague previous approaches to user interfaces. ZUIs present information graphically and exploit people's innate spatial abilities. Detail can be shown without losing context, since the user can always rediscover context by zooming out. ZUIs use screen real estate effectively, and have great potential

* Ben Bederson carried out much of the work presented in this paper at the University of New Mexico, Albuquerque, NM 87131, USA.
† Pad++ is available for non-commercial (educational, research and in-house) use from http://www.cs.umd.edu/hcil/pad++

CCC 0038–0644/98/101101–35$17.50 *Received 2 June 1997*
© 1998 John Wiley & Sons, Ltd. *Revised and Accepted 16 March 1998*

even on small screens. One way of thinking about ZUIs is that all the information you need is there if you look closely enough.

To explore these kinds of interfaces, we built a zooming graphics engine called Pad++, which brings together a unique combination of features, and supports smoothly animated zooming of large datasets using off-the-shelf PC-class hardware.

In this paper, we describe the design and implementation of the Pad++ engine, focusing in particular on rendering and data structure issues. Our goal is to present techniques that can be adopted in other real-time graphical systems, and also to show some of Pad++'s unique characteristics and describe how 2D zooming systems differ from other real time graphical systems. We have written elsewhere about applications we have built using Pad++.[1–3]

Background

Zooming interfaces have a long intellectual history. Over 30 years ago, Ivan Sutherland showed the first interactive object-oriented 2D graphics system.[4] This visionary system, called Sketchpad, demonstrated many components of today's interfaces. It even provided rudimentary zooming: every drawn object could be scaled and rotated. Fifteen years later, the Spatial Data Management System (SDMS)[5] used zooming as an integral part of the interface. SDMS was the first system to use zooming through a two dimensional metaphor for finding information. SDMS supported two semantic levels—an 'Overview' and a 'Zoom into Icon' level. However, SDMS was implemented with custom hardware, and consisted of several machines and displays integrated into a room, with all of the controls organized around a single large chair.

Eleven years later, a system called Pad[6] was developed and presented at an NSF workshop in 1989. Pad integrated zooming into a single program that ran on inexpensive hardware. Pad was programmable, and supported interactive text editing, drawing, documents and portals (views onto different areas of the work surface). It ran in black and white, and was entirely based on bitmaps, so drawings and text became pixelated as users zoomed in. Pad ran on Sun 3 computers, which do not provide enough processing speed to support continuous smooth zooming, so the interface only allowed users to zoom in and out by powers of two. Subsequently, we built Pad++, a direct but substantially more sophisticated successor to Pad.[7,8]

Zooming has been a component of other interface research as well, although not as a primary focus. Several researchers have investigated full 3D interfaces, and these interfaces have implicitly used zooming, since when a user moves close to an object, that object appears bigger. A system developed at Xerox PARC called the Information Visualizer[9] made extensive use of 3D, and showed several applications which took direct advantage of the difference in scale available in 3D.

We are aware of three other implementations of ZUIs. Two are specialized commercial products for World Wide Web navigation. They both use radial layouts to represent hierarchies of information.[10,11] The third, Tabula Rasa,[12] is a Scheme-based implementation created by David Fox in his PhD thesis research at NYU.

Other related systems include Self,[13] ARK (Alternate Reality Kit)[14] and Xerox Rooms.[15] Like virtual window managers such as fvwm,[16] these systems present the user with a 2D planar data surface which is much larger than a single screen, and which the user navigates by panning, or by clicking on an iconic map. However, they don't support smooth continuous zooming.

Requirements

Conceptually, ZUIs present the user with a zoomable view of a large information surface. The surface is populated by graphical objects—some of which are simple shapes (e.g. lines, images, text items), whereas others may be procedural (e.g. charts, animated objects, multiscale objects). The ZUI manages rendering and interaction with objects on the surface. It also handles resource allocation (colors, fonts, etc.). In this sense, a ZUI is very similar to a graphical window management system like X Windows, except that:

(a) The size of a ZUI data space is not limited by the size of the screen, but instead by the precision of the numeric format used to store coordinates.
(b) ZUIs must scale up to handle tens of thousands of objects, whereas windowing systems need only support a few thousand windows (this figure is naturally constrained by the limited resolution of raster displays).
(c) Objects are typically not rectangular, and they may be semi-transparent. Windows are expected to be both opaque and rectangular.
(d) Coordinates are floating point, rather than integer based.
(e) There may be many visible views of the surface through *portals* (we discuss portals in more detail later on, but briefly, they are objects on the data surface which show other areas of the surface, and allow interaction with the remote area through the portal). Windowing systems typically manage a single view.
(f) Rendering is double-buffered by default, since unbuffered pans and zooms produce a distracting screen flicker. Windowing systems let applications control screen pixels more directly.

Windowing systems typically provide applications with a blank window to paint on, and rely on the application itself to draw the contents of each window. This approach is well suited to a wide range of applications, since it gives applications a great deal of flexibility—each application can decide how much effort to spend organizing rendering, performing culling and clipping, handling resources, etc. The downside of this approach is that, in many cases, the code for culling, clipping, colormap handling, resource management, window updating and event handling is duplicated from application to application.

In Pad++, because objects can be non-rectangular and semi-transparent, and also because of portals, handling rendering, culling and clipping is more complex than it is under a windowing system. Consequently, by default, Pad++ performs these operations on behalf of the application. If applications have specialized needs, they can use procedural objects, which let the application handle rendering and clipping issues explicitly.

To help understand these differences, we started constructing a list of basic technical requirements that we felt our zooming user interface should meet. This list has evolved over time, since some of the requirements emerged from experiments involving Pad++. Even so, many of the items on our requirements list have been there since the outset of the project. The requirements list represents the technical problems that Pad++ aims to solve, so we present it here to underpin subsequent discussion. It is not meant as a formal definition of the requirements of a ZUI.

Zooming interface requirements

- *Maintain and render at least 20,000 objects with smooth interaction*: the number 20,000 is somewhat arbitrary, but we felt that this would give us the freedom to implement the kinds of interfaces we were imagining. Maintaining smooth real-time interaction is crucial. The entire metaphor is based on animation. If the system becomes slow and jerky, the metaphor dies. Frame rates of anything less than 10 frames per second are unacceptable. (Here, and in the rest of the paper, we use the term *object* to mean a graphical entity that is manipulated as a whole by the system. This might be a simple object such as a poly-line segment, or a compound object such as an HTML page composed of many characters, line segments and images).
- *Animate all transitions*: all screen changes, whether a change of view or an object movement, should be animated. Since the interface is based on navigating through a surface, it is important to give as much feedback as possible to users about where they are within this space.
- *Use off-the-shelf hardware*: we wanted to make this zooming engine widely available, and to have a wide range of people using Pad++. Therefore, we were obligated to use readily available hardware. In particular, we haven't used graphics accelerator cards or alternate input devices. Our reference platform is a 200 MHz Pentium Pro running Linux.
- *Support high quality 2D graphics*: since Pad++ is a graphical user interface system, it is crucial that the graphics are high quality. The engine must support good fonts, high quality images, transparency, rotation, and other graphical effects, matching or exceeding the capabilities already found in windowing systems.
- *Provide rapid prototyping facility*: ZUIs are new, and much of our work involves experimenting with variations of interfaces. As such, it is important for us to be able to quickly modify visualizations and applications. This requirement led to effort connecting Pad++ to various scripting languages so we could create applications through an interpreter.
- *Support rich dynamics*: in addition to zooming, our envisioned ZUI system would include the following features. Objects should be able to have different visual representations at different sizes. That is, zooming into an object should be able to automatically show more detail. We call this *context-sensitive rendering*. it should be easy to animate an object within the space or to animate the view. It should also be possible to create *lenses*—objects which when dragged over other objects change the visual representation of the object seen within the lens. Finally, we required *layers*, which provide a mechanism to easily change the visibility and drawing order of groups of objects.
- *Support rich navigation metaphors*: in addition to text, images, vector graphics, and hierarchical groups, ZUIs require a few special object types to support navigation. In a ZUI, objects sit at a specific place in the flat space, and yet it is sometimes necessary to be able to view two objects at the same time that are far apart. We use *portals* to solve this problem. Portals are objects on the information surface that look onto another part of the surface. Portals can be used to implement lenses. Also, when the view changes, objects on the screen normally move with the view. Sometimes it is useful to have an

object stay in the same position relative to the screen. We support this through *sticky* objects.

- *Support standard GUI widgets*: to build a complete interface system, a ZUI must also support user interface widgets to match those found in existing GUIs. Buttons, sliders, scrollbars, menus, etc. must all be accessible in the same zooming environment as the other graphical objects.
- *Offer a framework for handling events*: writing applications with zooming, layers, hierarchical groups, portals and sticky objects can be tricky. The event model must be rich enough to gracefully deal with all of these object types, and also simple enough so it is easy to use.
- *Run within existing windowing and operating system*: the ZUI must offer ways to work with existing applications as well as with new zooming applications. It should support established standards, such as X Windows, UNIX, and Microsoft Windows 95/NT, so that users can continue to use their current applications as well as zooming applications.

Having sketched the basic requirements of our ZUI, we next look at implementation issues. We will first look at rendering, then at visible object determination, and finally, at the overall structure of the Pad++ software.

PART I: RENDERING IN ZUIs

A large part of the work needed to build a ZUI involves developing a zooming renderer. Before looking in detail at how the Pad++ renderer works, we first present ZGA (Zooming Graphics Accelerator), an imaginary hardware graphics accelerator designed to support zooming user interfaces. The ZGA feature list was constructed by looking at existing graphics hardware and eliciting the features we felt were applicable to ZUIs (based on the requirements list we presented earlier). We describe ZGA here so as to offer a reference point to compare other graphics platforms against. After outlining the features of ZGA, we then go on to describe existing 3D and 2D graphics systems, showing how they overlap with the feature set of ZGA, and where they differ.

Our imaginary ZGA card features:

1. *Text*
   (a) High quality antialiased text which can be transformed and scaled rapidly.
   (b) Support for a wide range of fonts, include Type1 and TrueType.
   (c) International character set support.
2. *Lines*
   (a) A rich set of line drawing styles, including rounded ends, bevels, mitering, and dashes.
   (b) Scaleable line width and semi-transparent lines.
3. *Images and Movies*
   (a) Hardware-accelerated image scaling, preferably using filtering to produce smooth results.
   (b) Support for MPEG and QuickTime digital movies which can be scaled to any size and played at 30 frames a second.
   (c) Images are maintained and rendered in standard system memory (not specialized video memory) to support paint applications, and applications that present many images at once.

4. *General*
    (a) Two 24 bit color buffers (for double buffering), a 24 bit depth buffer, an 8 bit alpha buffer for transparency, and a 32 bit accumulation buffer for special effects.
    (b) Floating point coordinate system with support for affine transforms.
    (c) Clipping, including clipping to arbitrary 2D polygons
    (d) Fast rasterization of arbitrary 2D polygons.
    (e) Level-of-quality control over rendering routines for text and images (so the ZUI can trade quality for speed when system resources become overburdened).
    (f) Double buffering hardware which supports partial redraws and hardware pans.

In hardware terms, the ZGA graphics board is not outlandish. In fact, at first glance, many of these features are present on 3D graphics boards. A natural step is to try to capitalize on the graphical power of modern 3D graphics hardware when building ZUIs.

3D graphics hardware

Over the last 20 years, there has been a significant investment in technology to support real-time 3D interactive environments for use in virtual reality, visual simulations, games, information modeling and visualization, and other areas. This work has lead to the development of a number of software standards, such as OpenGL and VRML, as well as to cheap off-the-shelf 3D hardware.

These interactive 3D systems offer considerable graphical power. A high-end system can draw millions of polygons a second, and includes hardware support for anti-aliasing, double-buffering, texturing, lighting and other effects. A home PC with a modest 3D graphics card can support scenes containing rich textures, lights, thousands of polygons, and heart-pumping interactivity (as demonstrated by current games such as DOOM and its successors).

To what extent can alternative 2D systems such as ZUIs benefit from the 3D hardware/software now available? At first glance, real-time 3D systems appear to face many of the same technical challenges as a 2D ZUI such as Pad++. Both must maintain high frame rates, perform animated navigation, handle scene management, clipping, and event propagation, and deal with large numbers of objects. It seems natural to base a ZUI implementation on a 3D graphics API, such as Inventor or OpenGL.[17] An advantage of this approach is that the ZUI can take advantage of hardware acceleration when it is available.

In practice, while 3D programming interfaces such as OpenGL present a good starting point, they have a number of design traits that make their use in ZUIs challenging. For basic 2D graphical elements (such as polygons, lines, images and text), 3D APIs are often either overly rich (and hence wasteful of limited system resources), or have feature gaps (which require a slower software-based solution). Both traits require additional coding on the part of the programmer to produce good 2D results. We discuss some of these challenges below. The discussion is based upon our understanding of the OpenGL rendering API. Other 3D programming interfaces may differ.

Text, images and lines

Current 3D graphics cards can draw large numbers of polygons and lines a second. They often support features such as alpha blending (for transparency), clipping and antialiasing in hardware. However, these systems are optimized for rendering convex 3D polygons (notably triangles and quadrilaterals). Rasterizing arbitrary 2D polygons (for which there are well-known hardware optimizations) is not generally considered part of the 3D API. For example, rendering 2D non-convex polygons in OpenGL must be done by tessellating the polygons into triangles and quads, and then rendering the resulting geometry. This is not as efficient as scan converting the 2D polygons directly.

3D APIs tend to provide only a limited set of line styles. OpenGL offers some basic line drawing primitives, including thick lines and line stipples. However, the thick-line support in OpenGL draws multiple segment lines as a sequence of single segment lines. There is no support for controlling how lines are joined (such as mitering or beveling the joints). There is also no guarantee that pixels within a thick multi-segment line will only be drawn once. On the hardware we've tried, semi-transparent multi-segment lines look unattractive. The limitations with lines in OpenGL can be overcome by rendering line segments as polygons, although this makes line drawing more costly and also complicates the implementation.

For text, consider that a single page of text represents over 100,000 polygons. This is already near the limit of what a typical desktop computer can render at interactive rates. Rendering several pages of text at once (for example, in a zoomed out view) by drawing each polygon would be very slow. It is possible to draw each typeface into texture memory, and then use texture mapping as a way to generate scaled text in 3D.[18] For systems with hardware accelerated texture mapping, large quantities of text can be quickly handled using this technique. Unfortunately, texture memory is usually a scarce resource, so only a fairly small set of fonts can be supported in texture RAM. Also, this only works well for small point sizes—for large characters, rendering from polygon outlines produces cleaner results. In practical terms, implementing a rendering engine that can draw large quantities of readable text in many typefaces and sizes is not trivial.

Texture mapping can also be used effectively to scale 2D images. Images rendered from texture RAM can be drawn quickly at any scale and orientation. 3D graphics cards often provide filtering hardware for scaling textures, which produces good looking results. However, as we mentioned above, texture memory is currently a scarce resource in 3D graphics hardware. Handling many images at once (or handling editable images) requires a good 'texture residence' mechanism, which is not trivial to implement. For ZUIs, it is desirable to be able to zoom images held in normal process memory. OpenGL does have 2D image drawing capabilities which support scaling, but on all the platforms we have tried the image scaling is integer based, and doesn't work well for high scale factors.

3D graphics hardware availability

None of the challenges listed above prevent us from implementing ZUIs using OpenGL. However, the number of current machines that provide hardware acceleration for OpenGL graphics is still small. If ZUIs are to become popular, they must also run reasonably effectively on hardware that is in people's homes today. This means that they must run well on VGA-level graphics cards that only support 8-bit graphics. Eventually, graphics cards with feature sets similar to hypothetical ZGA card may be built. Until then, we must rely on techniques for supporting ZUIs on cheap VGA cards.

2D graphics hardware

In the last section we looked at some of the problems of using a 3D graphics system to implement ZUIs. What about using existing 2D graphics hardware instead?

Of course, existing 2D graphics systems are designed to handle 2D graphical elements such as lines, text and images. Surely they will be ideal for zooming user interfaces also?

In reality, 2D graphics APIs make heavy use of caching. They are designed to draw graphical elements repeatedly at a fixed scale. For example, the data structures provided in most windowing systems for fonts and images cache device-level information about how the font or image appears at a given scale. Generating this information for the same font or image at different scales is too slow to achieve continuously animated zooming.

Despite these limitations, we built Pad++ using X windows, a 2D graphics system, to give us a wide base of computers to run on. We utilize a number of techniques designed to overcome the shortcomings of X Windows, and meet our goal of smoothly animated pans and zooms. We discuss how Pad++ handles fonts and images below, and also discuss our approach to screen management. We are currently working on writing a Windows 95/NT version of Pad++.

Text in 2D

We have experimented with a number of techniques for drawing text in Pad++. One approach is to use the font capabilities built into the underlying windowing system. Unfortunately, most windowing systems utilize bitmap fonts, which are hard to scale continuously. Worse, each font takes up many kilobytes of memory. The overhead of generating a bitmap font at any given size is a significant number of milliseconds, because the windowing system must generate raster images for all the characters in the font at the given size. Once the font has been rasterized, rendering characters in the font is quick. However, the memory requirements for a large number of fonts at all possible scales are prohibitive, especially when you consider the very large font sizes possible in Pad++ (for example, when a character is scaled up to fill the whole screen).

Microsoft Windows offers TrueType fonts, which are stored in outline form and are therefore scaleable, but generating high quality characters (especially for smaller font sizes) from outline fonts is hard, and outline fonts are also slower to render than bitmap fonts. Windows applications still rely on generated bitmaps for rendering large quantities of text, and rendering high quality small text.

In Pad++, to produce fast zooming fonts, we initially used a simple line font. In this font, each character consists of one or more multi-segment lines, with fewer than 10 segments per character. Lines are quick to render, so this approach is fast, but the characters must be hand-designed and have few curves. The characters look unattractive compared to bitmap based fonts.

As a next step, we obtained an Adobe Type 1 font decoder and attached it to the polygon renderer in Pad++. This generates only a single non-convex polygon per character, but is still too slow for large quantities of text, and the characters suffer from aliasing artifacts (e.g. uneven stems, holes and 'pimples') when they are small, which makes small text hard to read (see Figure 1). There are outline font renderers that overcome the limitations of our simple polygon renderer, but they are slower.

To address the speed problem, we implemented a lightweight font cache mechanism. A *fontcache* contains 96 cells, representing the ASCII character set. Each cell holds a 100×100 pixel bitmap. The fontcache mechanism remembers what character size and typeface appears in each cell bitmap. To render a character, say the 'A' character, from a given typeface, the font cache mechanism looks in cell 65 (the ASCII code for 'A') and checks if the character drawn in the cell is the right typeface and size. If it is, then the bitmap is copied directly to the screen. If not, then the cell is cleared, the polygon form of the character is obtained and rendered into the cell, and then the cell is copied directly to the screen.

In Pad++, all the fonts used with the renderer share two global fontcaches. It would be possible to allocate a separate fontcache for each font, though this is expensive since each fontcache requires 96 bitmaps that are 100×100 pixels (implemented with a single 9600×100 bitmap, approximately 100 kilobytes in size).

The fontcache mechanism accelerates rendering of small-sized text, which contains repeated uses of a character in the same font. Since for English text documents and program code this is a frequent occurrence (see Figure 2), the speedup effects of the fontcache for many documents are appreciable. On our reference platform, a sample text document took 367 ms to render (2.7 frames per second—fps) with no font cache. With the font cache enabled, the same document took 68 ms (nearly 15 fps)—a speedup factor greater than five. On other systems, even greater speedups have been observed. Also, note that the overhead of changing the font size and typeface is also low, since only the characters that are actually rendered are placed in the cache.

An improvement to the fontcache might be to provide more cells in the cache for frequently used characters, observing that E is more frequent than T, which is more frequent than A, etc. We haven't explored this approach fully.

The fontcache lets Pad++ zoom text quickly, but the text is still hard to read at small sizes. Our solution for certain fonts (in particular Times and Helvetica) at

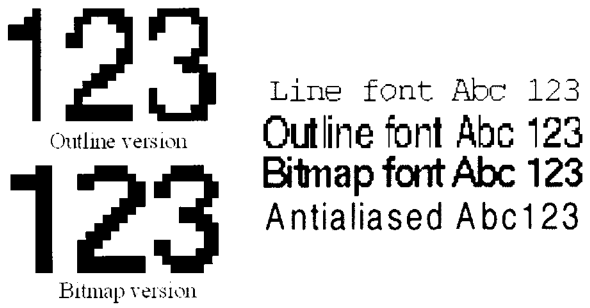


Figure 1. Various font rendering styles. Pad++ uses line fonts, bitmap fonts and outline fonts. It does not currently support antialiased fonts

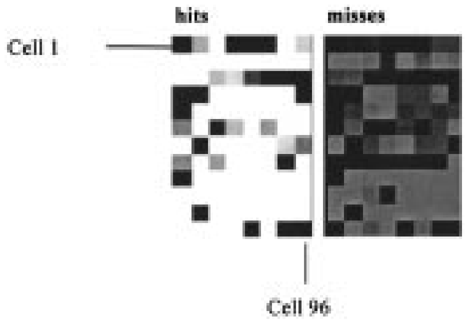


Figure 2. Fontcache hit/miss statistics for a piece of C++ program text. The left image depicts cache hits for character cells in the cache. The right image indicates cache misses for the same cells. The hit/miss count is reflected by brightness. Overall, there are more misses than hits. Most misses are in the higher numbered cells, representing characters 'a'–'z'

small sizes is to switch to X bitmap fonts during refinement (refinement is discussed in the section on '*Screen Management*'). As mentioned earlier, loading bitmap fonts is slow, but during refinement slow rendering is less important. This technique is only used for a few fonts (to keep memory usage down), and is not used for rotated text.

Note that if we use a bitmap font (or other windowing system font), we must discard the default spacing metrics for the font, and instead compute the spacing for each character explicitly, using floating point coordinates, based on the font metrics of the outline font. If we don't do this, text tends to 'jiggle' as it is zoomed, and the outline and bitmap fonts do not line up consistently. This is because bitmap fonts come only in sizes of one pixel increments, which is too coarse for smooth zooming, and because the font metrics are bitmap fonts is usually hand-tuned to produce more readable (but less mathematically accurate) letter spacing.

Eventually, we hope that there will be more hardware support for outline fonts and font antialiasing. Until then, text will remain a challenge for ZUIs, and multi-solution approaches such as that adopted by Pad++ will be necessary to achieve reasonable text performance.

Images in 2D

In this section we look at how Pad++ zooms images. We do not consider storage issues, or multiscale image representations, but discuss only how to render a moderate size image held in RAM.

The image scaling required in Pad++ is basically a constrained version of texture mapping. 3D graphics systems not only scale images, they also have to rotate and shear the image to account for perspective transforms, and map them onto complex polyhedra. The Pad++ requirements are much simpler, since images are only transformed using translation and scale.

The easiest image scaling algorithm magnifies images using pixel replication and shrinks them using pixel decimation. A simple version for this is:

```
Image zoomImage(Image src, float scale) {
 Image dst = new Image(src.width * scale, src.height * scale);
```

```
  for (int y = 0; y  dst.height; y++) {
   for (int x = 0; x  dst.width; x++) {
    dst.data[y][x] = src.data[(int)(y / scale)][(int)(x / scale)];
   }
  }
  return dst;
 }
```

However, a literal implementation of this does not produce fast results. Before becoming discouraged and looking for a more sophisticated technique for handling real time image scaling, consider how this code can be optimized. Examining this implementation at the machine code level, every iteration contains approximately 10 instructions:

1	Loop increment
1	Loop test
2	Divides
2	2D array conversion (source)
2	2D array conversion (dest)
1	Memory lookup
1	Memory storage
⇒	Total ~10 instructions

An optimized version of this algorithm takes advantage of the fact that each row is mapped in the same way. We can compute a lookup table for mapping the X coordinates of the first row, and reuse it for all the rows in the image. Further optimization can be achieved by ‘unrolling’ the inner loop—so that the overhead of incrementing the loop variable and testing is greatly reduced. In C++, by using pointers instead of 2D arrays, the algorithm runs faster still. The scaling code now looks something like:

```
// MAKE A PRECOMPUTED TABLE FOR MAPPING ROW X VALUES
int table = new int[dst.width];
for (int x = 0; x < dst.width; x++) {
 table[x] = (int)(x / scale);
}
// SCALE THE IMAGE
for (int y = 0; y < dst.height; y++) {
 long *srcPtr = src.data[y];
 long *dstPtr = dst.data[(int)(y / scale)];
 int *tablePtr = table;

 // SCALE THE ROW (UNROLLED)
  for (x = 0; x < dst.width & ~7; x += 8) {
   dstPtr[0] = srcPtr[tablePtr[0]];
   dstPtr[1] = srcPtr[tablePtr[1]];
   dstPtr[2] = srcPtr[tablePtr[2]];
   ...
```

```
        dstPtr[7] = srcPtr[tablePtr[7]];
        dstPtr += 8;
        tablePtr += 8;
       }
       // FINISH THE ROW (-NOT- UNROLLED)
       for (; x < dst.width; x++)
        *dstPtr++ = srcPtr[table[x]];
     }
    }
```

A rough machine instruction count for the inner loop of this version is:

	2	Memory lookup
	1	Memory storage
⇒	Total ~3 instructions	

This is about three times faster than the original version of the code, and produces fast real time image zooming with reasonable performance. With this algorithm Pad++ can zoom an 800×600 pixel image on a 200 MHz Pentium Pro at 30 frames per second.

We've ignored all the special cases: stippling images (for semi-transparent images), color dithering for simulating 24 bit color on 8 bit graphics cards (in Pad++ dithering is done during refinement), images with transparency masks, etc. To keep the image renderer fast, rather than adding statements to the inner loop code, we duplicate the scaling algorithm for each of the various combinations, leading to 16 versions of the same code, each with minor variations. This is unaesthetic but fast.

Decimation/Replication does not produce the best-looking image scaling. One problem is that the pixels tend to ripple as the image is scaled. MIP-mapping and bi-linear or tri-linear filtering, such as is performed in hardware by high-end 3D cards, produces much smoother results. In the future, this style of hardware is likely to become readily available on home PCs. Hopefully, simple 2D image scaling operations will also be supported in hardware by these cards.

Rotation

To save the overhead of adding a general-purpose transformation to the renderer, we add special code to rotate each object type. This code off-loads rotation computation from the renderer to the time of rotation. Thus, the Pad++ renderer maintains a transformation stack of just translation and scale, but does not include rotation. Polygonal objects are rotated by transforming the points. Images are rotated by computing a new rotated image from the original. Text, however, is rotated at render time, and thus rotated text is slightly slower than non-rotated text.

Screen management

Consider a 1280×1024 pixel screen. If the user is inserting a single text character in a small text object visible on the screen, it is undesirable to redraw the entire screen-full of information for just this minor change. Rather than redrawing the entire display, it is better to focus rendering resources on only the areas of the display that have changed. To do this, applications must maintain information about what areas of their display are out of date and perform rendering operations clipped to these areas. We call this technique 'screen management'.

Existing windowing systems do determine when windows need refreshing (e.g. because they have been moved or resized), and they also clip rendering operations to windows (so changes to one window do not effect other windows). In this sense, windowing systems carry out primitive screen management on behalf of applications. But windows are assumed to be rectangular and opaque (X Windows includes an extension to support non-rectangular windows, but there is a performance penalty for using it), and windowing systems still leave it up to the application to manage the contents of each window. To handle large windows effectively, applications must implement their own screen management schemes.

To help applications carry out screen management, all the common windowing systems also provide clipping primitives that use Shape algebra,[19] which lets applications clip 2D graphic operations to arbitrary polygonal shapes. Shape algebra is useful because it supports efficient clipping and boolean operations. Using Shape algebra, applications can implement sophisticated screen management schemes.

In a ZUI such as Pad++, screen management is further complicated by three factors. First, objects can be non-rectangular and semi-transparent, so a change to an object may require re-rendering the objects in front of and behind it. Secondly, rendering is double-buffered, so objects can never just directly modify pixels on the screen, but must coordinate with the double-buffering mechanism. Thirdly, because of portals, a change to an object may require several different parts of the screen to be redrawn.

Pad++ uses shape algebra in conjunction with a 'damage and restoration' scheme to keep the display up to date.

Damage and restoration

The Pad++ screen management system is based on a painting metaphor. The display is treated as a 'painting', and changes to objects visible on the display are seen as 'damage' to the painting. A 'restorer' is an object whose contract is to fix up damage. Each restorer maintains a Shape specifying what area of the display to work on, and an integer level indicating what refinement level to work at (more on refinement in a moment). At any time, a number of restorers may be working on different areas of the painting, each at different refinement levels. The system ensures that no two restorers are working on the same portion of the screen—newer restorers always receive priority over older restorers (their Shape is subtracted from the Shape stored in earlier restorers).

When objects change, they register damage on the Pad++ surface. Objects register damage in object coordinates by calling the *Damage* routine, passing it a bounding box. When damage occurs, Pad++ searches for a restorer with refinement level 0 to register the damage with, creating a new restorer if necessary. Once a restorer is obtained, Pad++ adds the bounding box to the Shape record within the restorer. In principle, it is desirable to maintain damage information in floating point surface coordinates. In practice, because Pad++ uses the Shape algebra utilities provided by the underlying windowing system, damage coordinates are always converted to integer screen coordinates before being added to the Shape.

Notice that some changes to objects may generate multiple calls to the Damage routine—for example, when an object is moved, it generates damage for both the old and the new bounding box of the object.

When the system becomes idle (when all the events in the input queue have been handled), any restorers in the pending queue are removed from the pending queue and 'run'—causing the appropriate regions of the display to be re-rendered. Applications can also force pending restorers to be rendered, for example during animations.

Damage through portals

Portals complicate damage, since a single object may be visible on the screen in multiple portals at difference scales. The system must also handle the case of damage to an object which is visible through a portal looking at a portal looking at the object (and so on). Damage to objects visible through portals must be clipped correctly to the portal's screen bounds.

A simple approach is to perform an exhaustive depth-first search every time an object is damaged, checking each portal to see if the damage is visible. This is too slow in practice, and doesn't scale up to support large numbers of portals. (Notice that the damage recording mechanism must be fast. During an animation, many objects may be changing every frame, generating large numbers of damage requests. If every damage request involves a large search operation, frame rate will drop significantly.)

A better approach is to maintain a data structure indicating, for each object, which portals can see that object. The damage routine can then just register damage with each portal directly. A problem with this approach is that maintaining the per-object portal lists is expensive. Every time an object is moved, the system must examine all of the portals to check if the object is visible (directly or indirectly) in the portal. The data structure must also be updated every time a portal's view is changed.

A good compromise is to have each view (i.e. each portal or top-level window) maintain a list of the portals visible within that view. Then, when damage occurs, the system can quickly determine which portals are visible on the screen and recursively check only those portals for damage. The visibility list is comparatively lightweight to maintain, since it only involves work when a view is changed, and not when other objects are changed.

A further optimization would be to divide portals into high-priority and low-priority portals. High-priority portals are used for areas which the user can actually interact with (main displays, editor windows, lenses, etc.), and which must be kept up-to-date. Low-priority portals are used for static non-interactive information displays (bookmarks, icons, snapshots, etc.). Whenever damage occurs, all the visible low-priority portals are scheduled for repainting after a time-out of one second, regardless of the visibility of the specific object in those portals. High-priority portals register damage recursively as described above. With this mechanism, high-priority portals are always kept up-to-date (but require more work to damage). Low-priority portals can become out-of-date, but they incur a smaller penalty during animations. We are currently implementing this last optimization.

Sticky objects

In a ZUI, it is sometimes important to have objects stay in a fixed location on the screen, and to not pan and zoom with other objects on the surface. These *sticky* objects are useful for bookmarks, cut buffers, status lines and other interface components. How should these sticky objects be handled?

Ideally, a hardware solution such as overlay planes (found, for example, on SGI workstations) would be used for sticky objects. Overlay planes let two distinct frame buffers be mapped to a single window. The hardware includes a mechanism for indicating which buffer is visible in each pixel location (e.g. via a special transparent pixel value).

Overlay planes would let Pad++ treat sticky objects and non-sticky objects as two separate planes of information. However, not all systems support overlay planes, and overlay planes prevent sticky objects from being beneath or interleaved with non-sticky objects.

A second approach to handle sticky objects is to maintain two separate scene graphs for each surface—one representing non-sticky objects, the other representing sticky objects. The renderer and event manager must then consult both graphs. The drawback of this approach is that it introduces complexity into the code, since functions must be aware of the two different cases (sticky and zoomable). We tried this approach, but abandoned it once we realized how many duplications this would introduce to the code.

Our current approach is to implement sticky objects as normal Pad++ objects with a simple one-way constraint: whenever the main view of a surface is changed, all the sticky objects on the surface are transformed by the inverse of the view's change. Since the number of sticky objects is generally small (limited screen real-estate offers a natural incentive to keep the number of sticky objects down), this does not impact hugely on the cost of view changes. To the user, the effect is that all sticky objects maintain a fixed location on the screen. Internally, sticky objects are maintained in the same scene graph as non-sticky objects.

An advantage of this approach is that it allows objects to be sticky only in one dimension, e.g. objects that pan left and right, but always stay the same size (we call these *sticky-z* objects). The notion of constraints makes it easy to customize the specific behavior of the sticky object.

Level of refinement

During pans and zooms, the main role of the display in a ZUI is to provide information as quickly as possible to orient the user—giving them visual cues indicating where in the space they are looking. But when the view is stationary (e.g. because the user is reading a document), text and diagrams should be as legible as possible. These two different modes of viewing introduce a design conflict, since producing legible text at arbitrary scales is a compute-intensive operation, and during pans and zooms you want to spend as little computation time as possible.

One solution to this problem is to render the contents of the display using fast low-quality algorithms during animated pans and zooms, and then use higher quality rendering algorithms to redraw the display when it has been stationary for a period of time.

To support this approach, restorers in Pad++ can work at different *Levels of*

Refinement (LOR). At LOR 0, the restorer draws objects as quickly as possible. At higher LORs, restorers draw with increasing detail, although they take longer to complete. In practice, only LORs 0, 1 and 2 are used.

When a restorer has finished work, it can schedule another restorer to redraw the same region at a higher LOR—allowing areas of the display to be 'refined' (see Figure 3). Restorers determine whether refinement is needed by querying each rendered object to see if it has more detail available. If an object has further detail available, its bounding box is added to a 'refining restorer'. At the end of the render, the refining restorer is scheduled to run. With this mechanism, a change to an object on one area of the screen may cause that area to iterate through a number of refinements. Changes to other areas of the screen do not interfere with this refinement process, but instead initiate new refinement processes. Damage to areas that overlap with a refining restorer cause that area to be removed from the shape managed by the refining restorer and rescheduled for rendering by a new restorer using LOR 0, potentially canceling the refining restorer if it becomes empty.

Refinements can be visually jarring to the user. For example, when switching from LOR 0 to LOR 1, small text alters its appearance from dashed lines ('greeked' text) to individual letters. The change in appearance can be distracting. To resolve this problem, Pad++ uses a dissolve transition to update areas of the screen as they are refined. Although this effect slows down the total time taken to refine a region, it also reduces screen flashing, so the overall impression is an improvement.

Interruption

Adding refinement to a system introduces a new problem. Refinements may take a considerable time to complete. For example, rendering a large image using color dithering takes nearly a second on a mid-range PC. During this time, the user may decide to interact with one of the objects on the surface, or pan to a different location. Forcing the user to wait for refinements to complete is undesirable.

In Pad++, refinements are interruptible. Periodically during a refining render, the system checks for input events (keyboard or mouse button events). If an event is

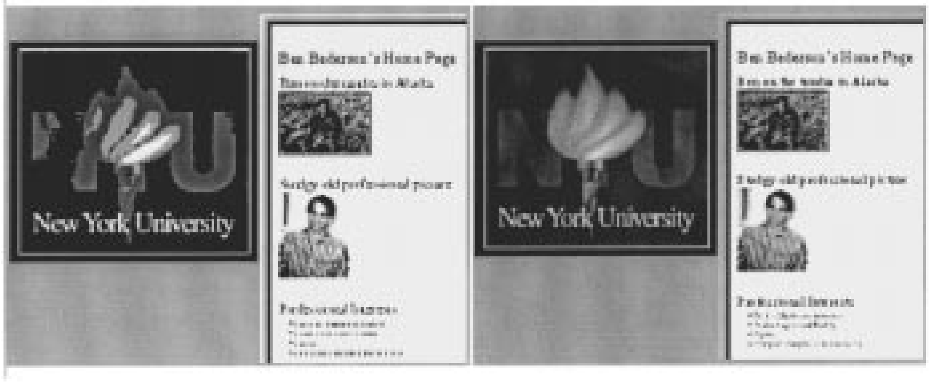


Figure 3. Before and after refinement. The left view shows the scene before refinement. The right view shows the same scene after refinement. Greeked text is replaced with fully rendered text. Images are refined using color dithering. Refinement is introduced using a dissolve effect, to reduce screen flashes

found on the input queue, the renderer finishes rendering the back buffer at LOR 0, leaves the front buffer unmodified, and schedules a new restorer to come back during idle time to redraw the region. Then the event is processed as normal.

Fast panning

A second problem with refinement is that when the user pans or zooms the whole display, it jumps back to refinement level 0. This introduces a distracting flash to the display, although this is less distracting than the flash produced by successive refinements, since it is user-initiated, and not initiated by the system during idle time. Still, considering that the display may contain an entire page of text rendered at a high refinement level, it is desirable to preserve this image at high refinement level when the user performs a small pan (e.g. to read the bottom of the page).

Using a BitBlt operation, it is comparatively fast to shift the contents of the back buffer by a number of pixels. Then restorers can be scheduled to redraw the strips at the edges of the back buffer that are out of date. With this approach, much of the display contents can be retained for incremental panning operations. A secondary advantage is that, for complex scenes, there is a performance improvement (since only the objects in the exposed strips need to be rendered, and not the whole display).

There are three complications to consider. First, if there are any sticky objects, the areas under the sticky objects must be damaged both before and after the BitBlt, so that the sticky objects are rendered in the new location. Secondly, any outstanding restorers must also have their Shape records updated to reflect the new screen coordinate system. Thirdly, there will inevitably be some aliasing problems. For example, consider a pan by 1.1 ‘pixels’ to the left. This is a legal operation in Pad++, which uses a floating point coordinate system. BitBlts, on the other hand, always move data a whole number of pixels. After ten successive pans by 1.1 pixels, the objects on the screen will have moved 10 pixels, whereas Pad will think they have moved 11. Tearing and other visual artifacts will become apparent.

To address this problem, after performing any BitBlt operations on the screen, Pad++ schedules a restorer to redraw the entire screen after one second of idle time.

PART II: VISIBLE OBJECT DETERMINATION

Having a powerful rendering engine is important, but rendering resources are always finite. For datasets containing only a few dozen objects this is unlikely to be a problem, but for datasets containing tens of thousands of objects, a more intelligent approach is needed.

A basic implementation of a ZUI would simply draw every object every frame, and rely on the clipping mechanisms provided in the graphics hardware to avoid modifying pixels that are outside the current clipping region. However, clipping is compute-intensive, and as the number of objects increases this approach becomes too slow. For a ZUI that must scale up to handle tens of thousands of objects, the approach is unworkable.

A slightly better approach is to test each object for visibility, and only draw objects that are visible in the current view. In this way, if an object is not within the current view (or outside the current clip region), it does not consume any rendering resources. In most applications, only a small fraction of the total objects in a world are visible at a usable size in a given view, so this approach can significantly improve frame rates. Frame rates also improve if only a small fraction of a view needs to be redrawn.

Bounding boxes are often used to perform these coarse visibility checks. Testing of two bounding boxes intersect is cheap. A Pentium can performs hundreds of thousands of 2D bounding box intersection tests a second. So for medium sized datasets, performing bounding box checking alone is enough to yield good frame rates. Tabula Rasa[12] and the original Pad implementation[6] both use this approach. However, since the time spent doing visibility checks is linearly proportional to the number of objects in the world, as the number of objects increases, there will be a point where the system spends more time checking than rendering, and frame rates drop.

For very large datasets, a faster mechanism for determining which objects are visible within a given view is essential. Once such a mechanism is available, it is also useful for event processing, since determining which objects lie underneath a point is essentially a constrained version of the same type of query.

Common approaches

There are several popular approaches to handle visible object determination. Windowing systems typically only manage a relatively small number of windows, and so don't need special purpose mechanisms for performing visibility tests. Windowing systems are also hierarchical by nature—top-level windows enclose sub-windows, which enclose sub-sub-windows, and so on. This enforced segregation means that windowing systems can quickly eliminate whole trees from consideration, further speeding up visibility checks. By comparison, ZUIs need to support datasets in which there is no rigorous enclosure hierarchy—such as you might find in a map.

ZUIs are more akin to 3D systems, in which objects can be distributed anywhere throughout space. The distribution of objects within space depends on the specific application, but there are several classes of applications that are instructive to look at. Two representative types of 3D systems are vehicle simulators and architectural walkthrough systems. Vehicle simulators typically have strict frame-rate requirements, and have objects distributed fairly uniformly through space, frequently lying on a large two-dimensional surface.[20] Architectural walkthrough systems on the other hand, tend to have a large number of objects concentrated in a small three-dimensional space.[21] In most 3D systems, there can be a fairly wide range of sizes of objects, as both large structures and fine details must be represented.

Object visibility in 3D systems is determined by what is called the *viewing frustum*.[22] This is the truncated pyramid of space that can be seen from the current view position. It is truncated by *near* and *far* clipping planes—so objects that are too near or too far from the view are clipped. Often, some kind of spatial indexing data structure is used to efficiently determine which objects overlap the viewing frustum. In addition, some systems use knowledge about the model to eliminate objects from consideration. For instance, in an architectural walkthrough of a room on one floor, no objects on any other floors are visible, and only a limited number of objects in other rooms on the same floor are visible.[23]

Other 2D systems that maintain large numbers of objects and must quickly determine object visibility include integrated circuit design systems (VLSI) and

Geographical Information Systems (GIS). Both VLSI and GIS systems manage very large numbers of 2D objects and support limited forms of zooming. VLSI and GIS systems also contain objects with a wide range of sizes.

The visible-object determination needs of a ZUI are similar to these 2D and 3D systems. In all cases, the basic problem is to quickly determine the set of objects that overlap a specified view. The differences between 2D and 3D approaches to this problem are not great, since most spatial indexing data structures can accommodate scenes of arbitrary dimensionality.

However, ZUIs impose two new constraints on the visible object retrieval system: small objects should be eliminated quickly, and objects must somehow be sorted by display-list order. We discuss these two issues below, comparing them with related issued in other graphical systems. Afterwards, we describe how spatial indexing is handled in Pad++.

Object ordering

In a ZUI, objects need to be rendered in a specific order—this order determines which objects are in front of other objects, and which are partially obscured or behind other objects.

Unfortunately, every spatial indexing data structure we found does not return the objects specified by a query in a guaranteed order. This is not a problem for 3D systems, which can render objects in any order and rely on a depth buffer (or 'Z-buffer') to perform hidden surface removal. Order is also not a problem for GIS and VLSI systems because those systems are designed to avoid display order requirements through the use of *layers*. In these systems, objects of different types are put on different logical layers, and each layer is rendered in its entirety before any objects of another layer are rendered. Objects within a layer should not overlap because there is no ability to control the rendering order of objects within a layer. For instance, in most GIS systems, streets may appear on one layer with highways on another layer, and county boundaries on a third layer. When streets do overlap each other, the order in which they are rendered cannot be controlled.

Two-dimensional windowing systems do require that windows get rendered with specified overlapping, but they do not use a spatial index. Rather, they maintain regions for each window specifying what portion of the window is not occluded, and only that portion of the window is rendered when the window contents change—thus maintaining the correct overlapping of windows on the screen. This approach is too costly when there are many thousands of objects, and doesn't handle semi-transparent objects well.

Small object elimination

Objects in a ZUI can differ in size by many orders of magnitude. In a zoomed out view, only the very large objects are visible, and many objects will be so small that they are less than a pixel in size. Spending time rendering these objects is wasteful, especially if rendering time is limited. Ideally, the visible object determination mechanism should quickly eliminate these small objects.

This requirement is similar for some other systems, but not identical. Windowing systems have no such requirement—they render every window unless it is occluded.

VLSI and GIS systems do avoid rendering small objects, but as with sorting, use layers to avoid a per object solution. That is, rather than deciding whether or not to render each object by size, VLSI and GIS systems decide whether to render an entire layer depending on the magnification.

3D systems usually base culling efforts on object position rather than on relative size. Instead of culling objects that are in the current view but are very small, they are often optimized to eliminate objects that are positioned beyond the viewing frustum. Small objects that are within the viewing frustum still get rendered, or at best are checked on an individual basis. This is not sufficient for ZUIs.

Spatial indexing in Pad++

Spatial indexing is the general term for a data structure that encodes spatial characteristics about a set of objects in *n*-dimensional space. These spatial characteristics are then used to provide efficient mechanisms for retrieving objects given spatial queries. One typical query is to retrieve all objects intersecting a given rectangle (for a two dimensional index.) This query can be used in a ZUI to return all objects visible in a given view, or to find all objects that overlap a single point in order to process events.

There are several widely used spatial indexing algorithms (a survey of spatial indexing algorithms and their applications can be found elsewhere.[24,25]) Many are hierarchical, and are based on partitioning space in smaller and smaller segments. Algorithms include R-trees,[26] MX-CIF quad-trees,[27] binary space partitioning trees[28] and k-d trees.[29] Each algorithm has maintenance (insertion and deletion) methods and query methods. Typically, there is a trade-off in time between these methods, where increasing the efficiency of one introduces a higher cost for the other. Consequently, the choice of algorithm depends partially upon whether the scene consists of static or dynamic objects. For static scenes, a higher maintenance cost is usually acceptable, whereas for dynamic scenes the maintenance costs become more critical.

In implementing Pad++, we chose to use an R-tree. An R-tree is a hierarchical structure based on *bounding regions*. Objects are contained in leaf nodes, and internal nodes contain regions that specify the bounds of its children. Each region has a specified minimum and maximum number of children. Regions can overlap, so while an object is only a member of a single region, a query can need to look at several regions at each level. R-trees are similar to quad-trees, except that they partition space into regions based on the objects within the space, and not based on fixed-sized grids. R-trees are balanced so they have a guaranteed maximum depth.

We chose to use R-trees rather than the more advanced R*-tree.[30] R*-trees produce more efficient structuring of regions, but they take longer to maintain. Since visible object determination is not the biggest factor determining rendering speeds (see the ‘Timing Results’ section below), we decided to choose an algorithm which offers effective indexing and very fast insertion and deletion times. This supports our goal to handle very dynamic data spaces, in which many objects move and resize over time. A basic R-tree appears to have lower maintenance costs than the other algorithms we examined. In addition, R-trees are amenable to efficient small object culling during queries.

Object elimination in Pad++

As mentioned earlier, a ZUI typically has overlapping objects with sizes that differ by several orders of magnitude, so it is crucial to quickly eliminate objects that are much smaller than the query window. For Pad++, we chose to ignore all objects that are less than one pixel in both screen dimensions. We cannot eliminate very large objects, because even if they are huge, they may still get rendered and have a large visual effect. Since the hierarchy within R-trees is based on enclosure, we are guaranteed that every child of a node is no larger than the parent node. Thus, if we determine that a given node is smaller than a minimum size, we do not have to descend that portion of the tree. In our implementation, in addition to the rectangle we pass in for each query, we also pass in a minimum size. All queries use this extra parameter to cull sub-trees that are too small during a search.

Even eliminating all non-visible and small objects may not be enough. For example, if the system is performing an animated zoom and attempting to maintain at least ten frames a second, what happens when the number of visible objects in a scene is too high, and there is not enough time to render every object?

A general approach is to render only a partial or fixed set of visible objects during animation frames, and then render the full set of objects during refinements. For example, the system can avoid rendering text labels during animated view changes, and only render them when the view remains stationary for a while. Alternatively, it could render only the first few hundred objects during an animation.

A drawback of this approach is that it doesn't adapt according to the hardware or data. Even for simple scenes containing only a few objects, the user is given an impoverished presentation of the data during the animation. Animations don't scale according to the hardware—yet people with faster computers expect to see more detail during animations.

Instead of adopting a simple, fixed approach to controlling frame rates, a better solution is to include adaptive mechanisms in the renderer. The idea is to monitor frame rates, and use the statistics about prior frame rates to control how much work is done during the current render.

There are several things that can potentially be controlled. The renderer can make decisions about what objects are rendered, culling more objects when frame rates are low. The renderer can also switch to lower-quality rendering algorithms (e.g. algorithms which have a lower level-of-detail) when frame rates drop.

In Pad++, in addition to using a spatial index to eliminate small and non-visible objects, we use frame rate statistics to control how text is rendered and how many small objects are rendered. If frame rates are low, the number of 'small' objects (objects beneath a certain pixel size in their largest dimension) rendered each frame is reduced, and the notion of what constitutes a 'small' object is increased to be more inclusive. Similarly, if frame rates are low, the system switches to using 'greeked' fonts for small text. If, during any frame, a time-limit is exceeded, the system can switch to an emergency mechanism—this renders the remaining large objects using the lowest level-of-detail possible.

Object ordering in Pad++

As described earlier, a query must not only retrieve the objects that overlap the specified rectangle, but it must retrieve those objects in display-list order, so they can be drawn with the correct overlapping behavior.

The easiest solution to this problem (and the one adopted by Pad++) is to simply retrieve the objects from the spatial index unsorted, and then apply a sorting algorithm (e.g. QuickSort). However, sorting can become quite expensive if a large number of objects are visible. Figure 4 shows the time to retrieve and sort objects in a scene depicting a map of New York City. In this case with a scene of over 2000 objects, the sorting time took longer than the retrieval when the number of objects to be sorted became greater than about 400.

We attempted to reduce this sorting time by maintaining the list of objects in each R-tree leaf node in sorted order. We then performed a merge sort, motivated by the fact that we had already done a fair amount of the sorting work since the objects at each node were sorted. Unfortunately, this approach did not result in sorting any faster than using the system QuickSort function on all the visible objects. It appears that the overhead of copying objects and maintaining extra lists required for a merge-sort consumes more time than is saved.

Compromise solutions do exist. For example, given that the number of overlapping objects in any scene is likely to be much less than the total number of objects in the world, we could provide for only a fixed number of different drawing depths (perhaps a few thousand). Objects with the same depth would be rendered in an arbitrary order—for objects that don't overlap this is not a problem. Objects with

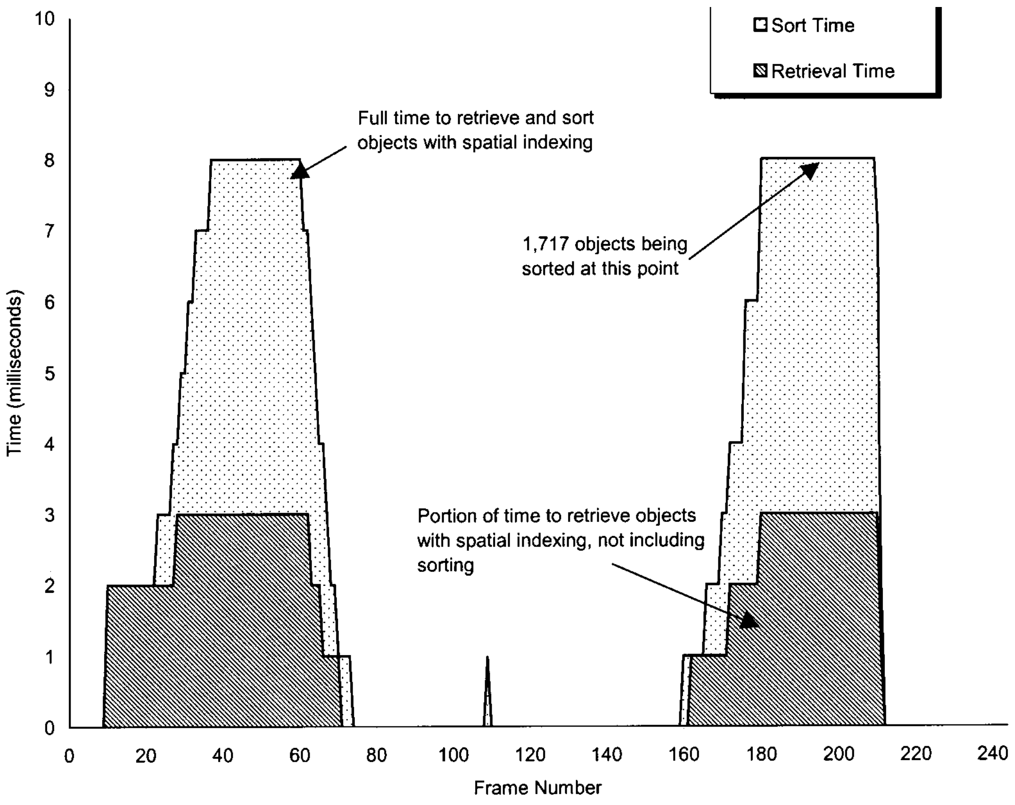


Figure 4. Time taken to retrieve and sort visible objects in a scene depicting a map of New York City containing 2135 objects

different depths are rendered in order according to their depth, with lower depth numbers being rendered first. This lets users specify overlapping constraints when they are important.

Another possibility is to use Binary Space Partition (BSP) trees[28] with the drawing order of each object representing its Z depth. This approach has the advantage of eliminating the sort entirely as BSP trees implicitly sort objects. However, maintaining BSP trees for dynamic scenes tends to be compute intensive.

We are currently experimenting with an indexed approach, which eliminates the need for sorting and is still reasonably efficient. We construct an array of flags, one flag per object. The flags are maintained in pages, each page containing 256 flags, as well as a *used* bit that is set whenever one of the flags in the page is set. The retrieval algorithm clears all the used bits, and then invokes the R-Tree spatial index described earlier to locate visible objects. For each visible object, the corresponding flag in the array is set. Whenever a page of flags is referenced for the first time, the page is marked as used and the flags in the page are cleared. After the spatial index query is complete, the renderer examines each of the pages. If a page is marked as used, then all the flags in the page are checked, and each object whose flag is set is rendered.

With this paging approach, if every object on a surface is visible the algorithm has a worst case running time of $O(N)$, where N is the number of objects on the surface. This is much better than the QuickSort approach, which is $O(N \log N)$. If no objects are visible, the paging approach still has a performance overhead of $O(N)$, since every used bit must be checked. But the costs involved are very low, since there is only a single used bit to check for each group of 256 objects. By paging the used bits themselves, the overheads can be reduced still further, leading to good overall performance. The timing tests in the next section do not reflect this new approach which is still under development.

A hardware solution to this problem also exists: using a depth buffer eliminates the need for sorting. Depth buffers are commonly found on many 3D cards, and also on our hypothetical ZGA card. To render the ZUI scene, the depth buffer is first cleared, then objects are drawn in arbitrary order, but with a Z-depth corresponding to their drawing order number. The graphics hardware then takes care of clipping objects according to their depth. For this to work well, the depth buffer must have at least 16 bits per pixel to offer a reasonable number of depths.

Timing results

In this section, we present the results of several timing tests of Pad++, showing the performance of rendering and visible object determination. Remember that the goal is frame times of at most 100 milliseconds (ms), or 10 frames per second (fps). All tests were performed on a 200 MHz Pentium Pro running Linux, with an 800×600 pixel graphics window. Window size is significant, because each frame render involves both clearing the back buffer to the background color, and copying the back buffer to the window. The overhead for this clearing and copying on the reference platform is 4 ms.

Four different tests were performed, to compare hand-created scenes with computer-generated scenes, and to compare scenes of uniformly sized objects with scenes of objects of many different sizes. See the graphs in Figures 5–8.

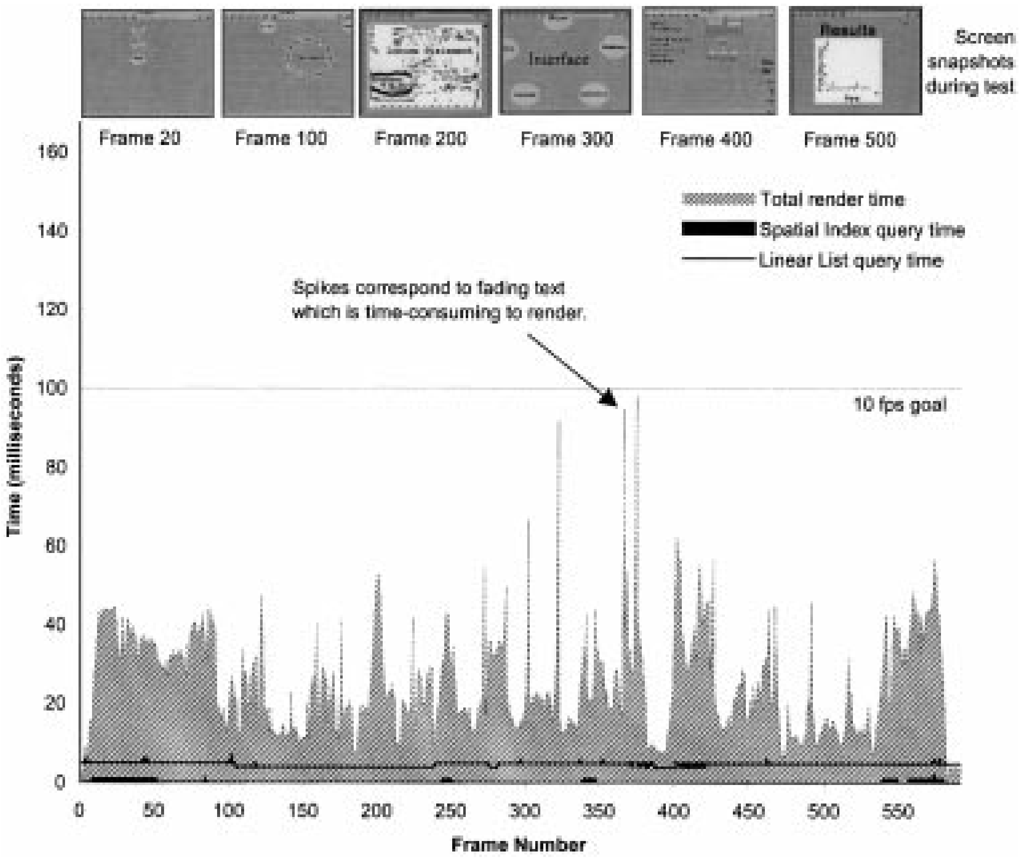


Figure 5. Times for 20 copies of a hand-created scene of a typical Pad++ document. Objects have very different sizes, and only a small number are visible at any given time. Total of 11,620 objects

Each graph shows the total rendering time for each frame as the view is changed to move through all parts of the scene, zooming in and out, and panning to each area of the scene. The total rendering time is further broken into two parts: time spent querying the spatial index (and sorting the results), and time spent actually doing the drawing.

Each graph also shows the time it takes to perform a simple linear visible object check (i.e. checking every object for visibility every frame). This lets us compare the effectiveness of the spatial index with a simple linear approach.

Intuitively, the time taken by the linear approach should be constant, since every object is checked in every frame. In practice, for visible objects the system also performs small-object culling, so the total time taken in each frame varies slightly according to the number of visible objects (i.e. visible objects take two checks, whereas objects that are not in the view require only one).

To evaluate the cost of maintaining the spatial index, we performed a test where we moved many objects at once. For the scenes used by Figures 7 and 8, we moved all the objects, with and without spatial indexing. The timing results are shown in Table I.

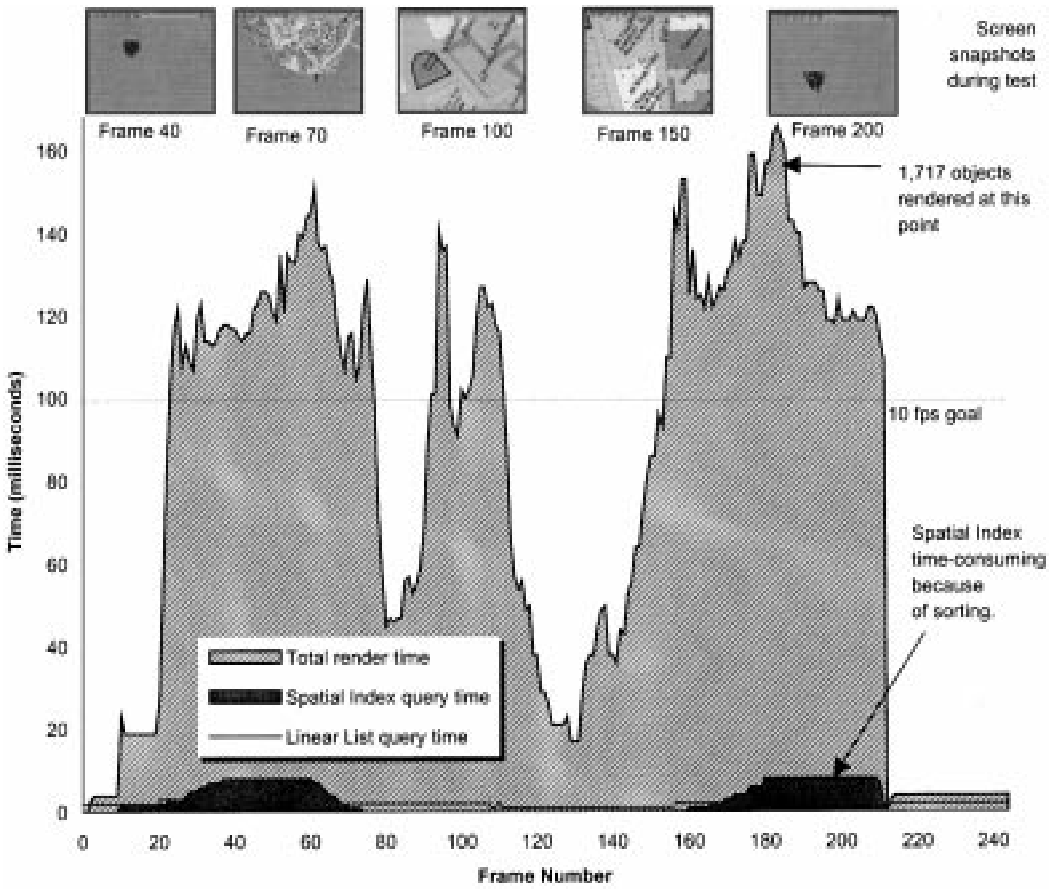


Figure 6. Times for hand-created scene of 2135 objects where all objects are of similar sizes. The scene depicts a map of lower Manhattan

We also tested the time it takes to build the index in the first place. For the first scene, building the index took 1.75 seconds, so about 12,485 objects can be inserted a second. The second scene, inserting objects into the index took 1.82 seconds, or 12,363 objects per second.

Analysis

These results collectively show that we are very close to meeting our goals for typical scenes of a ZUI. Many scenes met our frame rate goal of 10 fps, and the spatial index is effective. For most scenes, the spatial index is no slower than a simple linked list, and for scenes with many objects where only a small percentage of them are visible, the spatial index cuts down on visible object determination time substantially. For all scenes, the time spent detecting visible objects is a small fraction of the time spent actually drawing.

Regarding maintenance efficiency, Table I shows that maintaining the spatial index does impact object movement speed, but by a factor which is less than two. With

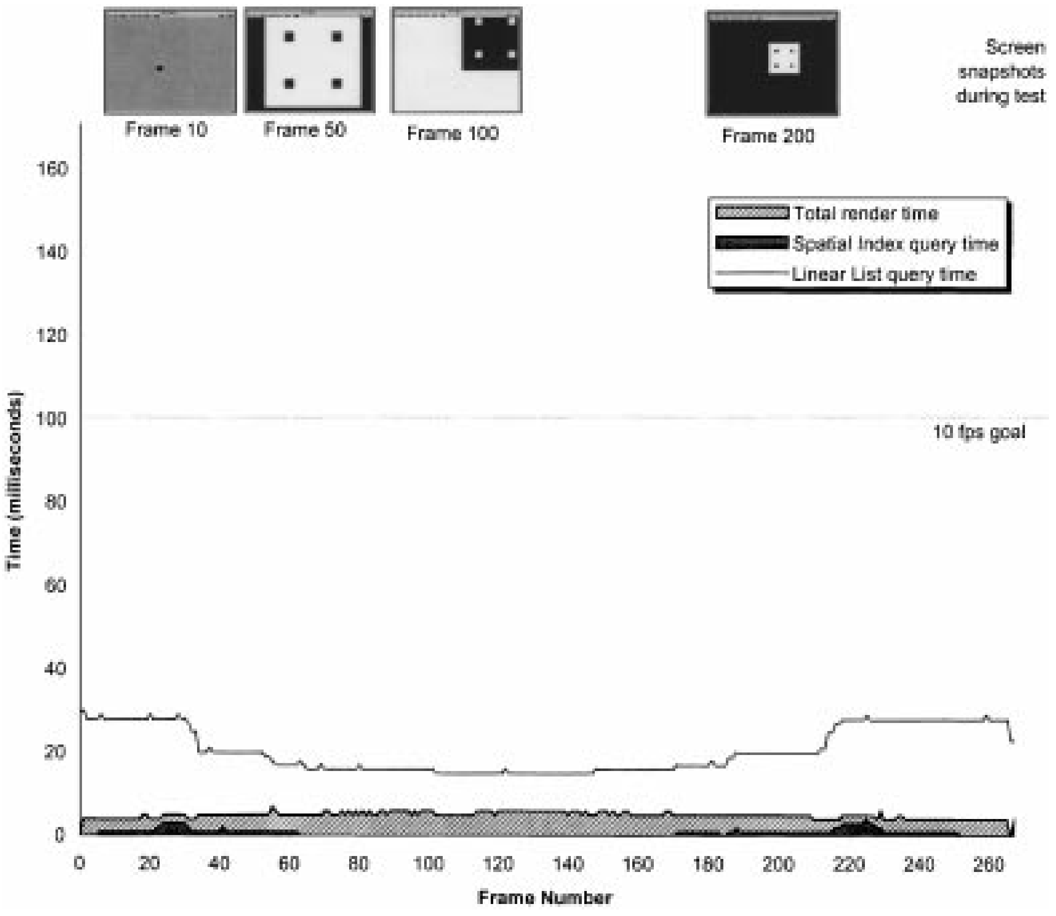


Figure 7. Times for a computer-generated scene of 21,849 objects with significant size differences between objects. The scene contains seven levels of nested rectangle where each rectangle contains four others

spatial indexing, we can move an average of 461 objects per second. With no indexing, we can move 633 objects per second (we suspect that rewriting the test in C instead of Tcl would increase both of these numbers substantially).

The one type of scene for which our implementation does not meet our rendering goals is where there are many objects of similar size. When zoomed out to a certain point, all of these objects become visible and the spatial index spends a lot of time sorting these objects, and the renderer spends even more time rendering all of these objects. Thus, our current implementation of Pad++ is effective for scenes that contain many objects at different sizes, but is not effective for scenes with many objects at similar sizes.

PART III: PAD++ STRUCTURE

Pad++ is an object-oriented graphical interface library which implements a Zooming User Interface (ZUI) as previously described. It is object-oriented in both its

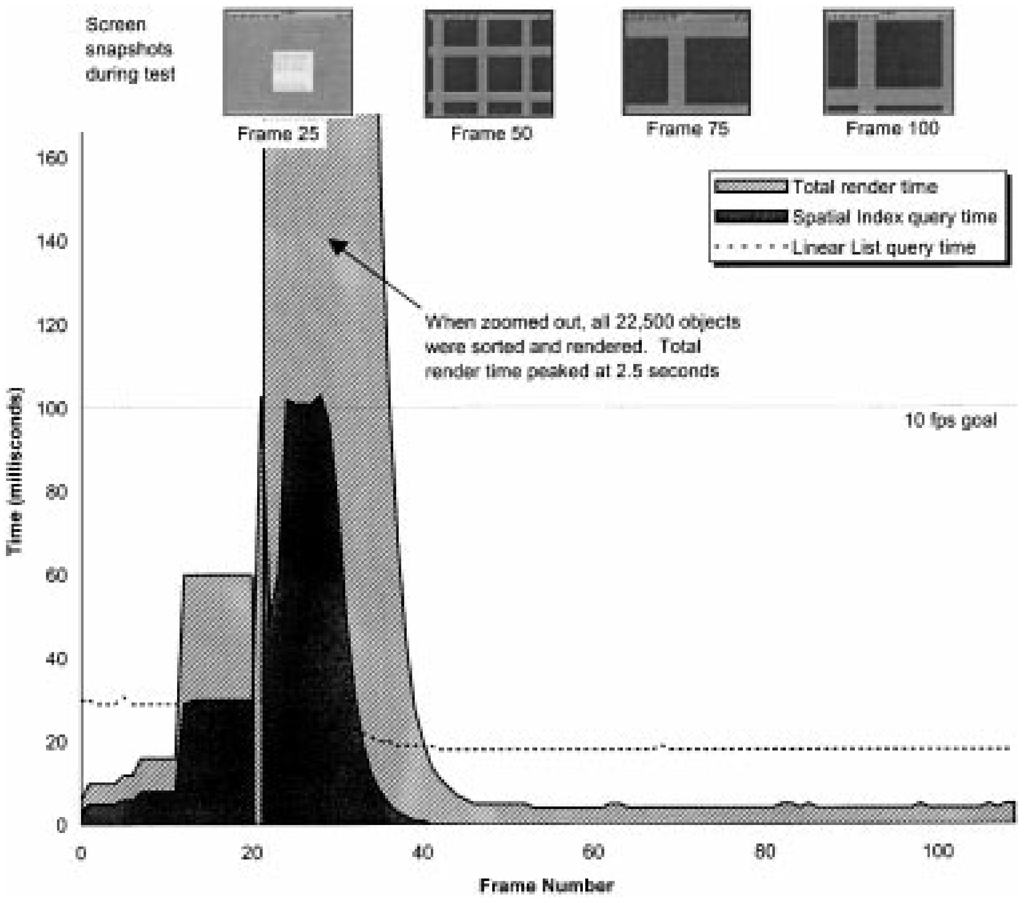


Figure 8. Times for a computer-generated scene of 22,500 objects with no size differences between objects. The scene contains an array of 150 × 150 identical rectangles. When zoomed out, all 22,500 rectangles are visible, and thus rendering times grow very high

Table I. Time taken to move many objects, comparing linear list approach with spatial index approach

	Number of Objects	Linear list Time to move every object (seconds)	Spatial index Time to move every object (seconds)	Cost ratio Spatial Index: Linear List
Scene from Figure 7	21,849	32	53	1.66
Scene from Figure 8	22,500	38	43	1.13

implementation and its use. All graphical elements in Pad++ are instances of objects and share certain functionality, such as the ability to be moved and resized.

This section describes the structure of Pad++, and how its different components are designed and implemented. We discuss the object hierarchy, and the implementation of unusual features.

Structure

Pad++ is implemented in C++, and provides application interfaces for several languages, including C++, Tcl, Scheme, Perl, and KPL (an in-house reverse-polish language for rendering). In addition, an interface to Java is under development. Most applications to date have been written in Tcl as that is the language we originally targeted, though we have also developed a procedural animation engine that runs within Pad++ and uses KPL.[31] While Pad++ now runs on Windows 95/NT, we originally designed and implemented Pad++ for UNIX with the X window system. It runs on many versions of UNIX, including Linux, SunOS, Solaris, IRIX, and FreeBSD. The Windows 95/NT version of Pad++ maps X windowing system calls to equivalent functions under Windows. Since in some cases this mapping is not straightforward, the Windows version is currently not as fast as the UNIX version. We are working on a renderer that uses the Windows features more directly.

One continuing goal has been to make Pad++ as portable as possible. Because people download, compile, and run Pad++ on all kinds of systems all around the world, we decided to use only the most commonly available and most reliable features of C++. For this reason, we decided not to use multiple inheritance or templates. In some places, this made the code more complicated, but we feel it was a necessary trade-off in order to minimize the time we spend supporting Pad++.

Pad++ consists of several components. At a high level, they are:

(a) *Renderer:* the renderer performs all the rendering to the screen. It maintains a stack of transformations that specify translation and scale.
(b) *Event Handler:* the event handler is responsible for processing all input events. It determines which objects receive events, it maps events through portals and it takes care of event grabbing (insuring that that mouse motion and release events go to the same object that received the associated press event.)
(c) *Surfaces:* a surface represents a single flat data space where graphical objects exist. Objects can exist at any position and scale on the surface.
(d) *Views:* surfaces are mapped to the screen through views. A view specifies the position and magnification at which the associated surface is seen. There are currently two places where views are used: for windows and for portals. The extent of the surface that is seen is specified indirectly by the size of the associated window or portal.
(e) *Objects:* every graphical item on a Pad++ surface is derived from a base Object class. This class defines much of the behavior common to all objects. It controls where and at what size an object appears, the object's transparency, and the range of sizes at which it is visible. It also controls the object's stickiness and drawing order, as well as what layer it is on. Figure 9 shows the hierarchy of all objects deriving from this base Object class.

This class hierarchy is somewhat complicated, partly because Pad++ is designed to

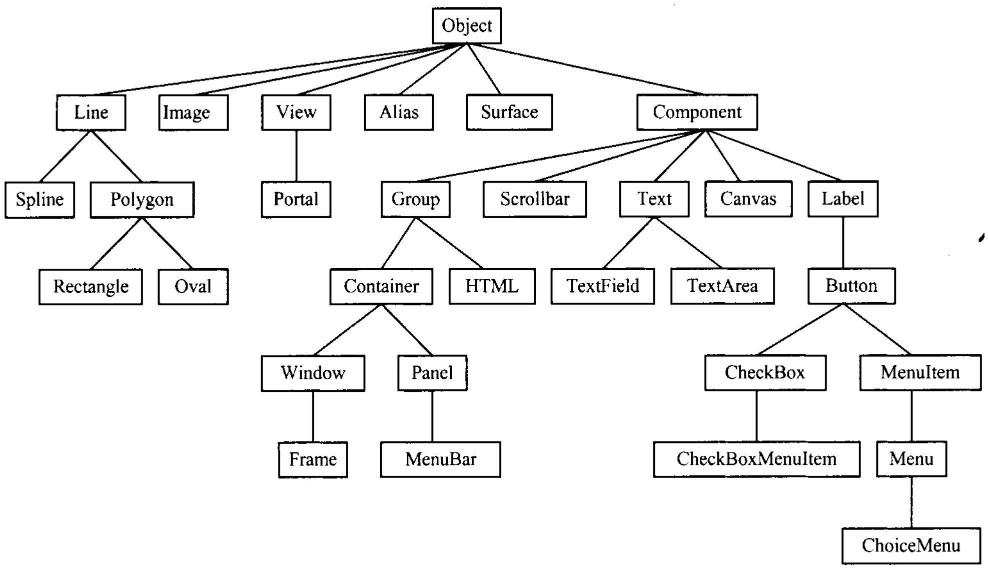


Figure 9. Pad++ object hierarchy

be connected to both Tcl and to Java. We made a strong effort to be consistent with the programming and visual interface standards of both languages.

Surfaces, views, and portals

The Pad++ dataspace is structured around surfaces and views. A surface is a plane on which objects exist. The Pad++ environment supports multiple surfaces simultaneously. Each surface can be visible within top-level windows, or can be seen within portals. Objects exist on a surface within the surface's coordinate system. The Pad++ coordinate system uses a standard right-handed Cartesian coordinate system. The X axis increases to the right and the Y axis increases *up*. Note that this is different from many graphical windowing systems. All coordinates are specified in floating point units that by default correspond to the dimensions of a single pixel on the screen. This means that, when the top-level view is at a scale of 2.0, a line drawn on the surface from 0.0 to 100.0 is 200 pixels long. Dimensions can also be specified in inches, millimeters, or points.

Surfaces are mapped to the screen through views. Each surface that is mapped to a top-level window implicitly gets a view that controls what part of the surface is visible in that window. Views specify the visible portion of a surface with a point and a magnification. The point specifies the portion of the surface that will appear at the center of the view. The magnification specifies how much the surface should be enlarged or shrunk.

Portals are a special type of view. They exist on a Pad++ surface, and can be moved and resized like other objects on the surface, but they also specify a 'lookon'—this specifies the surface that is visible through the portal. Portals can look onto the surface they themselves are on, or they can look onto any other Pad++ surface.

Portals are implemented with special rendering and event processing methods. A portal's render method sets the clipping region, and then renders the Portal's lookon surface, clipped by this region. A portal can render another portal within itself, but prevents recursive rendering of itself. Portals also process events specially. When an event hits a portal, if the event is on the portal's outer frame, the event is sent to the portal object itself. If the event is within the portal, the event is passed through to the surface the portal looks onto. When an object receives an event, it may query the event to determine the list of portals (if any) that it was passed through.

Sample Pad++ code

Here is a short example of some Tcl code that gives a feel for how some of the simple features of Pad++ can be used. This example creates a surface, puts it within a top-level window, and then creates several objects with event bindings, and changes the view. In this example, '>' represents a UNIX prompt, and '%' represents a Tcl prompt. Commands typed in by the user appear in bold face. The resulting output is shown in Figure 10.

```
padwish               ;# Start the Pad++ executable
% pad .pad            ;# Make a Pad++ surface
.pad% pack .pad       ;# Map it to a top-level window with a view
```

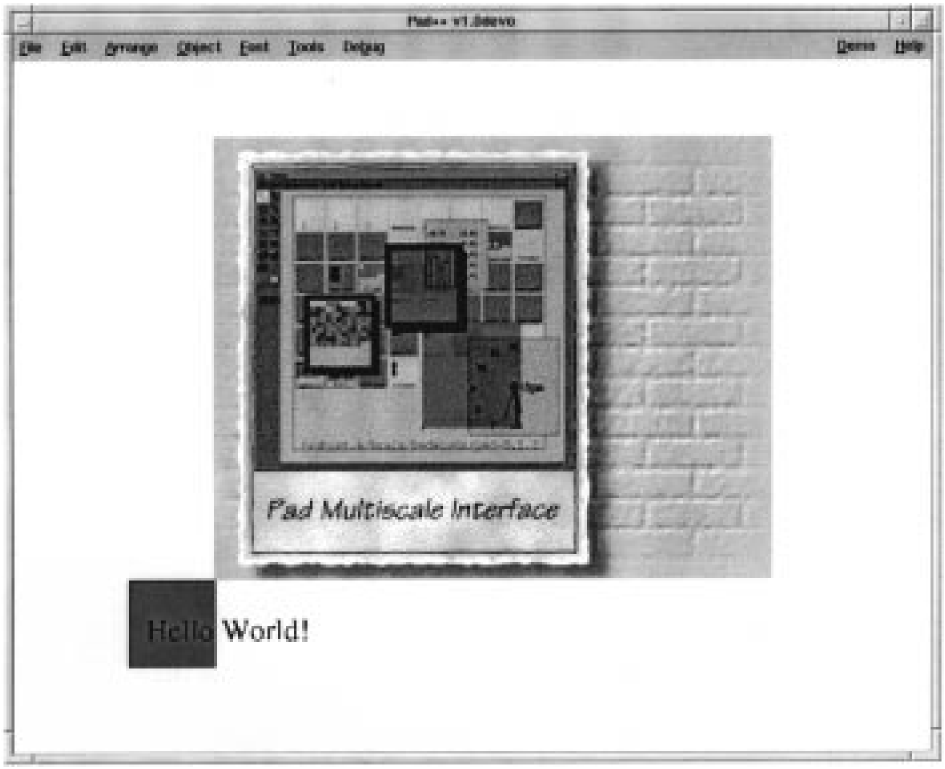


Figure 10. Snapshot of the Pad++ window after executing the Sample Pad++ code

```
                    ;# Create a rectangle. A unique integer is
                    ;# generated which is used to identify this
                    ;# rectangle in the future.
% .pad create rectangle 0 0 50 50 -fill red
3
                    ;# System returns ‘3’, uniquely ident-
                    ifying object

                    ;# Create some text and place it within
                    ;# rectangle
% .pad create text -text “Hello World!” -font “Times-12” \
-anchor sw -position “10 10 1”
4
                    ;# System returns ‘4’, uniquely
                    ;# identifying object

                    ;# Create an image from a file whose
                    ;# bottom-left
                    ;# corner is at the same place as the
                    ;# top-right corner of the rectangle.
% .pad create image -image “pad.gif” -anchor sw -position \
“50 50 1”
5                   ;# System returns ‘5’, uniquely
                    ;# identifying object

                    ;# Create event bindings so that moving the
                    ;# mouse over the rectangle highlights it,
                    ;# and clicking on the rectangle moves it
                    ;# 10 pixels to the right.
% .pad bind 3 ⟨Enter⟩ {%P itemconfig %O -pen blue}
% .pad bind 3 ⟨Leave⟩ {%P itemconfig %O -pen black}
% .pad bind 3 ⟨ButtonPress-1⟩ {%P slide %O 10 0}
                    ;# Zoom the view so that the center
                    ;# of the image is at the center of the
                    ;# view, and is 1.5 times larger than
                    ;# normal.
                    ;# The view change will be animated over a
                    ;# period of 1000 milliseconds.
% .pad moveto 200 150 1.5 1000
```

Quantity of zooming space

Just how big is the zooming space of Pad++? Ideally, a ZUI would have unlimited space so objects could be put at any position and at any scale. While this might seem unnecessary, applications can easily use a lot of space. For example, a visualization system that depicted hierarchies through containment uses a lot of depth. If each level of the hierarchy were an order of magnitude smaller than the parent, dozens of orders of magnitude of zooming could quickly be used up. In addition to depth, the amount of breadth is also a concern. Just how far apart can objects be placed? Again, this may not seem crucial, but as soon as you zoom out, even objects that are very far apart are brought closer together visually. A user can very easily zoom out, pan a little, and then zoom in with the result that they've covered a very wide expanse of space. (*Space-scale diagrams* provide an analytical tool that is useful for describing and analyzing these kinds of spaces.[32])

In Pad++, the view and object coordinates are stored with standard 32 bit floats that store roughly seven orders of magnitude of resolution (+/−). So, we expect 14 orders of magnitude for zooming and panning. Again, this may seem like a lot, but to go from a dot to full screen is roughly three orders of magnitude (on a 1,000 pixel screen). Pad++ can do that about five times.

Care must be taken when implementing coordinate transformations in a ZUI. Unfortunately, the most simple and intuitive approach to implementing transformations can result in reduced zooming space. To understand this, note that each view has an offset and a magnification, which are stored as a triplet (**`xview, yview, zoom`**). Every object has a position and a scale (**`xoffset, yoffset, scale`**). In addition, objects such as polygons and lines also have coordinates ($\mathbf{x_1}$**`,`** $\mathbf{y_1}$**`,`** …). To apply a coordinate transformation from object to screen coordinates, we start by applying the transform for the current view (in these examples, we show only the **`x`** coordinate):

```
-(xview * zoom)
```

Now, we apply the object's transformation, leading to the expression:

```
(xoffset * zoom)–(xview * zoom)
```

However, consider when you zoom in and pan off to the side so that **`xview`** and **`zoom`** both become large, the quantity (**`xview * zoom`**) can overflow quickly. This results in a reduced zooming space—that is, the more you zoom in, the less you can pan (the effective space is pyramid shaped).

A better approach is to instead combine the two terms and compute transformations using:

```
zoom * (xoffset–xview)
```

Since the offset of the object counters the offset of the view for visible objects, the effects of overflow are reduced, and the result is a larger usable zooming space.

In implementation terms, computing transformations in this manner complicates matters. In the earlier equation, view and object transformations are carried out separately—an approach that lends itself to using a transform stack. Stacks are an elegant mechanism for graphical systems, which often have hierarchical structures (such as nested hierarchical groups) that are easily handled using recursion and stacks.

To implement this in Pad++, we keep a separate stack of view transformations and object transformations. This lets us combine the two terms separately as we compute coordinates.

An alternative solution to this problem is to use larger precision floating point numbers within the coordinate transformation system (e.g. store the coordinates using floats and perform transformations using doubles). One problem with this approach is that higher precision arithmetic comes at a performance cost—the result is increased space but slower rendering. Also, if the coordinates are already stored in the highest precision format supported by hardware, this approach may not be feasible.

CONCLUSION

Pad++ is an implementation of a Zooming User Interface (ZUI). We achieved our goals of creating a substrate that supports rich zooming graphics with a large number of objects, and a consistent high frame rate for typical scenes.

ZUIs are strongly related to real time 3D graphical systems and 2D windowing systems. While we have reused existing real-time graphics techniques as much as possible, ZUIs present new and interesting challenges which distinguish them from their 2D and 3D cousins. Consequently, Pad++ represents a unique combination of features and implementation techniques.

There are still several areas where further research is required. Probably the most serious problem with Pad++ is its inability to maintain a high frame rate when many objects are visible simultaneously. Part of the difficulty here is that we have not solved the problem of visually presenting objects at many different levels of detail. While we do use level of detail to speed up rendering, object presentations do not change rapidly or extensively enough. Instead, the system gets slow and animations are poor when too many objects are visible in a scene. In 3D systems, rendering lower-resolution models generally solves this problem. But this is hard to do in a reasonable fashion for text and images, which form the bulk of the visual content in a 2D system.

Finally, for Pad++ to scale up, we need to be able to handle scenes with a much larger number of objects. Currently, Pad++ holds all objects in RAM, and so the maximum number of objects is limited more by memory than anything else. Perhaps the best way to scale up the number of objects would be to link Pad++ to a persistent database, and keep only a small cache of objects in memory.

ACKNOWLEDGEMENTS

While Pad++ was primarily implemented by the authors of this paper, many other people have contributed to both the code and to our understanding of ZUIs. Our primary collaborator at the University of New Mexico is Professor Jim Hollan. Our primary collaborator at New York University is Professor Ken Perlin, the originator of Pad. Other students and staff at UNM that we have worked with include Hugh Bivens, Allison Druin, George Hartogensis, Ron Hightower, Mohamad Ijadi, David Proft, Laura Ring, David Rogers, Jason Stewart, David Thompson, David Vick, and Ying Zhao. People we have worked with at New York University include David Bacon, Troy Downing, Athomas Goldberg, and David Fox. Finally, people that have been involved with Pad++ in other places include Mark Rosenstein, Larry Stead, and Kent Wittenburg at Bellcore, and George Furnas at the University of Michigan. It is interesting to note that six out of 24 people that worked on this project are named David!

In regards to the Pad++ implementation, we particularly appreciate the help of David Fox who demonstrated how images could be scaled so quickly under X windows. We also appreciate Paul Haeberli from SGI who donated code for polygonizing Adobe Type 1 fonts.

This project has been primarily supported by generous funding from DARPA contract #N66001–94-C-6039 to which we are grateful.

REFERENCES

1. B. B. Bederson, J. D. Hollan, K. Perlin, J. Meyer, D. Bacon and G. Furnas, 'Pad++: A zoomable graphical sketchpad for exploring alternate interface physics', *Journal of Visual Languages and Computing,* **7**, 3–31 (1996).
2. J. Meyer, 'EtchaPad—disposable sketch based interfaces', *Proceedings Companion of Human Factors in Computing Systems (CHI'96),* ACM, New York, 1996, pp. 195–196.
3. A. Druin, J. Stewart, D. Proft, B. Bederson and J. D. Hollan, 'KidPad: A design collaboration between children, technologists, and educators', *Proceedings of Human Factors in Computing Systems (CHI'97),* ACM, New York, 1997, pp. 463–470.
4. I. Sutherland, 'Sketchpad: A man-machine graphical communication system', *Tech Report #296*, MIT Lincoln Labs, Cambridge, MA, 1963.
5. W. C. Donelson, 'Spatial management of information', *Proceedings of Computer Graphics (SIGGRAPH '78),* ACM, New York, 1978, pp. 203–209.
6. K. Perlin and D. Fox. 'PAD: An alternative approach to the computer interface', *Proceedings of Computer Graphics (SIGGRAPH '93),* ACM, New York, 1993, pp. 57–64.
7. B. B. Bederson and J. D. Hollan, 'Pad++: A zooming graphical interface for exploring alternate interface physics', *Proceedings of User Interface Software and Technology (UIST '94)*, ACM, New York, 1994, pp. 17–26.
8. B. B. Bederson, J. D. Hollan, J. Stewart, D. Rogers, A. Druin, D. Vick, L. Ring, E. Grose and C. Forsythe, 'A zooming web browser', in C. Forsythe, J. Ratner and E. Grose (eds.), *Human Factors and Web Development*, Lawrence Earlbaum, 1997.
9. S. K. Card, G. G. Robertson and J. D. Mackinlay, 'The Information Visualizer, an Information Workspace', *Proceedings of Human Factors in Computing Systems (CHI'91),* ACM, New York, 1991, pp. 181–188.
10. Perspecta, Inc., http://www.perspecta.com, 1997.
11. Merzcom, Inc., http://www.merzcom.com, 1997.
12. D. Fox, 'Tab: The Tabula Rasa zooming user interface system', URL http://www/cat/nyu.edu/fox/tab.html, New York University, New York, 1997.
13. J. H. Maloney and R. B. Smith, 'Directness and liveness in the morphic user interface construction environment', *Proceedings of User Interface Software and Technology (UIST '95)*, ACM, New York, 1995, pp. 21–28.
14. R. B. Smith, 'Experiences with the alternate reality kit: An example of the tension between literalism and magic', *Proceedings of the Human Factors in Computing Systems and Graphics Interface Conference*, ACM, New York, 1987, pp. 61–67.
15. D. A. Henderson, Jr. and S. K. Card, 'Rooms: The use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface', *IEEE Transactions on Graphics,* **5**(3), 211–243 (1987).
16. FVWM, http://www.hpc.uh.edu/fvwm, 1997.
17. Architecture Review Board (ARB), *OpenGL Reference Manual*, Addison-Wesley, 1992.
18. P. Haeberli, and M. Segal, 'Texture mapping as a fundamental drawing primitive', in M. Cohen (ed.), *Fourth Eurographics Workshop on Rendering* (URL http:/www.sgi.com/grafica/texmap/index.html), Paris, France, June 1993.
19. J. D. Foley, A. van Dam, S. K. Feiner and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison-Wesley, 1990, pp. 992–996.
20. R. J. Deyo, A. Briggs and P. Doenges, 'Getting graphics in gear: Graphics and dynamics in driving simulation', *Proceedings of Computer Graphics (SIGGRAPH '88),* **24**(4), 317–326 (July 1988).
21. J. M. Airey, J. H. Rohlf and F. P. Brooks, Jr., 'Towards image realism with interactive update rates in complex virtual building environments', *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics,* **24**(2), 41–50 (1990).
22. E. Angel, *Interactive Computer Grapics*, Addison-Wesley, 1996.
23. T. Funkhouser, S. Teller, C. Sequin and D. Khorramabadi, 'The UC Berkeley system for interactive

visualization of large architectural models', *Presence: Teleoperators and Virtual Environments,* **5** (1) (Winter 1996).
24. H. Samet, *The Design and Analysis of Spatial Data Structures,* Addison-Wesley, 1990.
25. H. Samet, *Applications of Spatial Data Structures: Compter Graphics, Image Processing, and GIS,* Addison-Wesley, 1990.
26. A. Guttman, 'R-trees: A dynamic index structure for spatial searching', *Proceedings of the SIGMOD Conference*, Boson, MA, June 1984, pp. 47–57.
27. G. Kedem, 'The quad-CIF tree: A data structure for hierarchical on-line algorithms', *Proceedings of the Nineteenth Design Automation Conference,* Las Vegas, NV, June 1982, pp. 352–357.
28. H. Fuchs, Z. M. Kedem and B. F. Naylor, 'On visible surface generation by a priori tree structures'. *Computer Graphics,* **14**(3), 124–133 (July 1980).
29. J. L. Bentley, 'Multidimensional binary search trees used for associative searching', *Communications of the ACM* (18), 509–517 (September 1975).
30. N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, 'The R*-tree: An efficient and robust access method for points and rectangles', *Proceedings of the SIGMOD Conference*, Atlantic City, NJ, June 1990, pp. 322–331.
31. K. Perlin, 'Layered compositing of facial expression', *Visual Proceedings of Computer Graphics (SIGGRAPH '97),* ACM, New York, 1993, pp. 226–227.
32. G. W. Furnas, B. B. Bederson, 'Space-scale diagrams: Understanding multiscale interfaces', *Proceedings of Human Factors in Computing Systems (CHI '95),* ACM, New York, 1995, pp. 234–241.

Pad++ Bederson - 5

Journal of Visual Languages and Computing (1996) **7**, 3–31

Pad++: A Zoomable Graphical Sketchpad For Exploring Alternate Interface Physics

Benjamin B. Bederson,[*] James D. Hollan,[*] Ken Perlin,[†] Jonathan Meyer,[†] David Bacon[†] and George Furnas[‡]

[*]Computer Science Department, University of New Mexico, Albuquerque, NM 87131, U.S.A., [†]Media Research Laboratory, Computer Science Department, New York University, NY 10003, U.S.A., [‡]Bell Communications Research, 445 South Street, Morristown, NJ 07960, U.S.A.

Received 7 March 1995 and accepted 29 September 1995

We describe Pad++, a zoomable graphical sketchpad that we are exploring as an alternative to traditional window and icon-based interfaces. We discuss the motivation for Pad++, describe the implementation and present prototype applications. In addition, we introduce an informational physics strategy for interface design and briefly contrast it with current design strategies. We envision a rich world of dynamic persistent informational entities that operate according to multiple physics specifically designed to provide cognitively facile access and serve as the basis for the design of new computationally-based work materials.

1. Introduction

Imagine a computer screen made of a sheet of a miraculous new material that is stretchable like rubber but continues to display a crisp computer image, no matter what the sheet's size. Imagine that this sheet is very elastic and can stretch orders of magnitude more than rubber. Further, imagine that vast quantities of information are represented on the sheet, organized at different places and sizes. Everything you do on the computer is on this sheet. To access a piece of information you just stretch to the right part and there it is.

Imagine further that special lenses come with this sheet that let you look onto one part of the sheet while you have stretched another part. With these lenses, you can see and interact with many different pieces of data at the same time that would ordinarily be quite far apart. In addition, these lenses can filter the data in any way you would like, showing different representations of the same underlying data. The lenses can even filter out some of the data so that only relevant portions of the data appear.

Imagine also new stretching mechanisms that provide alternatives to scaling objects purely geometrically. For example, instead of representing a page of text so small that it is unreadable, it might make more sense to present an abstraction of the text, perhaps so that just a title that is readable. Similarly, when stretching out a spreadsheet,

[*] *(bederson, hollan)@ cs.umn.edu,* [†] *(perlin, meyer bacon)@ play.cs.nyu.edu,* [‡]*gwf@bellcore.com URL:http://www.cs.unm.edu/pad++*

1045-926X/96/010003 + 29 $18.00/0

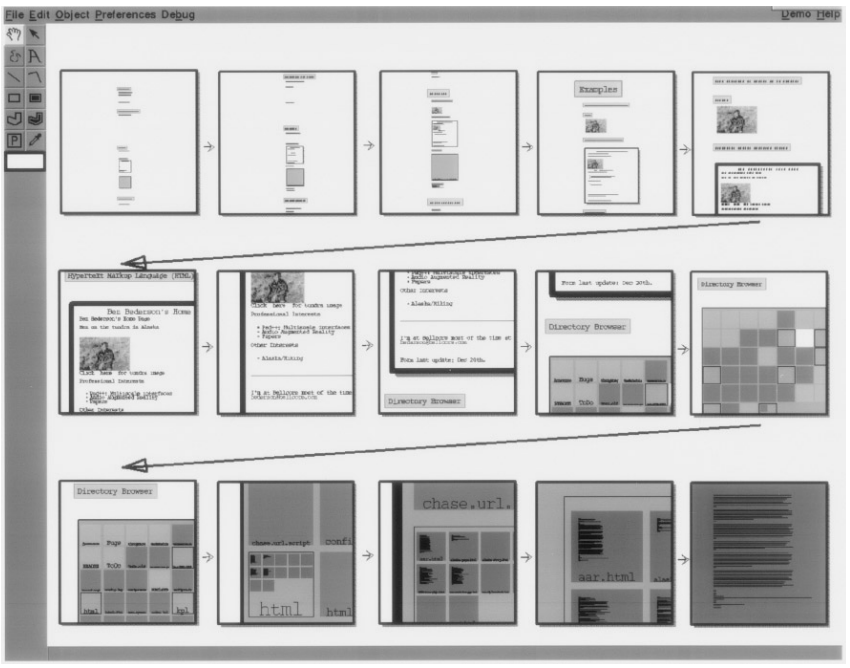


Figure 1. A sequence of views as we zoom into some data

instead of showing huge numbers it might make more sense to show the computations from which the numbers were derived or a history of interaction with them.

The beginnings of an interface like this sheet exists today in a program we call Pad++. We don't really stretch a huge rubber-like sheet, but we simulate it by *zooming* into the data. We use what we call *portals* to simulate lenses, and a notion we call *semantic zooming* to scale data in non-geometric ways. The user controls where they look on this vast data surface by panning and zooming. Portals are objects on the Pad++ data surface that can see anywhere on the surface, as well as filter data to represent it differently than it normally appears.

Panning and zooming allow navigation through a large information space via direct manipulation. By tapping into people's natural spatial abilities, we hope to increase users' intuitive access to information. Conventional computer search techniques are also provided in Pad++, bridging traditional and new interface metaphors. Figure 1 depicts a sequence of views as we pan and zoom into some data.

1.1. Motivation

If interface designers are to move beyond windows, icons, menus and pointers to explore a larger space of interface possibilities, additional ways of thinking about interfaces that go beyond the desktop metaphor are required.

There are myriad benefits associated with metaphor-based approaches, but they also orient designers to employ computation primarily to mimic mechanisms of older media. While there are important cognitive, cultural and engineering reasons to exploit earlier successful representations, this approach has the potential of underutilizing the mechanisms of new media.

For the last few years we have been exploring a different strategy [21] for interface design to help focus on novel mechanisms enabled by computation rather than on mimicking mechanisms of older media. Informally, the strategy consists of viewing interface design as the development of a physics of appearance and behavior for collections of informational objects.

For example, an effective informational physics might arrange for an object's representation to be a natural by-product of normal activity. This is similar to the physics of certain materials that evidence the wear associated with use. Such wear records a history of use and at times this can influence future use in positive ways. Used books crack open at frequently referenced places. It is common for recently consulted papers to be at the tops of piles on our desks. Usage dog-ears the corners and stains the surface of index cards and catalogs. All these wear marks provide representational cues as a natural product of doing, but the physics of materials limit what can be recorded and the ways it can influence future use.

Following an informational physics strategy has led us to explore history-enriched digital objects [18, 19]. Recording on objects (e.g. reports, forms, source-code, manual pages, email, spreadsheets) the interaction events that comprise their use makes it possible on future occasions, when the objects are used again, to display graphical abstractions of the accrued histories as parts of the objects themselves. For example, we depict the copy history on source code. This allows a developer to see that a particular section of code has been copied and perhaps be led to correct a bug not only in the piece of code being viewed but also in the code from which it was derived.

This informational physics strategy has also lead us to explore new physics for interacting with graphical data. As part of that exploration we have formed a research consortium to design a successor to Pad [25]. This new system, Pad++, serves as a substrate for exploration of novel interfaces for information visualization and browsing in complex, information-intensive domains. The system is being designed to operate on platforms ranging from high-end graphics workstations to PDAs (Personal Digital Assistants) and interactive set-top cable boxes. Here we describe the motivation behind the Pad++ development, report the status of the current implementation and present initial prototype applications.

Today, there is much more information available than we can access readily and effectively. The situation is further complicated by the fact that we are on the threshold of a vast increase in the availability of information because of new network and computational technologies. Paradoxically, while we continuously process massive amounts of perceptual data as we experience the world, we have perceptual access to very little of the information that resides within our computing systems or that is reachable via network connections. In addition, this information, unlike the world around is, is rarely presented in ways that reflect either its rich structure or dynamic character.

We envision a much richer world of dynamic persistent informational entities that operate according to multiple physics specifically designed to provide cognitively facile access. These physics need to be designed to exploit semantic relationships explicit and implicit in information-intensive tasks and in our interaction with these new kinds of computationally-based work materials.

One physics central to Pad++ supports viewing information at multiple scales and attempts to tap into our natural spatial ways of thinking. We address the information presentation problem of how to provide effective access to a large structure of information on a much smaller display. Furnas [15] explored degree of interest functions to determine the information visible at various distances from a central focal area. There is much to recommend the general approach of providing a central focus area of detail surrounded by a periphery that places the detail in a larger context.

With Pad++ we have moved beyond the simple binary choice of presenting or eliding particular information. We can also determine the scale of the information and, perhaps most importantly, the details of how it is rendered can be based on various semantic and task considerations that we describe below. This provides semantic task-based filtering of information that is similar to the early work at MCC on lens-based filtering of a knowledge base using HITS [20] and the recent work of moveable filters at Xerox [4] [30].

The ability to make it easier and more intuitive to find specific information in large dataspaces is one of the central motivations behind Pad++. The traditional approach is to filter or recommend a subset of the data, hopefully producing a small enough dataset for the user to navigate effectively. Pad++ is complementary to these filtering approaches in that it promises to provide a useful substrate to *structure* information.

2. Description

Pad++ is a general-purpose substrate for creating and interacting with structured information based on a zoomable interface. It adds scale as a first class parameter to all items, as well as various mechanisms for navigating through a multiscale space. It has several efficiency mechanisms which help maintain interactive frame-rates with large and complicated graphical scenes.

While Pad++ is not an application itself, it directly supports creation and manipulation of multiscale graphical objects, and navigation through spaces of these objects. It is implemented as a widget in Tcl/Tk [24] (described in a later section) which provides an interpreted scripting language for creating zoomable applications. The standard objects that pad++ supports are colored text, graphics, images, portals and hypertext markup language (HTML). Standard input widgets (buttons, sliders, etc.) are supplied as extensions.

One focus in the current implementation has been to provide smooth zooming within very large graphical datasets. The nature of the Pad++ interface requires consistent high frame-rate interactions, even as the dataspace becomes large and the scene gets complicated. In many applications, speed is important, but not critical to functionality. In Pad++, however, the interface paradigm is inherently interactive. One important searching strategy is to visually explore the dataspace while zooming through it, so it is essential that interactive frame rates be maintained.

A second focus has been to design Pad++ to make it relatively easy for third parties to build applications using it. To that end, we have made a clear division between what we call the 'substrate' and applications. The substrate, written in C++, is part of every release and has a well-defined API. It has been written with care to ensure efficiency and generality. It is connected to a scripting language (currently Tcl, but we are exploring alternatives) that provides a fairly high-level interface to the complex graphics and interactions available. While the scripting language runs quite slowly, it is used as a glue language for creating interfaces and putting them together. The actual interaction and rendering is performed by the C++ substrate. This approach allows people to develop applications for Pad++ while avoiding the complexities inherent in this type of system. (See the Implementation section for more information on this.)

2.1. PadDraw: A Sample Application

PadDraw is a sample drawing application built on top of Pad++. It supports interactive drawing and manipulation of objects as well as loading of predefined or programmatically created objects. This application is written entirely in Tcl (the scripting language) and was used to produce all the figures depicted in this paper. The tools, such as navigation aids, hyperlinks and the outline browser, that we discuss later, are part of this application.

The basic user interface for navigating in PadDraw uses a three button mouse. The left button is mode dependent and lets users select and move objects, draw graphical objects, follow hyperlinks, etc. The middle button zooms in and the right button zooms out. Zooming is always centered on the cursor, so moving the mouse while zooming lets the user dynamically control which point they are zooming around.

PadDraw has a primitive Graphical User Interface (GUI) builder that is in progress. Among other things, it allows the creation of active objects. Active objects can animate the view to other locations (a kind of hyperlink) or move other objects around on the surface.

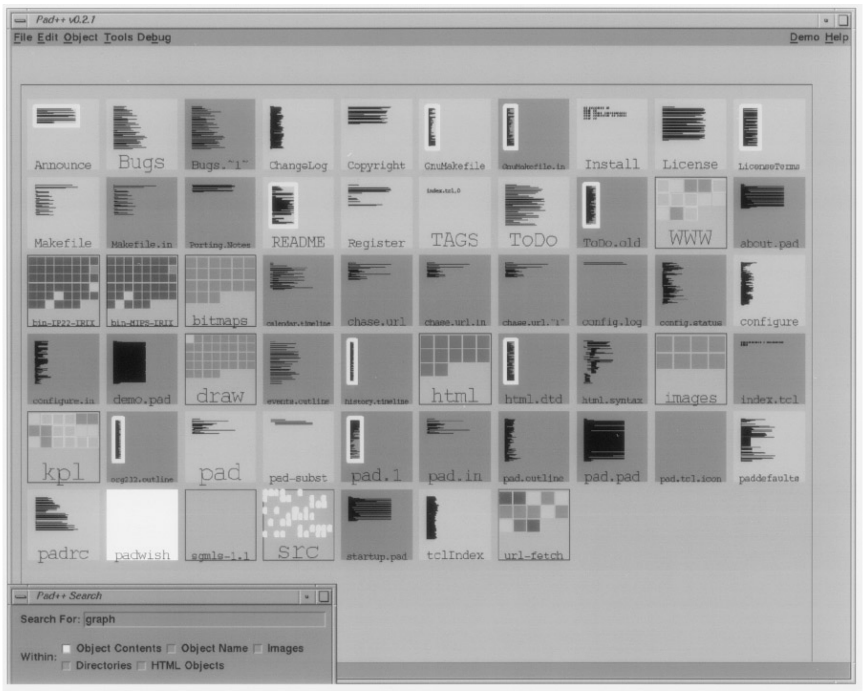
2.1.1. Navigation

Easily finding information on the Pad++ surface is obviously very important since intuitive navigation through large dataspaces is one of its primary motivations. Pad++ supports visual searching with direct manipulation panning and zooming in addition to traditional mechanisms, such as content-based search.

Some applications animate the view to a certain piece of data. These animations interpolate in pan and zoom to bring the view to the specified location. If the end point is further than one screen width away from the starting point, the animation zooms out to a point midway between the starting and ending points, far enough out so that both points are visible. The animation then smoothly zooms in to the destination. This gives both a sense of context to the viewer as well as speeding up the animation since most of the panning is performed when zoomed out which covers much more ground than panning while zoomed in. See the section on Space-Scale Diagrams for more detail on the surprisingly complex topic of multiscale navigation.

Content-based search mechanisms support search for text and object names. Entering text in a search menu results in a list of all of the objects that contain that text. Clicking on an element of this list produces an automatic animation to that object. The search also highlights objects on the data surface that match the search specification with special markers (currently a bright yellow outline) that remain visible no matter how far you zoom out. Even though the object may be so small as to be invisible, its marker will still be visible. This is a simple example of task-based semantic zooming. See Figure 2 for a depiction of the content-based search mechanism.

We have also implemented visual bookmarks as another navigational aid. Users can



Search For: graph

Within: [✓ Object Contents] Object Name Images Directories HTML Objects

Figure 2. The content-based search window lets users search for text and names, and then animate to any of those objects by clicking on the search entry

remember places they have been, and maintain miniature views onto those places. Moving the mouse over one of these bookmark views places a marker in the main view to identify where it will take you (although the marker may be off to the side and hence not visible). Clicking on a view animates the main view to that place (Figure 3).

2.2. Portals

Portals are special items that provide views onto other areas of the Pad++ surface, or even other surfaces. Each portal passes interaction events that occur within it to the place it is looking. Thus, you can pan and zoom within a portal. In fact, you can perform any kind of interaction through a portal. Portals can filter input events, providing a mechanism for changing behavior of objects when viewed through a portal. Portals can also change the way objects are presented. When used in this fashion, we call them *lenses* (see below).

Portals can be used to replicate information efficiently, and also provide a method to bring physically separate data near each other. Figure 1 was created using several portals, each looking at approximately the same place at different magnifications.

Portals can also be used to create indices. For example, creating a portal that looks onto a hyperlink allows the hyperlink to be followed by clicking on it within the portal, changing the main view. This however, may move the hyperlink off the screen.

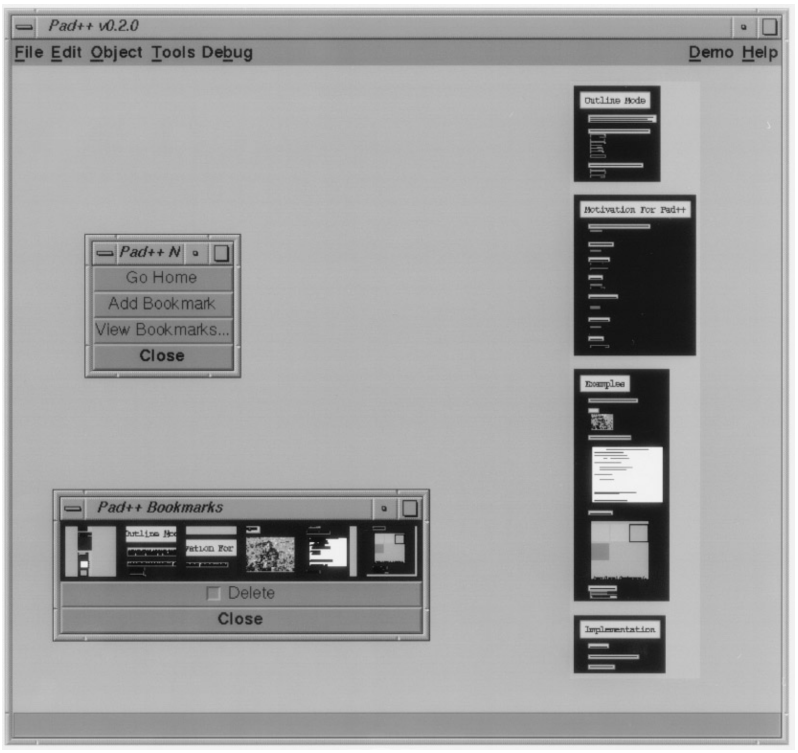


Figure 3. Visual bookmarks let users remember interesting places they have been by showing miniature views of those places. Clicking on one of the views animates the main view to the location

We can solve this by making the portal (or any other object for that matter) *sticky*, which is a method of keeping the portal from moving around as the user pans and zooms. Making an object sticky effectively lifts it off the Pad++ surface and sticks it to the monitor glass. Thus, clicking on a hyperlink through a sticky portal brings you to the link destination, but the portal index is not lost and can continue to be used.

2.3. Lenses

Designing user interfaces is typically done at a low level, focusing on user interface components rather than on the task at hand. If the task is to enter a number, we should be able to place a generic number entry mechanism in the interface. However, typically, once the specific number entry widget, such as a slider or dial, is decided on, it is fixed in the interface.

We can use lenses to design interfaces at the level of specific tasks. For example, we have designed a pair of number entry lenses for Pad++ that can change a generic number entry mechanism into a slider or dial, as the user prefers. By default the generic number entry mechanism allows entering a number by typing. However, dragging the *slider lens* over it changes the representation of the number from text to a slider, and now the mouse can be used to change the number. Another lens shows the data as a dial and lets you modify that with a mouse as well.

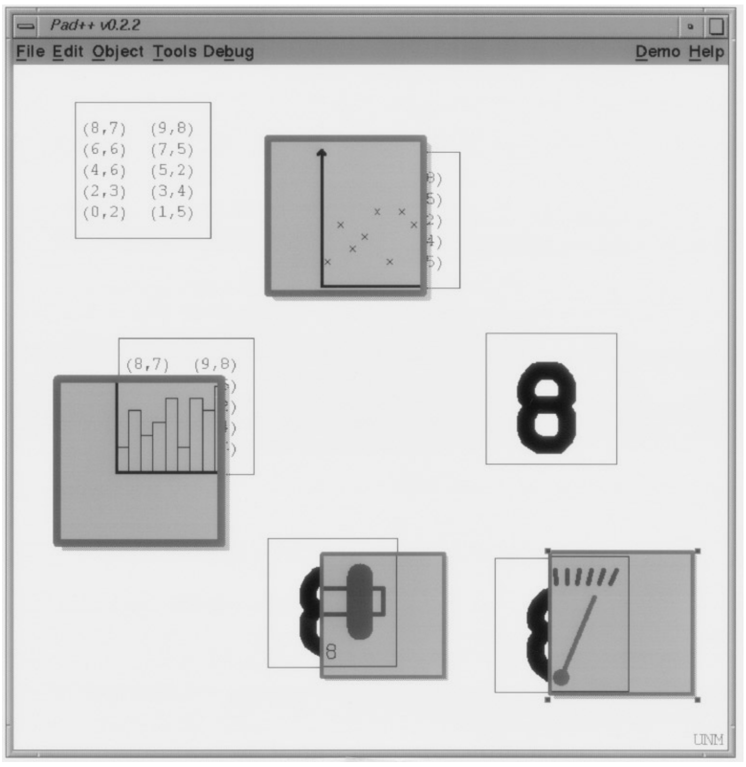


Figure 4. Lenses that show textual data as scatter plots and bar charts

More generally, lenses are objects that alter appearance and behavior of components seen through them. They can be dragged around the Pad++ surface examining existing data. For example, data might normally be depicted by columns of numbers. However, looking at the same data through a lens could show that data as a scatter plot, or a bar chart (see Figure 4).

Lenses such as these support multiple representations so that information can be displayed in ways most effective for the task at hand. They make the notion of multiple representations of the same underlying data more intuitive and can be used to show linkages between the representations. For example, if the slider lens only partially covers the text number entry widge, then modifying the underlying number with either mechanism (text or mouse) modifies both. So typing in the text entry moves the slider, and vice versa.

2.4. Semantic Zooming

Once we make zooming a standard part of the interface, many parts of the interface need to be reevaluated. For example, we can use semantic zooming to change the way things look depending on their size. As we mentioned, zooming provides a natural mechanism for representing abstractions of objects. It is natural to see extra details of an object when zoomed in and viewing it up close. When zoomed out, instead of simply seeing a scaled down version of the object, it is potentially more effective to see a different representation of it.

For example, we implemented a digital clock that at normal size shows the hours and minutes. When zooming in, instead of making the text very large, it shows the seconds, and then eventually the date as well. Similarly, zooming out shows just the hour. An analog clock (implemented as a lens that can be positioned over a digital clock) is similar—it does not show the second hand or the minute markings when zoomed out.

Semantic zooming can take an even more active role in the interface. It can be used as a primary mechanism for retrieving data. We have built prototype tools for accessing system usage including information about the print queue, the system load and the users on the machine. They are depicted as small objects with labels. Zooming into each of them starts a process which gathers the appropriate information and shows it in the now larger object. Zooming out makes the information disappear and the data-gathering process inactive.

3. Visualizations

We are exploring several different types of interactive visualizations within Pad++, some of which are described briefly here. Each takes advantage of the variable resolution available for both representation and interaction.

Layout of graphical objects within a multi-resolution space is an interesting problem, and is quite different than traditional fixed-resolution layout. Deciding how to visually represent an arbitrary graph on a non-zoomable surface is extremely difficult. Often it is impossible to position all objects near logically related objects. In addition, representing the links between objects often requires overlapping or crossing edges. Even laying out a tree is difficult because, generally speaking, there are an exponential number of children that will not fit in a fixed size space.

Traditional layout techniques use sophisticated iterative, adaptive algorithms for laying out general graphs, and still result in graphs that are hard to understand. Large trees are often represented hierarchically with one sub-tree depicted by a single box that references another tree.

Using an interactive zoomable surface, however, allows very different methods of visually representing large data structures. The fact that there is always more room to put information 'between the cracks' gives many more options. Pad++ is particularly well suited to visualizing hierarchical data because information that is deeper in the hierarchy can be made smaller. Accessing this information is accomplished by zooming.

3.1. Hypertext Markup Language (HTML)

In traditional window-based systems, there is no graphical depiction of the relationship among windows even when there is a strong semantic relationship. For example, in many hypertext systems, clicking on a hyperlink brings up a new window with the linked text (or alternatively replaces the contents of the existing window). While there is an important relationship between these windows (parent and child), this relationship is not represented.

We are experimenting with multiscale layouts of hypertext document traversals where the parent–child relationships between links is represented visually. The layout represents a tree that is distorted so that the page that has the focus (i.e. the one being looked at) is quite large. As nodes get further away from the focus, they get smaller. The distortion is controllable with a pop-up window. This is an example of a graphical fisheye view [15]. As links are followed, they are added to the tree and become the current focus. The view is animated so that the new node is centered and large enough to read.

Pad++ reads hypertext written in the Hypertext Markup Language (HTML), the language used to describe hypertext documents used by WWW browsers such as Mosaic and Netscape. Pad++ also can follow links across the internet. Figure 5 shows a snapshot where several hypertext links have been followed. Two views show the same tree focused on different nodes. The Pad++ user interface for accessing hypertext is similar to traditional systems, but zooming mechanisms are employed. There are also special mechanisms to return to an object's parent.

An alternative layout technique (not shown here) uses a camera with a special zoomed in view of the tree. The idea is to give an overview of the tree in one view while allowing individual pages to be read in another view. This gives both a global context and local detail simultaneously. The camera can be dragged around the overview, and the detail view is updated to see what the camera is pointing at. Clicking on a page causes the camera to animate to that page and, when a new page is brought in, the camera centers its view on it.

This layout problem is challenging because the underlying data can be an arbitrary cyclic graph. Any graph can be viewed as a hierarchy by taking a single node and calling it the root node. Imagine taking that node and shaking the graph out. Its neighbors become children, and the children's neighbors become grandchildren, etc. We use this approach to display HTML documents where the order of the links that are followed describe the particular hierarchy imposed on the data. When a cycle is encountered (i.e. a link is followed to a page that is already loaded), the user is brought to the original copy of the page that was loaded, and the focus is put upon it.

3.2. Directory Browser

We built a zoomable directory browser as another exploration of multiscale layout. The directory browser provides a graphical interface for accessing the directory structure of a filesystem (see Figure 6). Each directory is represented by a folder icon and files are represented by solid squares colored by file type. Both directories and files show their filenames as labels when the user is sufficiently close to be able to read them. Each directory has all of its subdirectories and files organized alphabetically inside it. Searching through the directory structure can be done by zooming in and out of the directory tree, or by using the content based search mechanisms described above. Zooming into a file automatically loads its text or image inside the colored square and it can then be annotated. At any particular view, typically three levels of the hierarchy are visible.

3.2.1. Timeline

Scale can be used to convey temporal information. Events which take place over a long period of time use a large scale and brief events are shown at a small scale. We used this notion to visualize some of the history of computing and user interfaces.

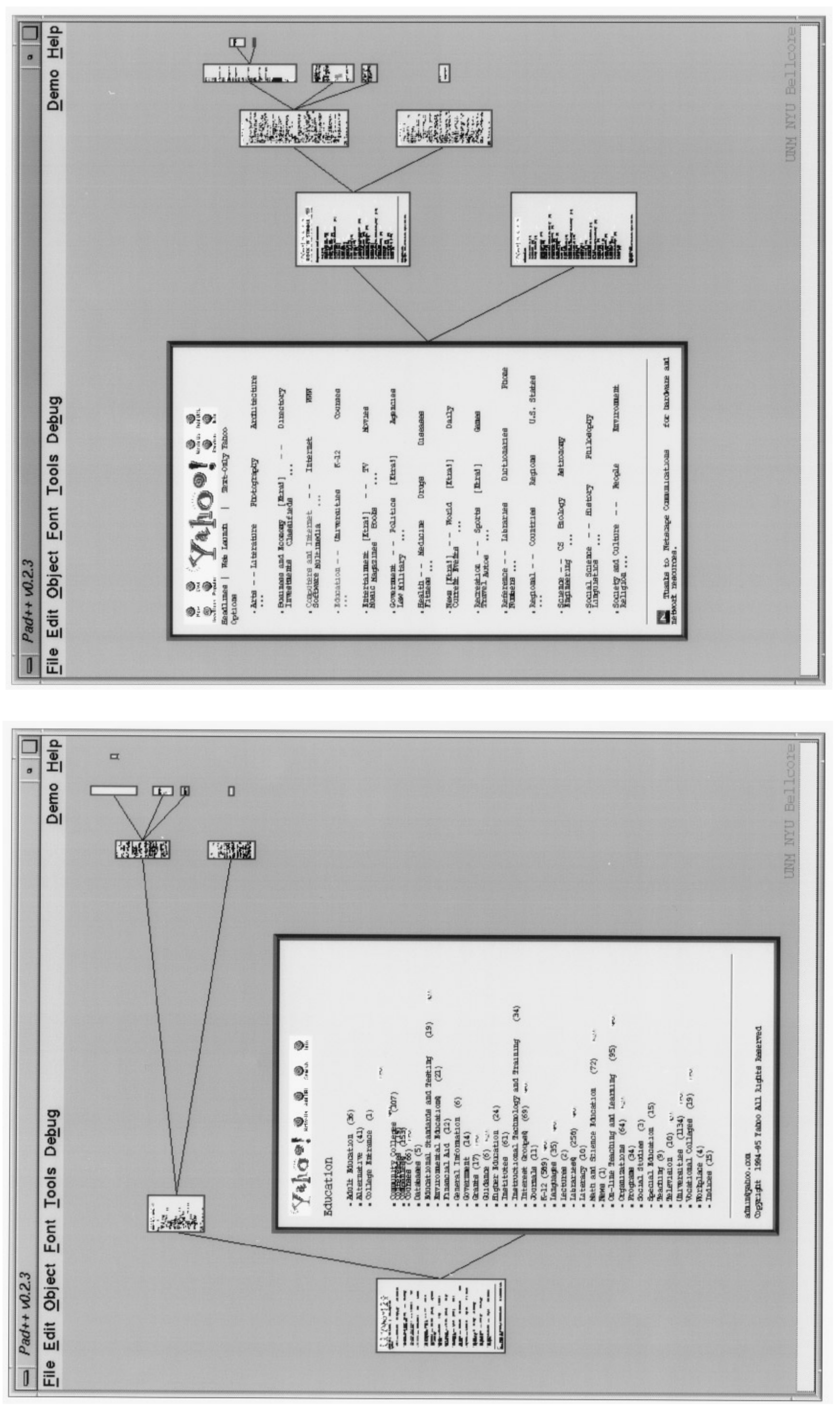


Figure 5. Many different HTML documents loaded in Pad++. Their layout implicitly shows the history of the user's interaction. The two views show the same tree focused on different nodes

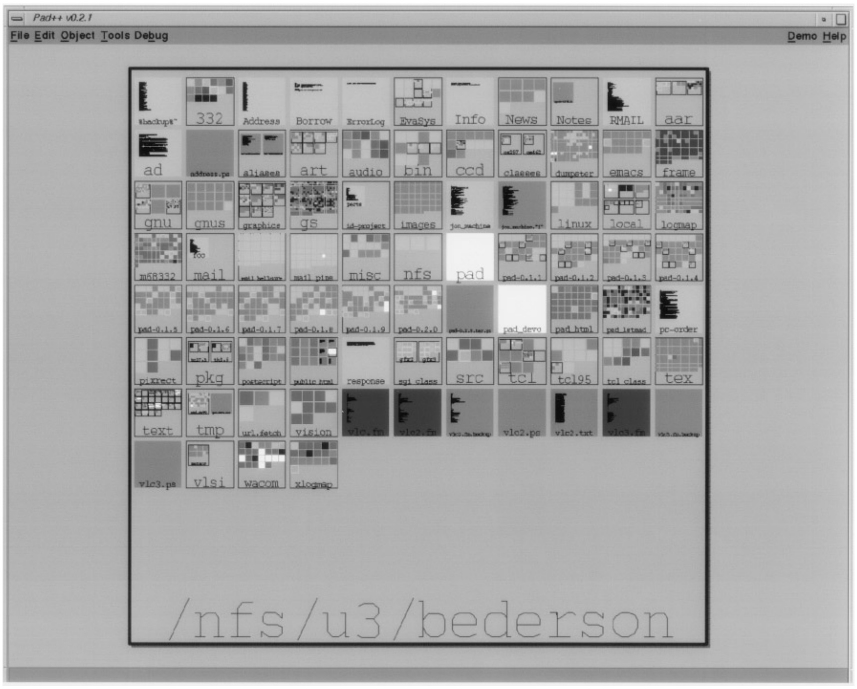


Figure 6. A view of our file system

The timeline visualization shows decades as large numbers. Zooming in on a decade reveals the years within that decade. Further zooming on a particular year shows events which took place during that year. Figure 7 shows a sequence of snapshots as the view is zoomed in.

3.2.2. *Oval Document Layout*

Since objects on the Pad++ surface reside at absolute locations, the relative positions of objects can be used to encode information. Thus, with the Pad++ HTML browser, position is used to encode the order of a user's traversal of a hypertext document. In the Oval Document Layout, position is used to reinforce the narrative structure of documents (such as guided tours) in which the reader follows a sequence of steps which eventually lead back to the starting point (Figure 8).

In this layout, the first page is placed at the bottom edge of an arc. Subsequent pages are placed around the edge of the arc and are drawn at a scale which reflects their position in the tour—middle pages are shown distant and small, whereas start and end pages appear larger and closer to the user.

Navigation buttons at the bottom edge of each page are used to advance through the document. Clicking on a page when it is distant causes Pad++ to pan and zoom so that the page fills most of the screen.

One advantage of this layout is that as the system animates from one page to the next, the user can infer progress through the document by the direction of the animation: near the start, pages move down and to the left; towards the end, pages move up and to the right.

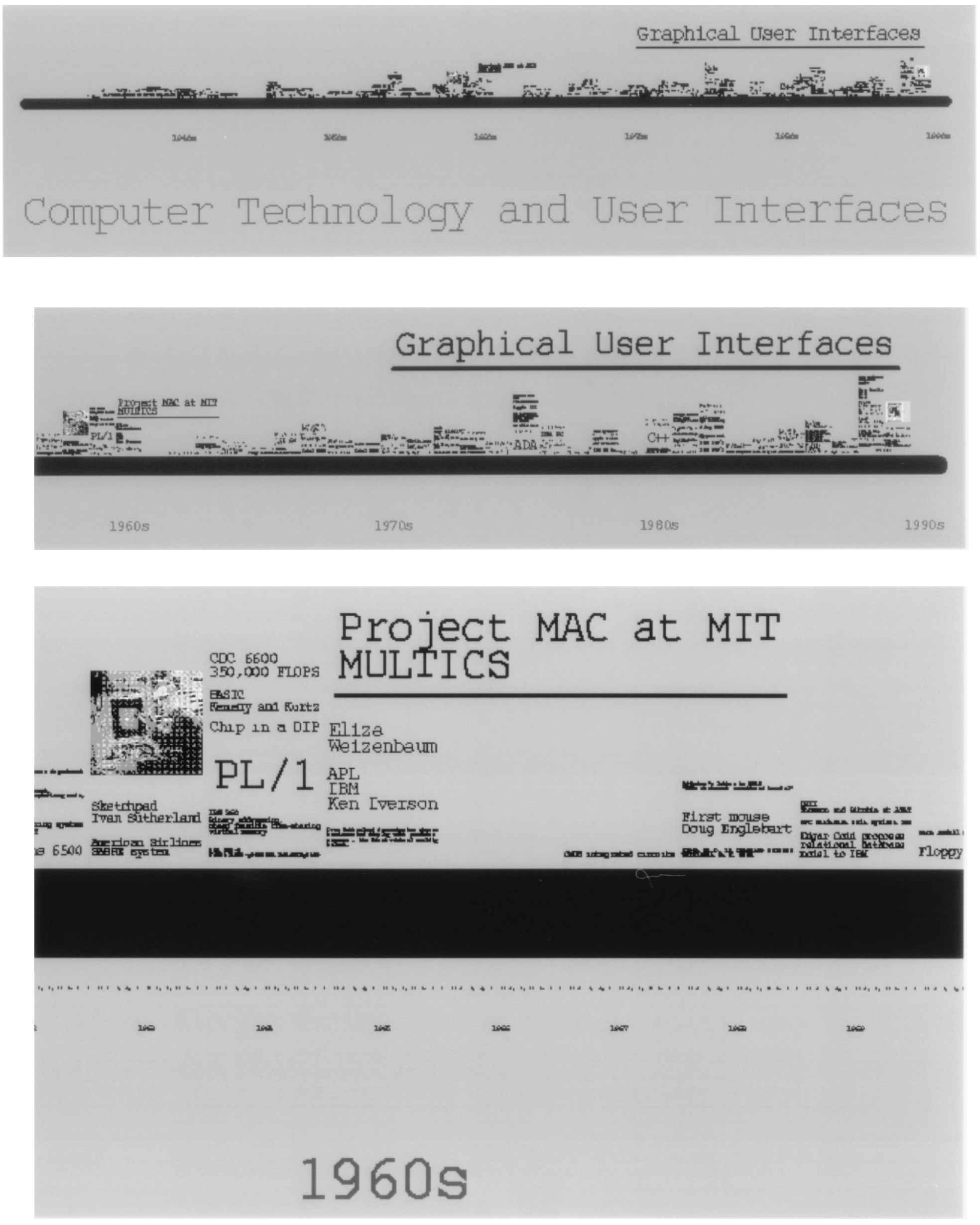


Figure 7. A sequence that views the history of computers and interfaces

The layout is also effective for non-linear access to pages within the document. Zooming out a small distance reveals the whole document, and clicking on a page within the document takes you to that page.

Hotwords and hyperlink buttons in an oval document can be shown with arrows which point towards the destination object. Clicking on the hyperlink animates the Pad++ surface in the direction indicated by the arrow, reducing the

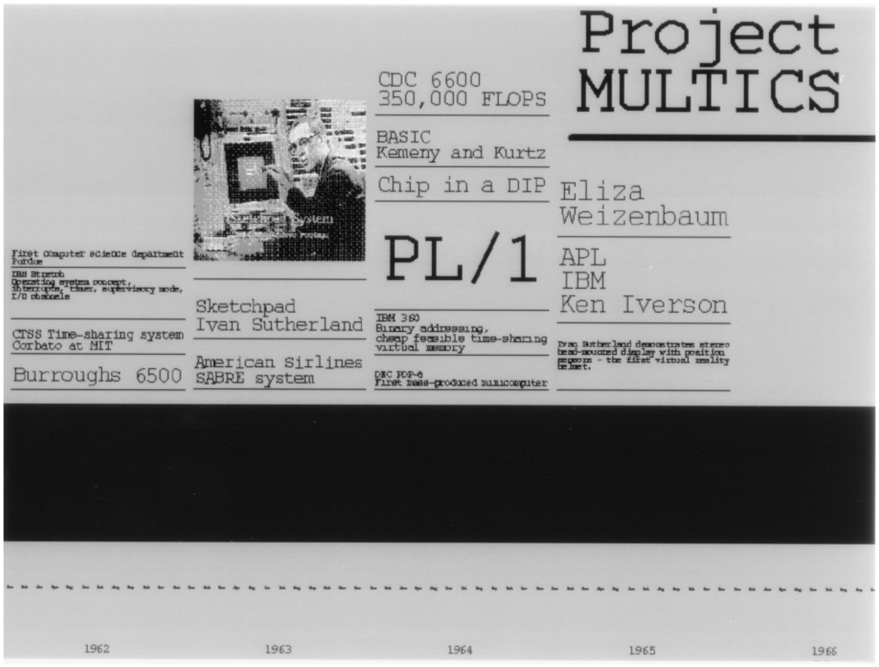


Figure 7. *Continued*

sense of disorientation that many users experience when navigating hypertext documents.

The Oval document view illustrates that a pan/zoom coordinate system can lead to interesting new ways of laying out even traditional page based material. However, the layout has several drawbacks. It is only practical for relatively short documents and for documents which adopt a circular narrative structure.

4. Space-Scale Diagrams

In an effort to understand multiscale spaces better, we have developed an analytical tool for describing them which we call *space-scale diagrams.* By representing the spatial structure of an information world at all its different magnifications simultaneously, these diagrams allow us to visualize various aspects of zoomable interfaces and analyze their properties. We discuss these diagrams briefly here. They are discussed in more detail in [16].

While Pad++ provides panning and zooming interactions over a two dimensional surface, the basic ideas of a space-scale diagram are most easily illustrated in one dimension. This would typically be a slice through a two-dimensional world.

The basic one-dimensional diagram concept is illustrated in Figure 9. This diagram shows six points that are copied over and over at all possible magnifications. These copies are stacked up systematically to create a two dimensional diagram whose horizontal axes represents the original spatial dimension and whose vertical axis represents the degree of magnification (or scale). Because the diagram shows an infinite number of magnifications, each point is represented by a line emanating from

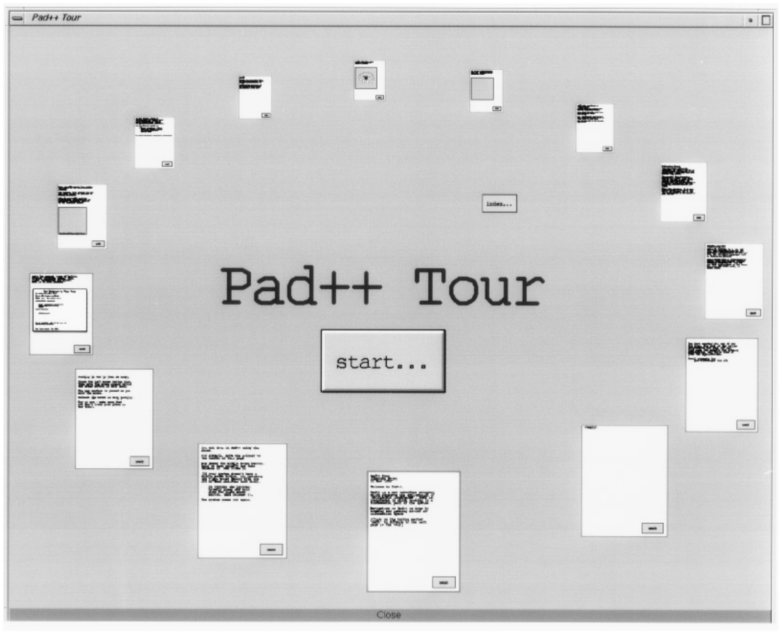


Figure 8. Pad++ help screen with oval document layout

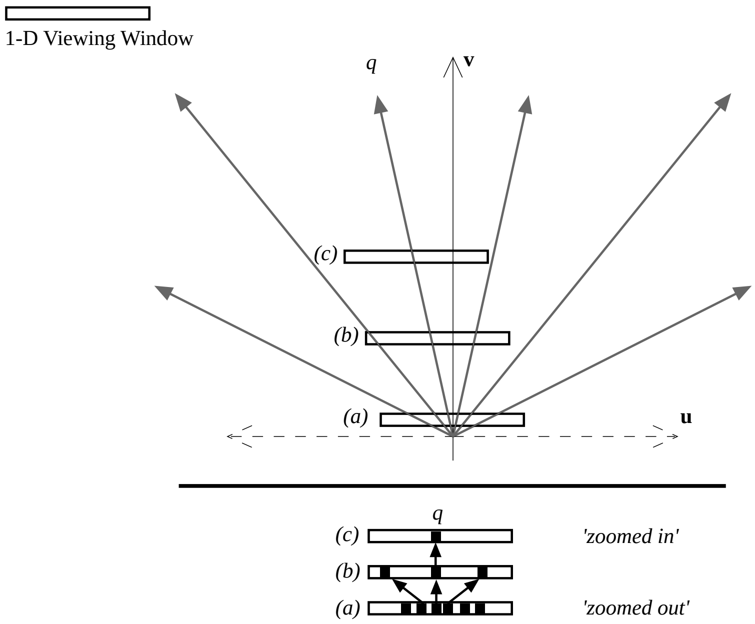


Figure 9. A one dimensional space-scale diagram of six points as the view zooms in from (*a*) to (*b*) to (*c*) around point *q*

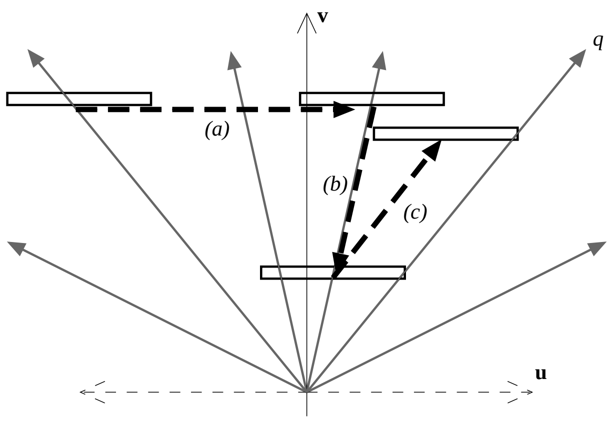


Figure 10. Basic pan–zoom trajectories are shown in the heavy dashed lines; (*a*) is pure pan, (*b*) is pure zoom, (*c*) is zoom around point *q*

the origin. We call these lines *great rays.* In the 2-D analog, whole 2D pictures would be stacked up at all magnifications, forming a 3D space-scale diagram, with points still becoming great rays and 2D regions becoming cones.

For comparison, compare this with a standard one-dimensional world. Here, a standard viewer is a small one-dimensional window that shows a small piece of the world (e.g. a view of a local-piece of a time-line). As the window is panned around it moves to different parts of that time line. As it is zoomed in, it would narrow its scope and look at a smaller region of the time line in detail. As it is zoomed out, it looks at a larger section.

In space-scale diagrams, however, while the viewing window is also represented as a one-dimensional segment, it has a constant size and is located at a particular place in both space and scale. Thus, as the user pans and zooms around the world, the viewing segment is moved rigidly (i.e. without changing its size) in space-scale. A whole sequence of such movements can be represented by a path through the space-scale diagram (Figure 10). Thus, the first advantage of these diagrams is that, by reifying scale, they allow these multiscale movements to be represented statically and so are easier to analyze. For example, a pan operation becomes a horizontal part of such a path. A zoom becomes a movement along a great ray. Other types of movement correspond to curves of other characteristic shapes.

The ability of space-scale diagrams to represent pan-zoom movements as a path in space-scale has allowed us to solve two concrete problems in designing good pan-zoom interactions. Both concern situations where the system needs to move the user's view automatically to another point in the space. This might happen, for example, as the result of following some sort of hyperlink mechanism, or jumping to the result of some content-based search.

The first problem occurs when the interface needs to not only move the user to some other region of the world, but also needs to zoom in. The solution to doing these actions in parallel, jointly panning and zooming to the new view, is not as simple as it might seem. However, if one simply computes how much to pan and how much to zoom and does the two independently in parallel, the result is disconcertingly non-monotonic. The pan covers distance at a constant pace while the zoom-in is magnifying the world exponentially. The result is that the target location first rushes

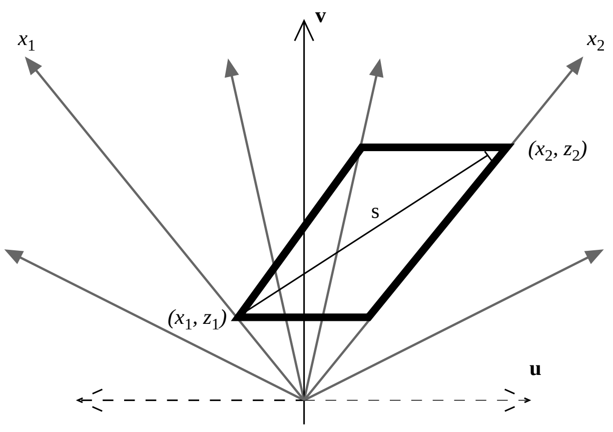


Figure 11. Solution to the simple joint pan–zoom problem. The trajectory s monotonically approaches point (x_2, z_2) in both pan and zoom

away due to the magnification, and only later does the pan catch up. Various hacks to fix this, taking logs and powers of various things, did not work.

Fortunately, using space scale diagrams, a monotonic approach to the target view is captured by a kind of parallelogram constraint on trajectories in space-scale. In Figure 11, a path from (x_1, z_1) to (x_2, z_2) that goes outside such a parallelogram is non-monotonic. If the path exits the sides of the parallelogram, it will violate the monotonicity requirement in space. If the path exits the top or bottom of the parallelogram, it will violate the monotonicity requirement in zoom. A simple path that is monotonic is just the diagonal of this parallelogram. A simple coordinate transform that defines these diagrams (given in [16]) allows one to define this path analytically, and yields a rather unintuitive hyperbolic relationship between the two. We have implemented the trajectory derived from the space-scale diagram analysis and have indeed found it far superior to any uninformed effort.

A second pan-zoom problem concerns the notion of shortest paths between two points in this pan-zoom parameter space. This is a curious question because, in space-scale motions, the shortest distance between two points is not generally a straight line. This is because while panning may be expected to take a time or have a cost that is linear in the spatial separation, zoom is logarithmic so that the fastest way to get from some point p to some point q that is far away would be very tedious by pan alone. It is in fact much shorter to zoom out, make a small pan, and then zoom in (see Figure 12.) Inspired by the space-scale diagrams we were able to define an information theoretic metric over space-scale interactions: the cost of a path is a function of the number of bits that would take to transmit a movie of the motion. Then we addressed the question of finding good paths through the space, i.e. make the movie as small as possible. We found that for points less than a few window widths away, a pure panning motion is pretty good, but for points far away, zoom must play a major role.

Another use of space-scale diagrams is to represent *semantic zooming,* where objects change not just their size but also their appearance when they are magnified. For example, an object could appear as a point when small. As it grows, it could then in turn appear as a solid rectangle, then a labeled rectangle, then a page of text, etc.

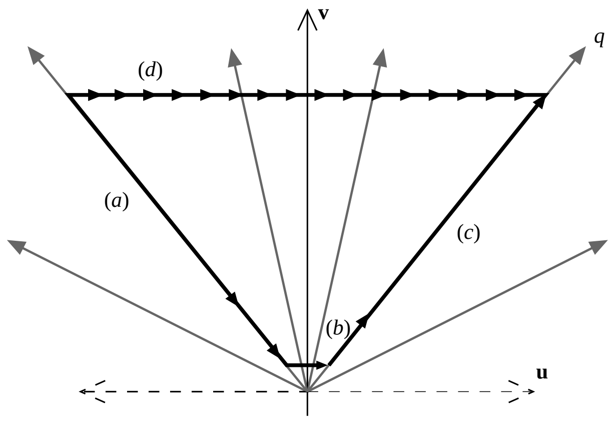


Figure 12. The shortest path between two points is often not a straight line. Here, each arrow represents one unit of cost. Because zoom is logarithmic, it is often shorter to zoom out (*a*), make a small pan (*b*) and zoom back in (*c*), than to make a large pan (*d*)

Figure 13 shows how semantic zooming differs from ordinary geometric zooming, in that the triangular regions change along the scale axis. By explicitly representing scale, the scale-dependent aspects of an object's representation can be made visible. We intend to use such diagrams to help create semantically zoomable objects. The idea is to provide an editing environment where transition boundaries could be moved or aligned by direct manipulation.

All these uses of multiscale diagrams capitalize on the fact that they statically represent scale so that multiscale concepts, which are inherently temporal, are more readily analyzed.

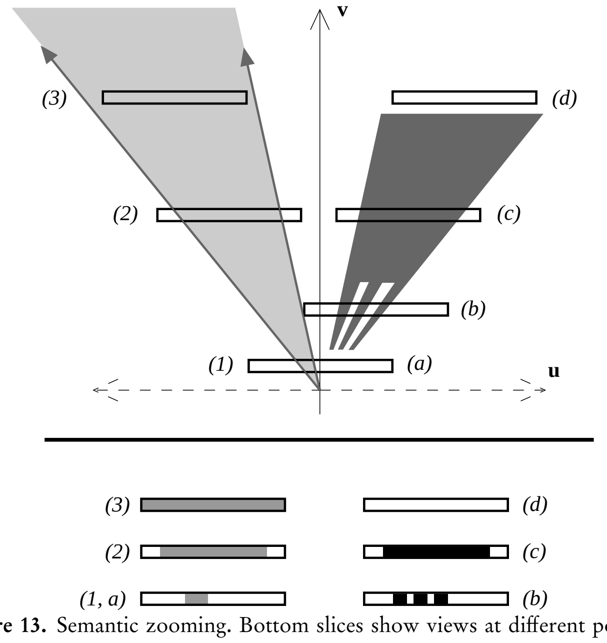


Figure 13. Semantic zooming. Bottom slices show views at different points

5. Procedural Animation

We are also using Pad++ as a substrate for building user-definable animated objects such as complex interface widgets. We have recently applied the same techniques to create animated human-like actors [27]. Although the widgets are much simpler, we employ the same mechanisms that allow us to control human-like movements and gestures to simulate personality and intentionality. The ultimate goal is to support an informational physics in which objects animate naturally. Using these tools, the Pad++ application designer can always convey to the user a clearly structured animated narrative instead of merely an assortment of disjoint temporal events.

We approach this goal by providing a mechanism for the definition of moveable graphical objects. In addition, we define high-level hierarchical control mechanisms for the movements. We are starting to define simple widgets such as buttons and sliders at a behavioral level that makes it easier for application developers to easily change the look and feel of an application. While the widget definitions we supply have a traditional Motif-like look and feel, a designer can easily change their visual style or interaction mechanism.

In addition, we are exploring novel widgets that take advantage of the Pad++ zooming environment. We used our extension mechanisms to implement a choice widget that provides an alternative to the traditional pop-up menu. Figure 14 shows two views of the same widget. The view on the left shows a zoomed out view. Here the widget just shows its current value. On the right we have zoomed into the widget and now the available choices become visible for user selection.

These widgets are implemented with our KPL rendering language. This language was designed to allow very fast run-time recompilation compact representation and efficient execution (roughly 100 times faster execution time than Tcl). It is a post-fix

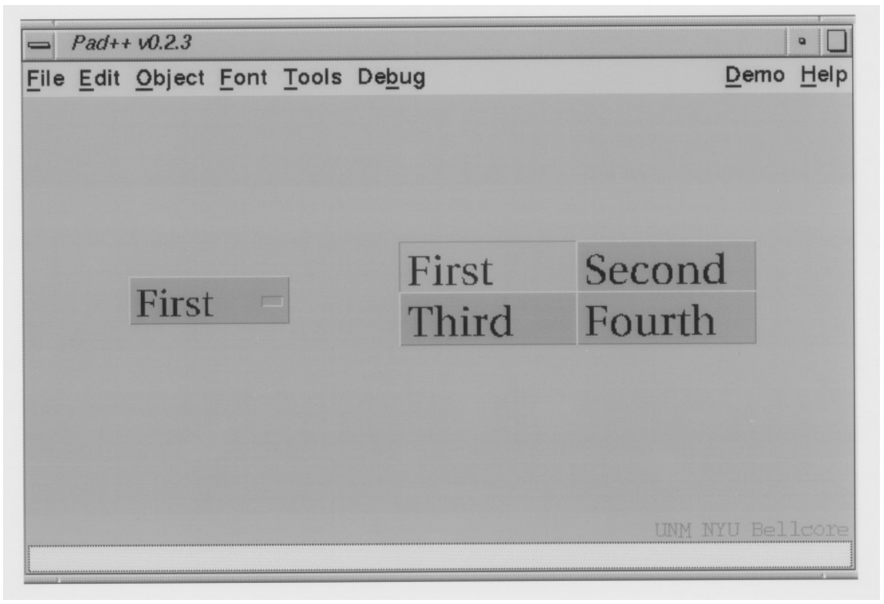


Figure 14. A prototype zoomable choice widget

stack language whose simple structure allows execution roughly 10 times faster than other interpreted or byte-compiled languages. KPL's speed allows us to execute scripts during each render. Without this efficient mechanism, we would only be able to render items pre-defined in the C++ substrate.

The next step uses KPL to create complex animations by the definition of simple repetitive motions of objects based on stochastic processes along with a built-in mechanism to make an automatic transition between different motions. The stochastic processes are defined by rotation axes, periods and magnitudes with some coherent noise [27] applied to give more natural behavior. The mechanism to change between motions gives the hierarchical control described above.

By changing the parameters of the stochastic movement in response to the environment and chaining sequences of motions, together with the transitioning mechanism, we are able to build complex animated behavior in the user interface.

We handle transitions between two actions having different tempos via morphing approach. At the start of the transition, we use the tempo of the first action, and at the end, we use the tempo of the second action. During the time of the transition, we continuously vary the speed of the master clock from the first to the second tempo. In this way, any phase dependent synchronization of the two actions is always preserved during transitions. We may also define new actions as extended transitions between two or more other actions. When there are multiple actors, each actor maintains its own individual tempo.

A related notion that we are exploring is peripheral attention. How does an actor convey that a process is proceeding normally or abnormally, without distracting the user from his/her current tasks? This is especially important in a zoomable environment where the ability to provide peripheral awareness of processes is an important attribute of the paradigm.

We are also studying the semantics of the discrete state transitions that visually represent shifts in attention. In this way an actor on the Pad++ surface can quickly convey to users which other actors and users it is interacting with. We are also interested in determining to what extent we can encode the texture of interactions in order to convey the visual impression of complex activities going on at different scales without requiring all the detail to be specified. We suspect that some of the same techniques used in character animation might be effective here too.

6. Implementation

Pad++ is implemented in C++ under various versions of the Unix operating system using the standard X graphics library system. It currently runs on SGIs, Suns, IBM RS-6000s, PCs running Linux, and should be trivially portable to other Unix systems. Pad++ is implemented as a widget in Tcl/Tk and thus allows applications to be written in the interpreted Tcl language. All Pad++ features are accessible through Tcl making it unnecessary to write any C++ code for new applications.

6.1. Efficiency

In order to keep the animation frame-rate up as the dataspace size and complexity increases, we utilized several standard efficiency methods in our implementation which taken together create a powerful system. We have successfully loaded over 600 000 objects (with the directory browser) and maintained interactive rates of about 10 frames per second. Even when objects are not visible, appropriate checks must be done each time there is movement to see if those objects should now be visible. The key is that the rendering system takes a time roughly proportional to the number of visible objects, independent of the number of objects in the database (on average).

Briefly, the efficiency methods we use in Pad++ include:

- **Spatial Indexing:** Objects are stored internally in a hierarchy based on bounding boxes which allow fast indexing to visible objects.
- **Clustering:** Pad++ automatically restructures the hierarchy of objects to maintain a balanced tree which is necessary for the fastest indexing.
- **Refinement:** Renders fast while navigating by using lower resolution, and not drawing very small items. When the system is idle for a short time, the scene is successively refined, until it is drawn at maximum resolution.
- **Level-Of-Detail:** Renders items differently depending on how large they appear on the screen. If they are small, renders them with lower resolution.
- **Region Management:** Only updates the portion of the screen that has been changed. Linked with refinement, this allows different areas of the screen to refine independently.
- **Clipping:** Only renders the portions of large objects that are actually visible. This applies to images and text.
- **Adjustable Frame Rate:** Animations and zooming maintain constant perceptual flow, independent of processor speed, scene complexity, and window size. This is accomplished by rendering more or fewer frames, as time allows.
- **Interruption:** Slow tasks, such as animation and refinement, are interruptible by certain input events (such as key-presses and mouse-clicks). Animations are immediately brought to their end state and refinement is interrupted, immediately returning control to the user.
- **Ephemeral Objects:** Certain objects that represent large disk-based datasets (such as the directory browser) can be tagged ephemeral. They will automatically get removed when they have not been rendered for some time, and then will get reloaded when they become visible again.
- **Optimized Image Rendering:** The code to render zoomed images has been very carefully optimized and allows real-time zooming of high-resolution images.

6.2. Scripting Language Interface

An important consideration in the design and implementation of Pad++ is how to create a very fast and efficient graphics system, and yet still make it extensible. We wanted to make sure that we and others would be able to experiment easily with new interface mechanisms. Originally, Pad++ was implemented entirely in C++, making it very difficult for anyone but the authors to add new objects and interactions. Even for the authors, going through the compile and link cycle was very slow and tedious, making it difficult to do much experimentation.

We decided to create an interpreted scripting language interface to Pad++ to get around this problem. This approach is becoming quite common, and works well as long as the scripting language is at the right level. On one side, you want as much as possible to be in the scripting language so that the system is as easy to modify as possible. On the other side, it is critical that all speed-critical code be written as efficiently as possible. In a system like ours, there are three classes of code, each of which have different speed requirements:

- Create objects: Slow—Scripting language is fine
- Handle events: Medium—Small amount of scripting language is ok
- Render scene: Fast—C++ or byte-compiled languages only

Rendering is done in C++ (for built-in Pad++ items) or in an efficient byte-compiled language such as KPL (for user defined widgets or animated items). This results in animation performance which is quite good, even on Linux based PC platforms.

We chose Tcl [24] as our primary scripting language, largely because it comes in combination with Tk, a Motif-like library for creating graphical user interfaces. Pad++ is built as a new widget in Tk. This allows it to be used in combination with standard, non-zooming widgets such as menubars, buttons, slidets, etc. This lets us make complete applications while we build and debug widgets within Pad++. Just as importantly, it provides a mechanism to compare zoomable interfaces with traditional interface mechanisms in the same system.

The Tcl interface to Pad++ is designed to be very similar to the interface to the Tk Canvas widget (which provides a surface for drawing structured graphics). While Pad++ does not implement everything in the Tk Canvas yet, it adds many extra features. The Tcl interface to Pad++ is summarized here to give a feel for what it is like to program Pad++.

We are also experimenting with other scripting languages which are better suited to some tasks—primarily those requiring higher speeds. As mentioned previously, we use KPL for high-speed animations. We also are considering incorporating an alternative language, such as Scheme or Java, for more general programming which needs high speed interaction.

6.3. TCL Interface

There are many commands that create and manipulate objects, each referring to the object's unique integer id. Objects may be grouped by using *tags*, a mechanism for associating data with each object. Every command can be directed to either a specific object id or to a tag, in which case it will apply to all objects that share that tag. Each Pad++ widget has its own name, and all commands start with the name of that widget. In the examples that follow, the name of the widget is `.pad`.

Examples:

- A red rectangle with a black outline is created whose corners are at the points (0, 0) and (200, 100):
  ```
  .pad create rectangle 0 0 200 100 -fill red -pen black
  ```
- Put item number 5 at the point (30, 30), make the object twice as big, and make the object anchored at that point on its northwest corner:
  ```
  .pad itemconfig 5 -anchor nw -place ''30 30 2''
  ```
- Specify that item number 5 should be visible only when its largest dimension is greater than 20 pixels and less than 100 pixels.
  ```
  .pad itemconfig 5 -minsize 20 -maxsize 100
  ```

It is straightforward to get scripts evaluated when specific events hit objects or groups of objects. Simple macros get expanded within the event script to specify information specific to that event. Some examples follow:

- Make all items with tag `foo` turn blue when the left button of the mouse is pressed over any of those objects:
  ```
  .pad bind foo ⟨ButtonPress-1⟩ {
    .pad itemconfig foo -fill blue
  }
  ```
- This is a single event binding for a group of objects that affects just the object clicked on, using the macro '`%O`' to expand to the specific object:
  ```
  .pad bind foo ⟨ButtonPress-1⟩ {
    .pad itemconfig %O -fill blue
  }
  ```

Some basic navigation and searching mechanisms are provided by the Tcl interface. A few basic ones are:

- Smoothly go to the location $(100, 0)$ with a magnification of 5, and take 1000 milliseconds for the animation:
  ```
  .pad moveto 100 0 5 1000
  ```
- Smoothly go to the location such that object number 37 is centered, and fills three quarters of the screen, and take 500 milliseconds for the animation:
  ```
  .pad center 37 500
  ```
- Return the list of objects ids that contain the text 'foo'
  ```
  .pad find withtext foo
  ```

6.4. Events

As briefly mentioned, it is possible to attach event handlers to items on the Pad++ surface so that when a specific event (such as ButtonPress, KeyPress, etc.) hits an item, the appropriate event handler is evaluated. This system operates much as it does with the Tk Canvas widget, but there are several significant additions:

- **Extra Macro Expansions**
  When a command is invoked, several substitutions are made in the text of the command that describe the specific event that invoked the command. In addition to the substitutions that the Tk bind command makes, Pad++ makes a few more. These include mechanisms to find the pad widget and item that actually received the event, the coordinates of the event in Pad++ coordinates, which portals the event went through, and a few other related items.
- **New Events**
  Several new events were created that get fired at special times, depending on the semantics of the event. ⟨Create⟩ gets fired whenever new Pad++ items are created. ⟨Modify⟩ gets fired whenever an item is modified. ⟨Delete⟩ gets fired whenever an item is deleted. ⟨Write⟩ gets fired whenever an item is written out with the Pad++ write command. ⟨PortalIntercept⟩ gets fired just before an event passes through a portal. If the event handler executes the break command, then the event stops at the portal and does not pass through.

- **User-Specified Modifiers**
  Event handlers are defined by sequences of the same format as the Tk bind command. A sequence contains a list of modifiers which are direct mappings hardware such as the shift key, control key, etc. Event handlers only fire sequences with modifiers that are active, as defined by the hardware.
  Pad++ allows user-defined modifiers where the user can control which one of the user-defined modifiers is active (if any). The advantage of modifiers is that many different sets of event bindings may be declared all at once—each with a different user-defined modifier. Then, the application may choose which set of event bindings is active by setting the active user-defined modifier. This situation comes up frequently with many graphical programs where there are modes, and the effect of interacting with the system depends on the current mode.

6.5. 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 currently three types of these callbacks:

- **Render Callbacks**
  A render callback script gets evaluated every time the item is rendered. The script gets executed when the object normally would have been rendered. By default, the object will not get rendered, but the script may render the object at any time with the `renderitem` function. An example follows where item number 22 is modified to call the Tcl procedures beforeMethod and afterMethod surrounding the object's rendering.
  ```
  .pad itemconfig 22 -renderscript {
    beforeMethod
    .pad renderitem
    afterMethod
  }
  ```
  Instead of calling the `renderitem` command, an object can render itself. Several rendering routines are available to render scripts, making it possible to define an object that has any appearance whatsoever. Items which define a render script are called *procedural objects* and are used for creating animated objects (those that change the way they look on every render) and custom objects. They also can be used to implement semantically zoomable objects, since the size of an object is available within the callback.
- **Timer Callbacks**
  A timer callback script gets evaluated at regular intervals, independent of whether the item is being rendered, or receiving events.
- **Zooming Callbacks**
  Zooming callback scripts are evaluated when an item gets rendered at a different size than its previous render, crossing a pre-defined threshold. These are typically used for creating efficient semantically zoomable objects. Since many objects do not change the way they look except when crossing size borders, it is more efficient to avoid having scripts evaluated except for when those borders are crossed.

6.6. Extensions

Pad++ is extensible with Tcl scripts (i.e. no C/C++ code). This provides an easy to use mechanism to define new Pad++ commands as well as compound object types that 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.

Extensions are defined by creating Tcl commands with specific prefixes. Each extension is defined by three commands which allow creation, configuration and invocation of the extension, respectively. Defining the procedures makes them automatically available to Pad++. No specific registration is necessary. All three procedure definitions are necessary for creation of new Pad++ object types, but it is possible to define just the command procedure for defining new commands without defining new object types.

7. Physics-Based Strategies For Interface Design

Exploration of Pad++ is part of a research program to develop alternative strategies for interface design. Our goal is to move beyond mimicking the mechanisms of earlier media and to start to more fully exploit radical new computer-based mechanisms. We propose an information physics view of interface objects that we think provides an effective complement to traditional metaphor-based approaches.

While an informational physics strategy for interface design certainly involves metaphor, we think there is much that is distinctive about a physics-based approach. Traditional metaphor-based approaches map at the level of high-level objects and functionality. They yield interfaces with objects such as windows, trash cans and menus, and functions like opening and closing windows and choosing from menus. While there are ease-of-use benefits from such mappings, they also orient designers towards mimicking mechanisms of earlier media rather than towards exploring potentially more effective computer-based mechanisms. Semantic zooming is but one example mechanism that we think arises more naturally from adopting an informational physics strategy. Even geometric zooming, especially with the orders of magnitude possible in Pad++, is not a mechanism that traditional metaphors would lead designers to investigate.

We are not the first to follow a physics-inspired course in thinking about interface design. It derives, like most interesting interface ideas, from the seminal work of Sutherland [31] on Sketchpad. Simulations and constraint-based interfaces that led to the development of direct manipulation style interfaces are other examples of this general approach. They too derive from Sutherland and continue to inspire developments. Recent examples include the work of Borning and his students [5, 6]. Witkin [33] in particular has taken a physics-as-interface approach to construction of dynamic interactive interfaces.

Smith's Alternate Reality [28, 29] and languages such as Self [32] are also examples of following a physics-based strategy for interface design. These systems make use of techniques normally associated with simulation to help 'blur the distinction between data and interface by unifying both simulation objects and interface objects as concrete objects' [9]. More importantly, they are based on implementation of mechanisms at a different level than is traditional. Smith, for example, gives users access to control of parameters of the underlying physics in his Alternate Reality Kit. With this approach comes the realization that one can do much more than just mimic reality. As Chang and Unger [9] point out about their use of cartoon animation mechanisms in Self, ‘adhering to what is possible in the physical world is not only limiting, but also less effective in achieving *realism.*’

It is important to look at the costs as well as the benefits of traditional, metaphor-based strategies. They can lead away from exploration of new mechanisms and limit views of possible interfaces in at least four ways.

First, metaphors necessarily pre-exist their use. Pre-Copernicans could never have used the metaphor of the solar system for describing the atom. In designing interfaces, one is limited to the metaphorical resources at hand. In addition, the metaphorical reference must be familiar in order to work. An unfamiliar interface metaphor is functionally no metaphor at all. One can never design metaphors the way one can design self-consistent physical descriptions of appearance and behavior. Thus, as an interface design strategy physics, in the sense described above, offers more design options than traditional metaphor-based approaches.

Second, metaphors are temporary bridging concepts. When they become ubiquitous, they die. In the same way that linguistic metaphors lose their metaphorical impact (e.g. *foot of the mountain* or *leg of table*), successful metaphors also wind up as dead metaphors (e.g. file, menu, window, desktop). The familiarity provided by the metaphor during earlier stages of use gives way to a familiarity with the interface due to actual experience.

Thus, after a while, it is the actual details of appearance and behavior (i.e. the physics) rather than any overarching metaphor that form much of the substantive knowledge of an experienced user. Any restrictions that are imposed on the behaviors of the entities of the interface to avoid violations of the initial metaphor are potential restrictions of functionality that may have been employed to better support the users’ tasks and allow the interface to continue to evolve along with the users’ increasing competency.

Similarly, the pervasiveness of dead metaphors, such as files, menus and windows, may well restrict us from thinking about alternative organizations of interaction with the computer. There is a clash between the dead metaphor of a file and newer concepts of persistent distributed object hierarchies.

Third, since the sheer amount and complexity of information with which we need to interact continues to grow, we require interface design strategies that *scale.* A traditional metaphor-based strategy does not scale. A physics approach, on the other hand, scales to organize greater and greater complexity by uniform application of sets of simple laws. In contrast, the greater the complexity of the metaphorical reference, the less likely it is that any particular structural correspondence between metaphorical target and reference will be useful. We see this often as designers start to merge the functionality of separate applications to better serve the integrated nature of complex tasks. Metaphors that work well with the individual simple component applications typically do not integrate smoothly to support the more complex task.

Fourth, it is clear that metaphors can be harmful as well as helpful since they may well lead users to import knowledge not supported by the interface. Our point is not that metaphors are not useful but that, as the primary design strategy, they may well restrict the range of interfaces designers consider and impose less effective trade-offs than those designers might come to if they were led to consider a larger space of possible interfaces.

There are, of course, also costs associated in following a physics-based design strategy. One cost is that designers can no longer rely as heavily on users' familiarity with the metaphorical reference (at least at the level of traditional objects and functionality), and so physics-based designs may take longer to learn. However, the power of metaphor comes early in usage and is rapidly superceded by the power of actual experience. One might want to focus on easily discoverable physics. As is the case with metaphors, all physics are not created equally discoverable or equally fitted to the requirements of human cognition.

8. Future Directions

To adequately explore the effectiveness of the Pad++ substrate and the informational physics design strategy discussed here will require development of a wide range of applications. One domain we plan to investigate is construction of active documents. Most tools for interacting with documents (like World-Wide Web browsers such as Mosaic and Netscape) predefine all of the interactive widgets within the client. Hooks are provided so that documents may access those widgets but there is no method to provide new ones, except to re-define the standards, modify the client and distribute the client to enough of the user population so it becomes the new standard in practice.

Pad++'s extensibility ensures that new widgets can be defined by scripts which can be included with a document. This will allow documents to provide new forms of interactivity without depending on the client to supply it. We are currently in the design stages of an extension to HTML, we call the MultiScale Markup Language (MSML), that will be the markup language to describe documents within Pad++. MSML will allow logical formatting of documents with different sized components and will provide a method for allowing Pad++ scripts to be included with documents—allowing truly active documents.

In addition to data visualizations, we are investigating the use of Pad++ as a replacement for the standard windowing system. PadWin currently consists of a few basic semantically zoomable gauges which display statistics such as the list of tasks being scheduled, the state of the printer queue or the names of people who are logged on. We intend to extend these tools so that most of the computers resources and facilities are accessible through navigation within PadWin. We are also producing a suite of zoomable applications for use in PadWin.

In order to support existing non-zoomable applications, PadWin will incorporate a mechanism to control the placement of application windows on the screen to make them blend into the Pad++ surface. By mapping, unmapping and moving these windows appropriately, PadWin will act as an extended virtual window manager where the effective screen size is huge, and where zoomable and non-zoomable applications reside side by side.

Pad++ also seems well-suited to a collaborative work environment. While the original Pad implementation allowed some basic shared workspaces (running from a single process displaying on multiple X servers), we are designing a more sophisticated approach. The goal is to be able to use portals to look remotely on to any Pad++ surfaces on the network (assuming that the right permissions are set). Each user's system will contain a spatial database server that will send updates to all other systems that have portals looking on to it. With this approach, there may be a lag in retrieving others' data but, once it arrives, it will be cached within the local system so the high-speed interactivity of Pad++ will not be lost.

Finally, we are building a completely visual interface to Pad++ for creation of an interactive visual dataspace. Multimedia authoring tools such as MacroMedia Director™ and Apple's Multimedia Authoring Tool™ are letting visual designers without programming experience create beautiful and complex interactive hypertext data retrieval systems. As we discussed with the layout of HTML, however, having a huge data surface potentially alleviates some of the problems of navigating within a large hypertext document. To this end, we are building a set of tools that will allow non-technical visual designers to create interactive zoomable multimedia systems.

9. Availability

The Pad++ substrate is approaching the point where we can start to make it available to a wider community. Our goal is to make it freely available for non-commercial use. See the Pad++ project home page (http://www.cs.unm.edu/pad++) for current information.

Acknowledgments

This work was supported in part by ARPA contract N66001-94-C-6039 to the University of New Mexico. We especially appreciate the support we have received from Craig Wier as part of the new HCI Initiative at ARPA. We thank David Fox and Matthew Fuchs at NYU and Eric De Mund, David Vick and Jason Stewart at UNM for enjoyable discussions about zoomable interfaces. We also would like to acknowledge members of the Computer Graphics and Interactive Media Research Group at Bellcore for discussions shared during our continuing search for the best cheeseburger.

References

1. R. M. Baecker (1990) *Human factors and typography for more readable programs.* ACM Press, Denver.
2. B. B. Bederson, L. Stead & J. D. Hollan (1994) *Pad++*: Advances in multiscale interfaces. In: *Proceedings of ACM SIGCHI Conference* (*CHI'94*). Addison-Wesley, Reading, MA.
3. B. B. Bederson & J. D. Hollan (1994) *Pad++*: A zooming graphical interface for exploring alternate interface physics. In: *Proceedings of ACM Symposium on User Interface Software and Technology* (*UIST '94*). ACM Press, New York.
4. E. A. Bier, M. C. Stone, K. Pier, W. Buxton & T. D. DeRose (1993) Toolglass and magic lenses: the see-through interface. In: *Proceedings of ACM SIGGRAPH Conference* (*Sigraph '93*). Addison-Wesley, Reading, MA.
5. A. Borning (1979) *Thinglab*: *a constraint-oriented simulation laboratory.* Technical Report SSL-79-3, Xerox Palo Alto Research Center.
6. A. Borning & R. Duisberg (1986) Constraint-based tools for building user interface. *ACM Transactions on graphics,* **5(4)**, 345–374.
7. R. Brooks (1986) A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation* **2(1)**, 14–23,
8. S. K. Card, G. G. Robertson & J. D. Mackinlay (1991) The information visualizer, an information workspace. In: *Proceedings of ACM Human Factors in Computing Systems Conference* (*CHI'91*). Addison-Wesley, Reading, MA.

9. B.-W. Chang & D. Ungar. Animation: from cartoons to the user interface. In: *Proceedings of ACM Symposium on User Interface Software and Technology* (*UIST'93*). ACM Press, New York.
10. S. Deerwester, S. T. Dunais, G. W. Furmas, T. K. Landauer & R. Harshman (1990) Indexing by latent semantic analysis. In: *Journal of American Society of Information Science* **41**, 391–407,
11. W. C. Donelson (1978) Spatial management of information. In: *Proceedings of 1978 ACM SIGGRAPH Conference.* Addison-Wesley, Reading, MA.
12. D. Ebert, *Texturing and Modeling, A Procedural Approach.* Academic Press, London.
13. S. G. Eick, J. L. Steffen & E. E. Sumner, Jr (1992) Seesoft: a tool for visualizing line-oriented software statistics. In: *IEEE Transactions on Software Engineering* **18(11)**, 957–968.
14. K. M. Fairchild, S. E. Poltrock & G. W. Furnas (1980) SemNet: three-dimensional graphic representations of large knowledge bases. In: *Cognitive Science and its Applications for Human-Computer Interaction.* Lawrence Erlbaum Associates, Princetown.
15. G. W. Furnas (1986) Generalized fisheye views. In: *Proceedings of 1986 ACM SIGCHI Conference.* Addison-Wesley, Reading, MA.
16. G. W. Furnas & B. B. Bederson (in press) Space-scale diagrams: understanding multiscale interfaces. In: *Proceedings of ACM SIGCHI'95.* Addison-Wesley, Reading, MA.
17. M. Gleicher (1992) Briar: a constraint-based drawing program. In: *CHI'92 Formal Video Program.* Addison-Wesley, Reading, MA.
18. W. C. Hill, J. D. Hollan, D. Wroblewski & T. McCandless (1992) Edit wear and read wear. In: *Proceedings of ACM SIGCHI'92.* Addison-Wesley, Reading, MA.
19. W. C. Hill & J. D. Hollam (1994) History-enriched digital objects: prototypes and policy issues. *The Information Society,* **10**, 139–145.
20. J. D. Hollan, E. Rich, W. Hill, D. Wroblewski, W. Wilner, K. Wittenburg, J. Grudin & Members of the Human Interface Laboratory (1991) An introduction to HITS: human interface tool suite. In: *Intelligent User Interfaces* (Sullivan & Tyler, eds) pp. 293–337.
21. J. D. Hollan & S. Stornetta (1993) Beyond being there. In: *Proceedings of the ACM SIGCHI'92.* Addison-Wesley, Reading, MA.
22. G. Lakoff & M. Johnson (1980) *Metaphors We Live By.* University of Chicago Press, Illinois.
23. J. D. Mackinlay, G. G. Robertson & S. K. Card (1991) The perspective wall; detail and context smoothly integrated. In: *Proceedings of CHI'91 Human Factors in Computing Systems.* Addison-Wesley, Reading, MA.
24. J. K. Ousterhout (1994) *Tcl and the Tk Toolkit.* Addison Wesley, New York.
25. K. Perlin & D. Fox (1993) Pad: an alternative approach to the computer interface. In: *Proceedings of 1993 ACM SIGGRAPH Conference.* Addison-Wesley, Reading, MA.
26. K. Perlin (1985) An image synthesizer. In: *Proceedings of ACM SIGGRAPH '85* **19**, 287–293.
27. K. Perlin (1994) Danse interactif. In: *Video Proceedings of ACM SIGGRAPH '94,* **28(3)**.
28. R. B. Smith (1986) The alternate reality kit: an animated environment for creating interactive simulations. In: *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages,* Rome.
29. R. B. Smith (1987) Experiences with the alternate reality kit: an example of the tension between literalism and magic. In: *Proceedings of ACM CHI + GI '87 Conference.* ACM Press, New York.
30. M. C. Stone, K. Fishkin & E. A. Bier (in press) The movable filter as a user interface tool. In: *Proceedings of ACM SIGCHI'94.* Addison-Wesley, Reading, MA.
31. I. E. Sutherland (1963) Sketchpad: A man-machine graphical communications systems. In: *Proceedings of the Spring Joint Computer Conference.* Spartan Books, Baltimore, pp. 329–346.
32. D. Ungar & R. B. Smith (1987) Self: The Power of Simplicity. In: *Proceedings of OOPSLA '87 Conference.*
33. A. Witkin, M. Gleicher & W. Welch (1990) Interactive dynamics. *Computer Graphics* **24(2)**, 11–21.

Pad++ Programmer's Guide

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

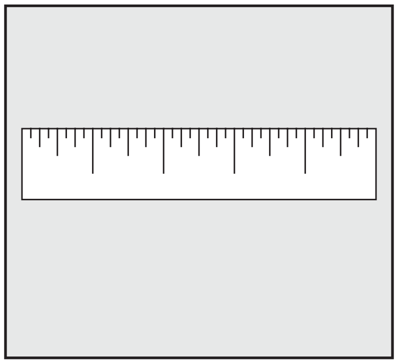
Coordinates are specified and are returned in the current Pad++ units. The default units are pixels (see `-units` configuration option). All coordinates relating to the Pad++ surface are stored as floating-point numbers. All coordinates refer to the Pad++ surface and are independent of the current view. 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
```



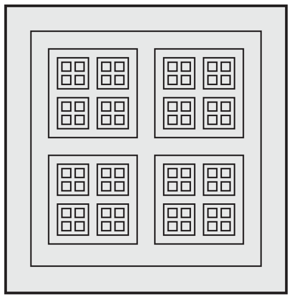
Views are defined by a list of three numbers which specify the (x, y) point at the center of the window, and the magnification of the view, respectively. The default view, (0, 0, 1), has the origin at the center of the screen and is not magnified. The getview command returns the current view, and the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

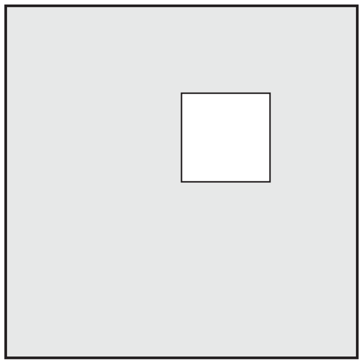
Every item on the Pad++ surface has an anchor and an anchor point associated with it that controls the item's position and size. The anchor and the position of the anchor point can be accessed directly with the `-anchor`, and `-place` itemconfigure options. The `-place` consists of a list of three values 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.

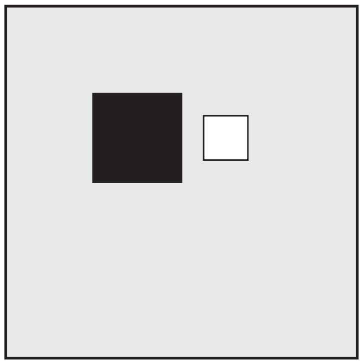
```
.pad create rectangle 0 0 100 100 -tags "rect1" -fill black
```

```
    .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
```

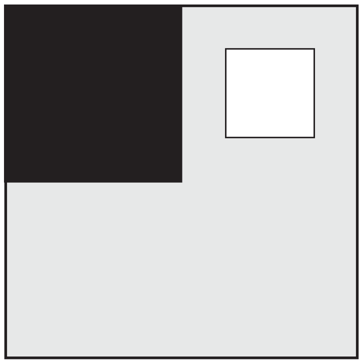


These commands operate by modifying the item's configuration. We can see this by using the **itemconfigure** command to look at them.

```
    .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.

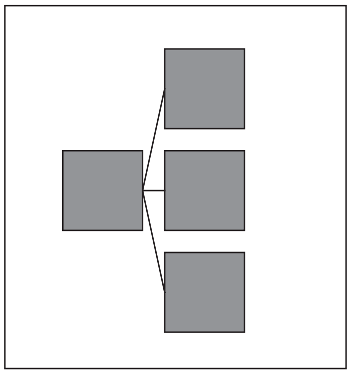
A dynamic tree is a hierarchical information space which is managed by layout objects which control the relative positioning of the pad items they are associated with. Currently, pad++ supports one default type of layout - a hierarchical tree layout which places child nodes to the right of the node.

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
```

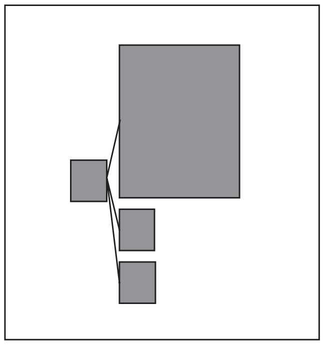
Layout

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.

Pad++ Tour

A Brief Tour Through Pad++

Jonathan Meyer, April 1997

Home Next

The Pad Metaphor

Imagine that the computer screen is a section of wall about the size of a typical bulletin board or whiteboard. Any area of this surface can then be accessed comfortably without leaving one's chair.

Imagine further that by applying extraordinarily good eyesight and eye-hand coordination, a user can both read and write as comfortably on any micron wide section of this surface as on any larger section. This would allow the full use of a surface which is several million pixels long and high, on which one can comfortably create, move, read and compare information at many different scales.

The above scenario would, if feasible, put vast quantities of information directly at the user's fingertips. For example, several million pages of text could be fit on the surface by reducing it sufficiently in scale, making any number of on-line information services, encyclopedias, etc., directly available. In practice one would arrange such a work surface hierarchically, to make things easier to find. In a collaborative environment, one could then see the layout (in miniature) of many other collaborators' surfaces at a glance.

Home Contents Prev Next

The Pad++ Metaphor (continued)

The above scenario is impossible because we can't read or write at microscopic scale. Yet the concept is very natural since it mimics the way we continually manage to find things by giving everything a physical place. A good approximation to the ideal depicted would be to provide ourselves with some sort of system of `magic magnifying glasses' through which we can read, write, or create cross-references on an indefinitely enlargeable (`zoomable') surface.

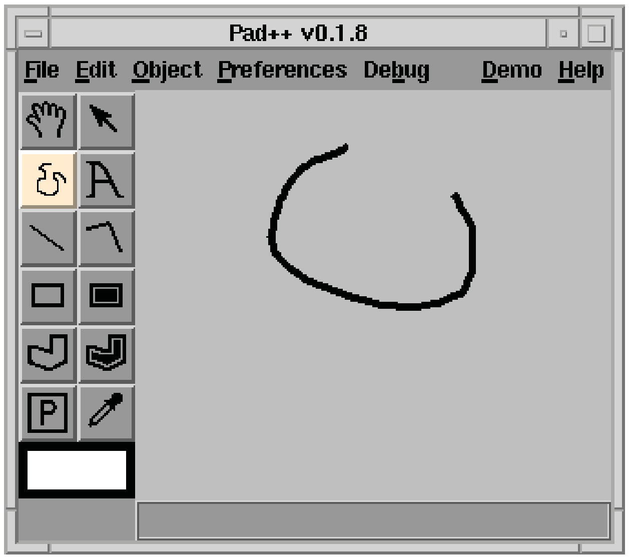
Pad++ gives users this zoomable surface. Before we look in detail at Pad++, though, lets clarify what we mean by zooming.

What exactly is meant by zooming?

To illustrate what we mean by zooming, there follows a series of snapshots of a Pad++ session which show how you might interact with a zoomable drawing application.

Click on the Next button below to read on.

Home Contents Prev Next

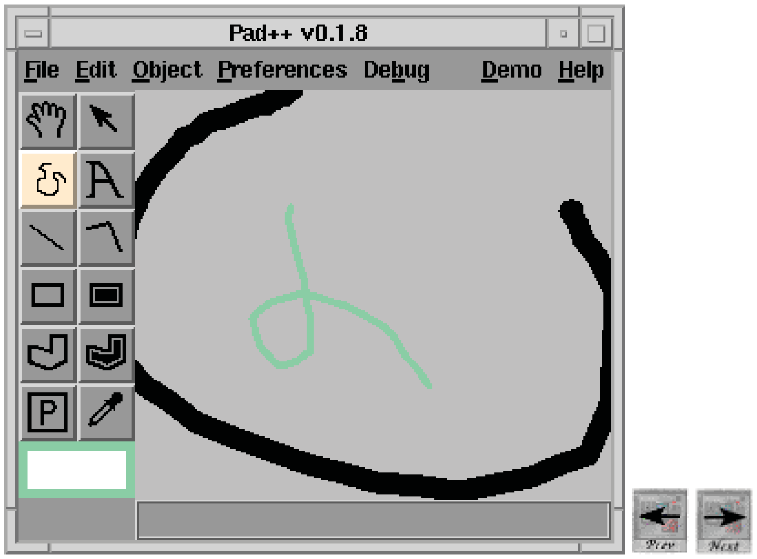


Here we see a simple shape drawn on surface of our Pad++ drawing application.

In Pad++, clicking and holding on the middle mouse button performs an smooth animated zoom in.

Your browser, unfortunately, can't show this! Instead, click on the next button above to single-step to the next image in the sequence...

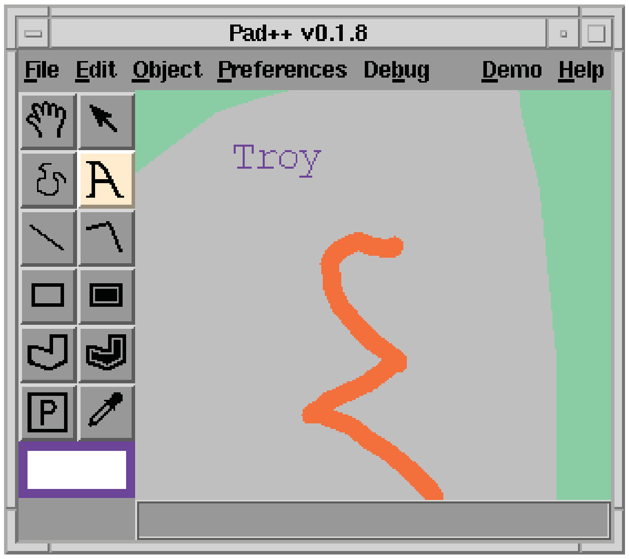
Click on the next button to go on.

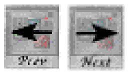


After zooming in a little, we have drawn a second, smaller shape.

This process can be repeated indefinitely...

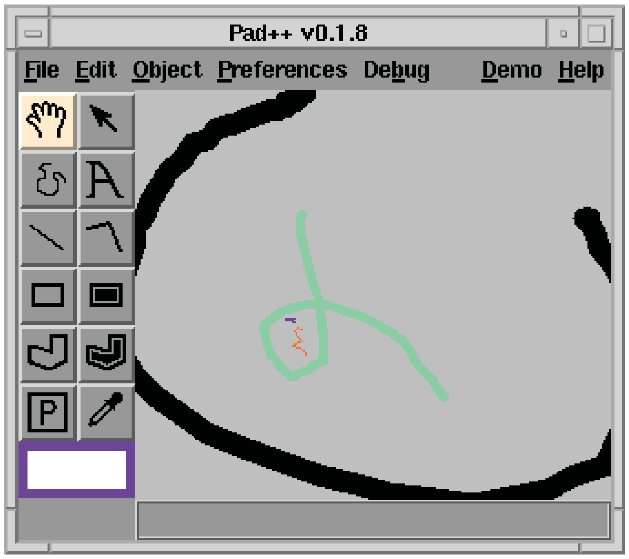
Click on the next button to go on.



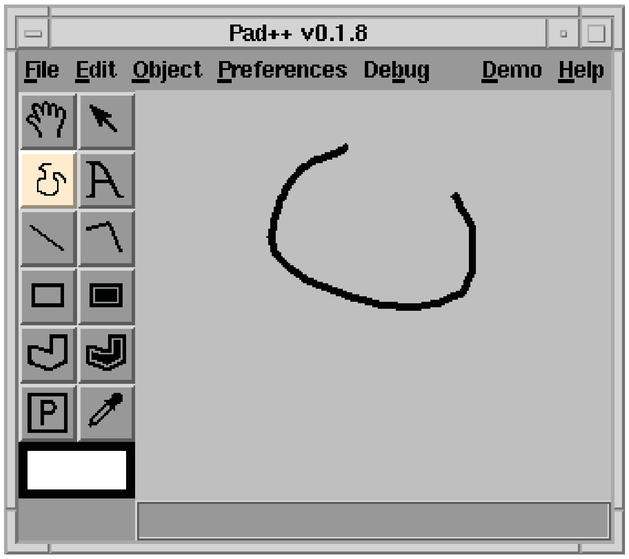


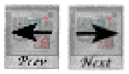
Here we have zoomed in a little more and drawn a third shape and also some text.

Click on the next button to go on.



At any point, you zoom out to see an overview of what's been done.



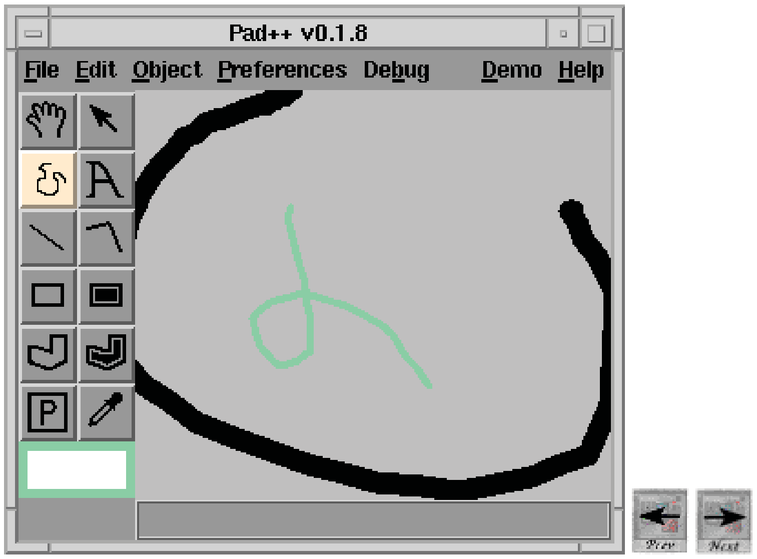


Here we see a simple shape drawn on surface of our Pad++ drawing application.

In Pad++, clicking and holding on the middle mouse button performs an smooth animated zoom in.

Your browser, unfortunately, can't show this! Instead, click on the next button above to single-step to the next image in the sequence...

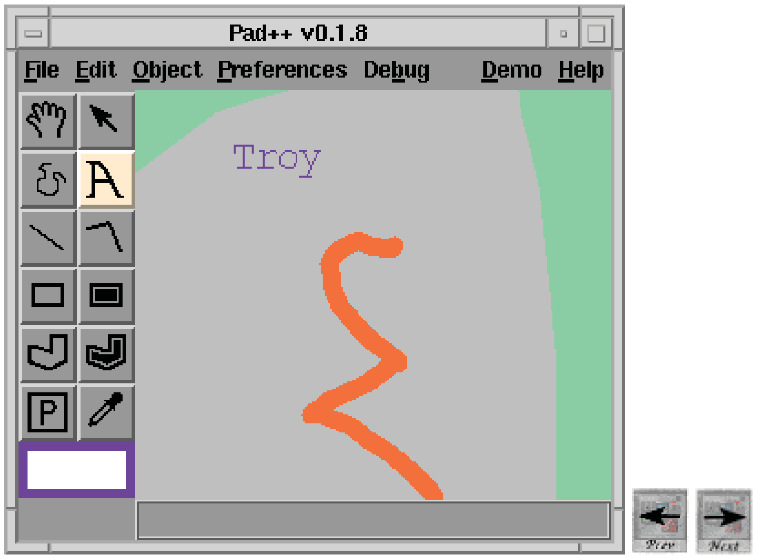
Click on the next button to go on.



After zooming in a little, we have drawn a second, smaller shape.

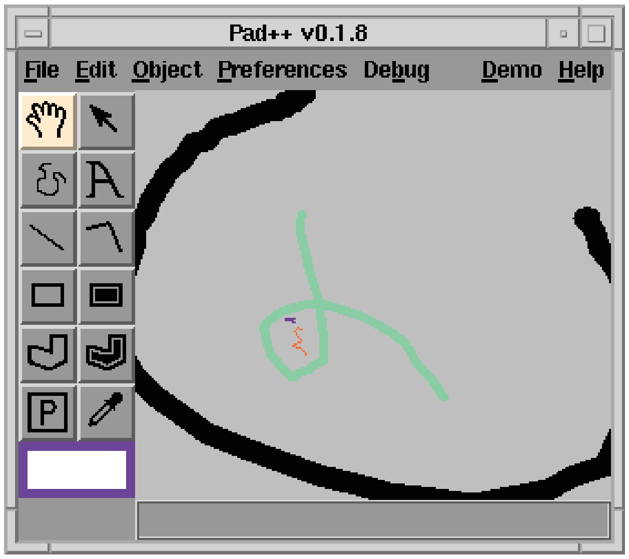
This process can be repeated indefinitely...

Click on the next button to go on.

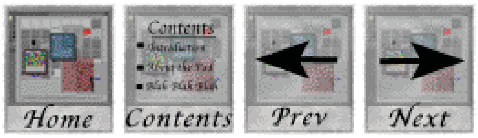


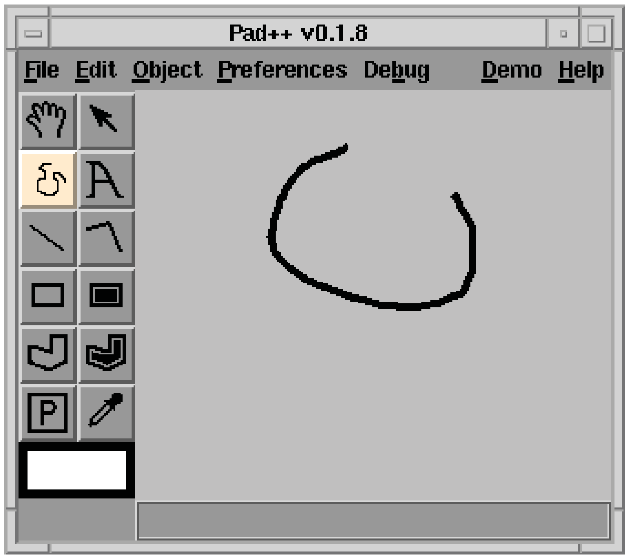
Here we have zoomed in a little more and drawn a third shape and also some text.

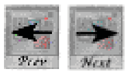
Click on the next button to go on.



At any point, you zoom out to see an overview of what's been done.





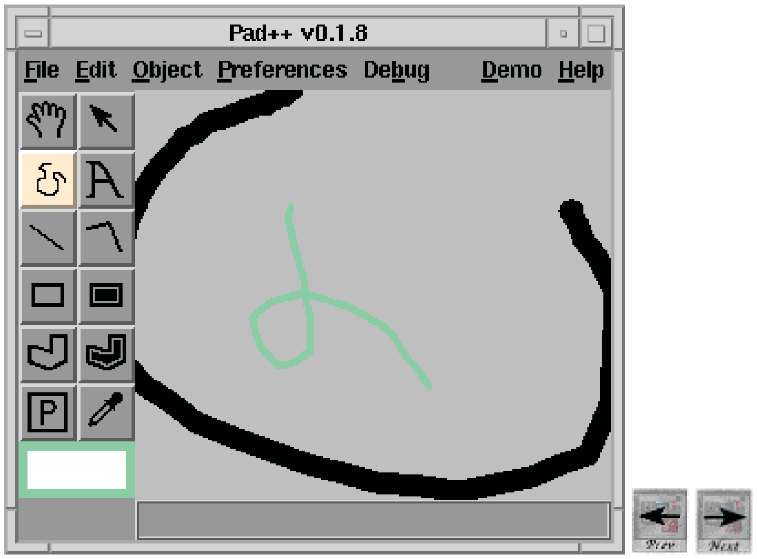


Here we see a simple shape drawn on surface of our Pad++ drawing application.

In Pad++, clicking and holding on the middle mouse button performs an smooth animated zoom in.

Your browser, unfortunately, can't show this! Instead, click on the next button above to single-step to the next image in the sequence...

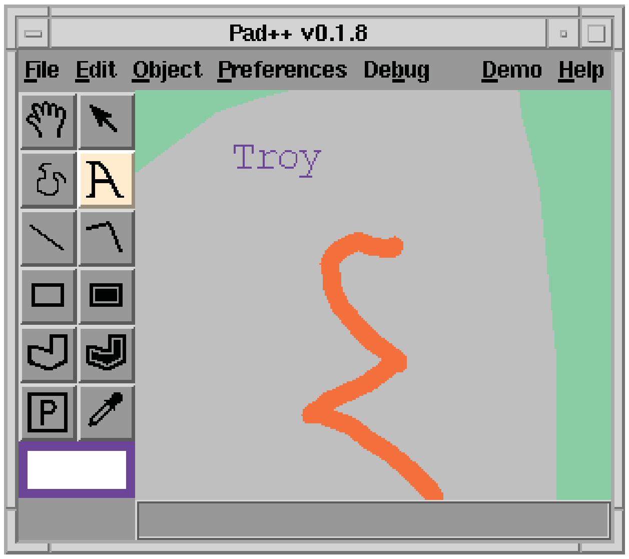
Click on the next button to go on.

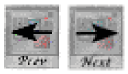


After zooming in a little, we have drawn a second, smaller shape.

This process can be repeated indefinitely...

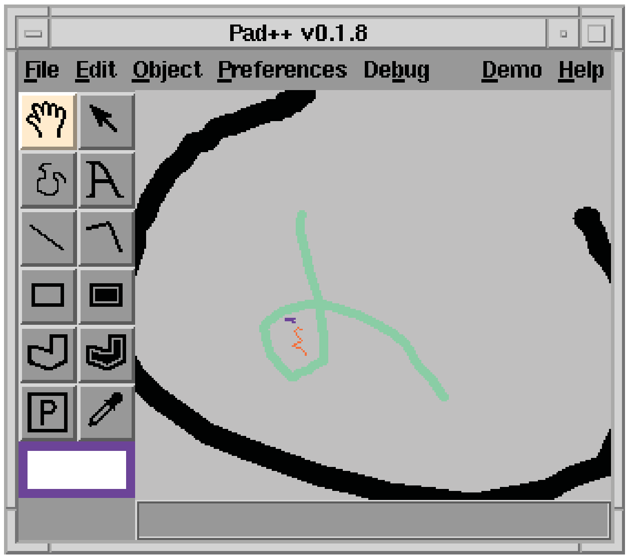
Click on the next button to go on.



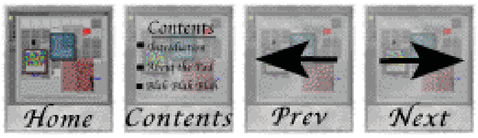


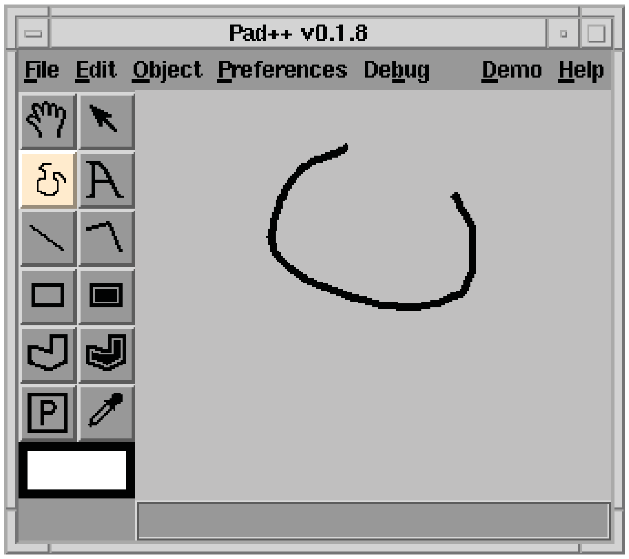
Here we have zoomed in a little more and drawn a third shape and also some text.

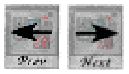
Click on the next button to go on.



At any point, you zoom out to see an overview of what's been done.





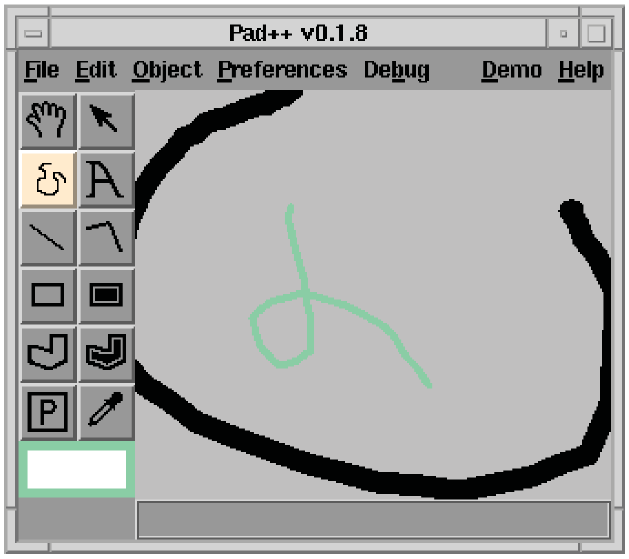


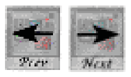
Here we see a simple shape drawn on surface of our Pad++ drawing application.

In Pad++, clicking and holding on the middle mouse button performs an smooth animated zoom in.

Your browser, unfortunately, can't show this! Instead, click on the next button above to single-step to the next image in the sequence...

Click on the next button to go on.

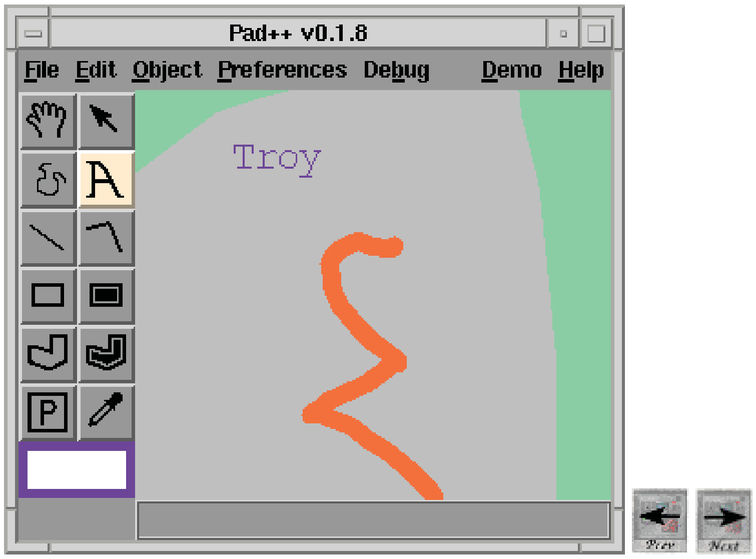




After zooming in a little, we have drawn a second, smaller shape.

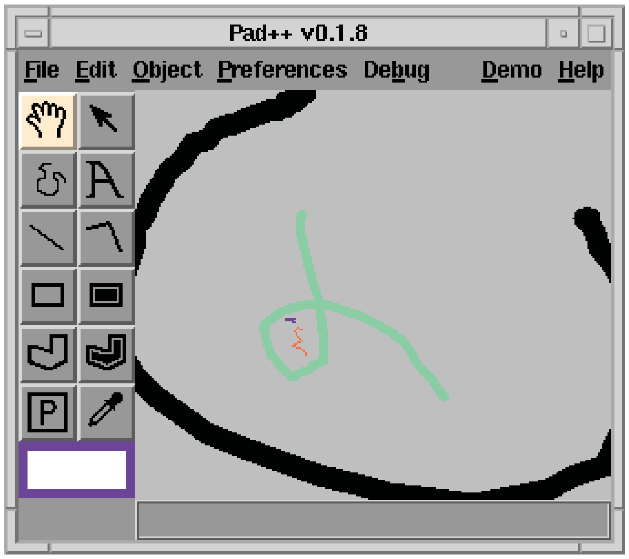
This process can be repeated indefinitely...

Click on the next button to go on.

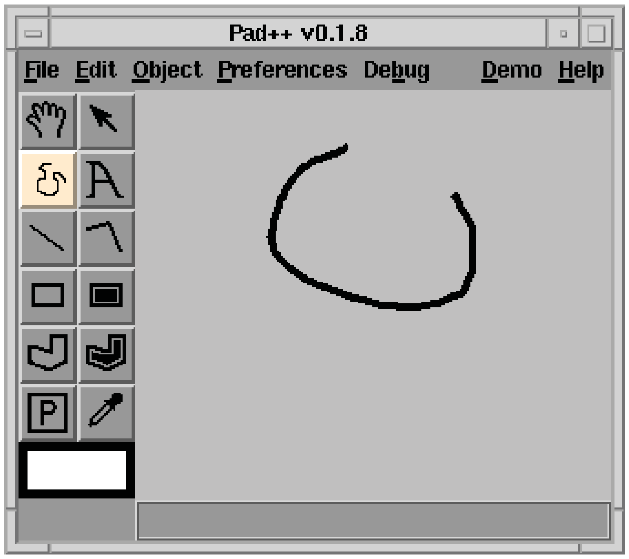


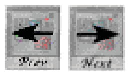
Here we have zoomed in a little more and drawn a third shape and also some text.

Click on the next button to go on.



At any point, you zoom out to see an overview of what's been done.



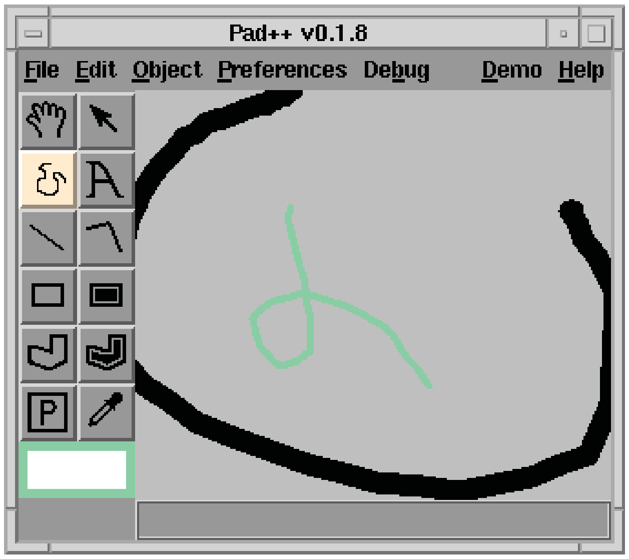


Here we see a simple shape drawn on surface of our Pad++ drawing application.

In Pad++, clicking and holding on the middle mouse button performs an smooth animated zoom in.

Your browser, unfortunately, can't show this! Instead, click on the next button above to single-step to the next image in the sequence...

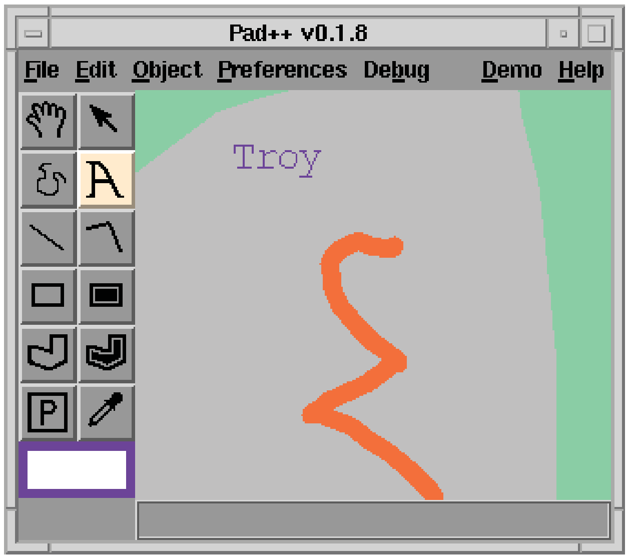
Click on the next button to go on.

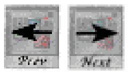


After zooming in a little, we have drawn a second, smaller shape.

This process can be repeated indefinitely...

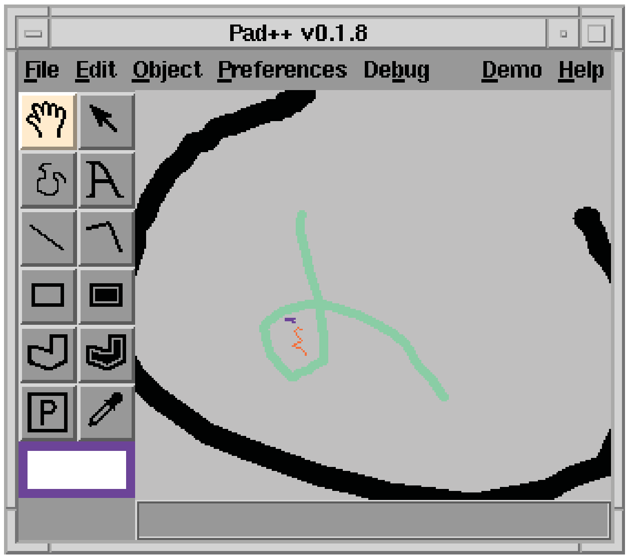
Click on the next button to go on.



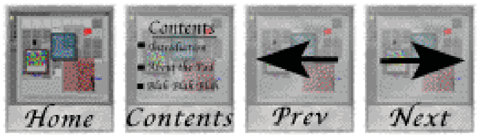


Here we have zoomed in a little more and drawn a third shape and also some text.

Click on the next button to go on.



At any point, you zoom out to see an overview of what's been done.



Why zoom? Three example applications

Once you add zooming to a user interface, new kinds of interactions are possible. The following sections introduce three sample applications that we have developed which use zooming.

1. An outline viewer

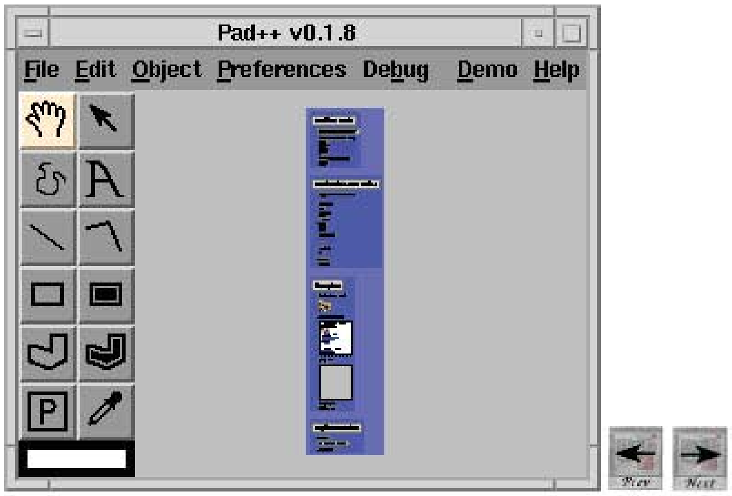
The Pan/Zoom space is a powerful visualisation for any kind of hierarchical data - nested details can be shown at a smaller scale, conveying the relations between objects very naturally.

For example, we have developed an interactive outline viewer which we use for giving presentations. The outline viewer shows the text of the whole presentation - you simply zoom in on the points you want to make.

Main topic headings are shown at a large scale, subheadings are shown at a smaller scale, and individual points are shown even smaller.

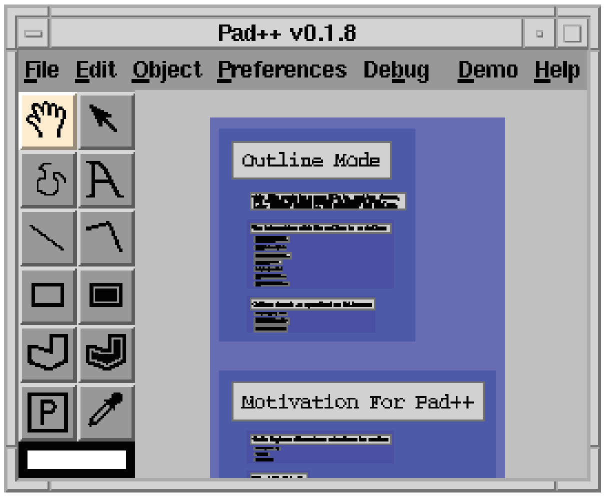
Click on the Next button below to read on.

Home Contents Prev Next



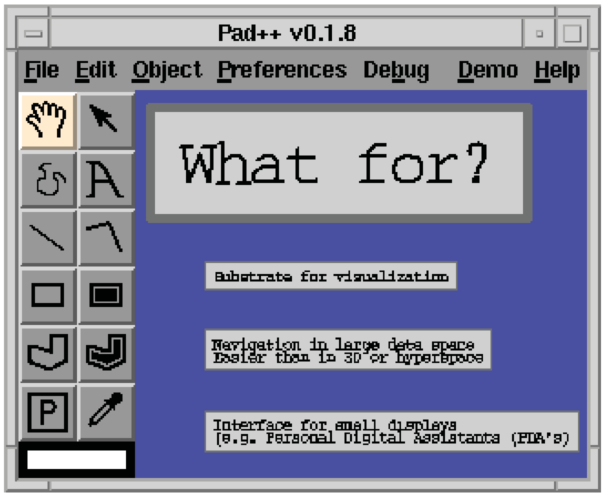
For example, here we see a distant view of an outline for a presentation about Pad++ itself.

Click on the next button to go on.



When you zoom in, the major headings become legible.

Click on the next button to go on.



What for?

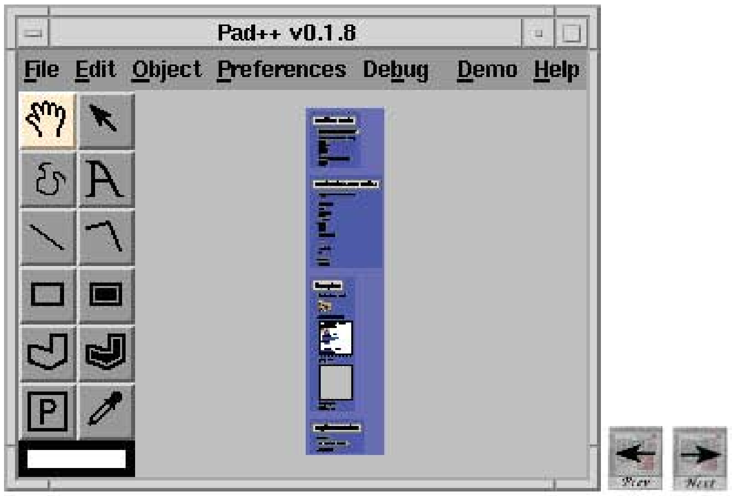
Substrate for visualization

Navigation in large data space
Easier than in 3D or hyperspace

Interface for small displays
(e.g. Personal Digital Assistants (PDA's)

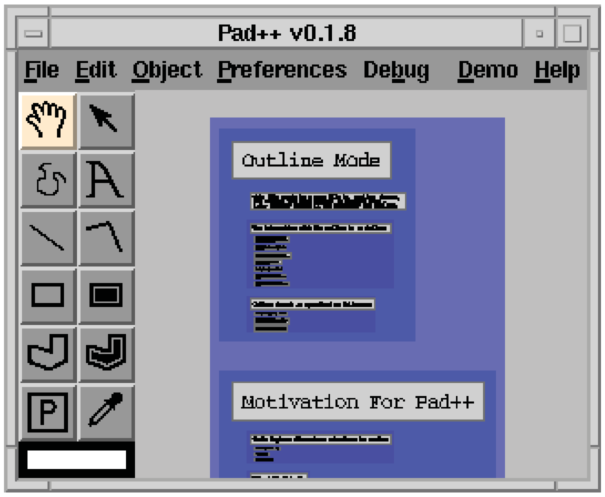
When you click on a piece of text, Pad++ zooms in so that that text fills the screen horizontally and appears near the top of the screen vertically.

Home Contents Prev Next



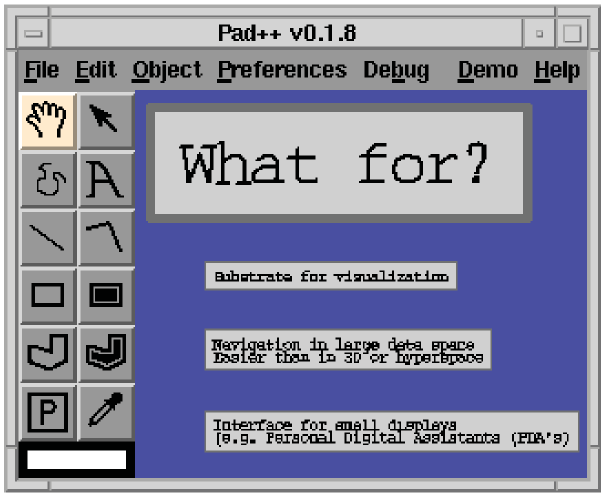
For example, here we see a distant view of an outline for a presentation about Pad++ itself.

Click on the next button to go on.



When you zoom in, the major headings become legible.

Click on the next button to go on.



What for?

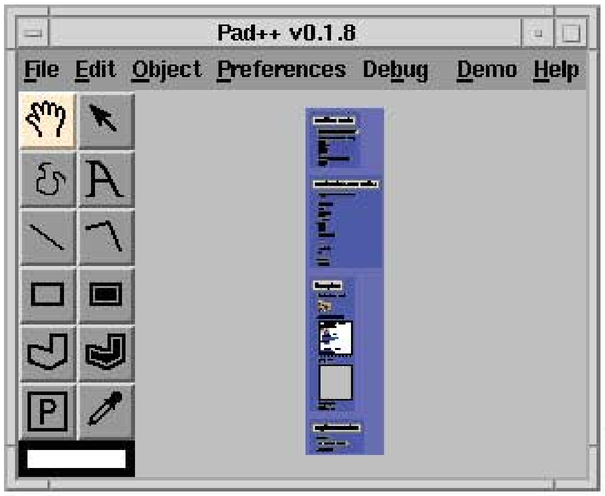
Substrate for visualization

Navigation in large data space
Easier than in 3D or hyperspace

Interface for small displays
(e.g. Personal Digital Assistants (PDA's)

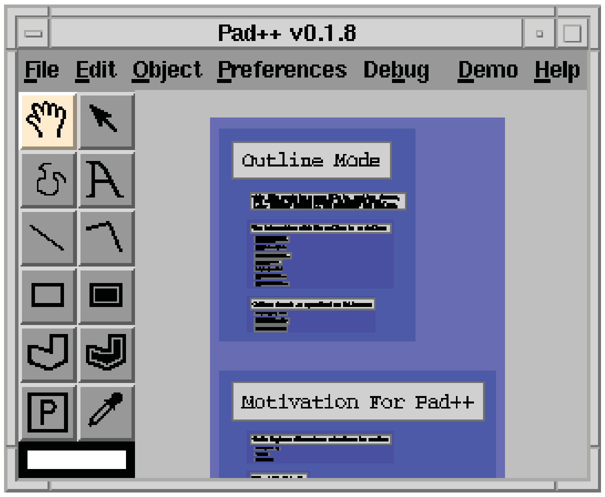
When you click on a piece of text, Pad++ zooms in so that that text fills the screen horizontally and appears near the top of the screen vertically.

Home Contents Prev Next



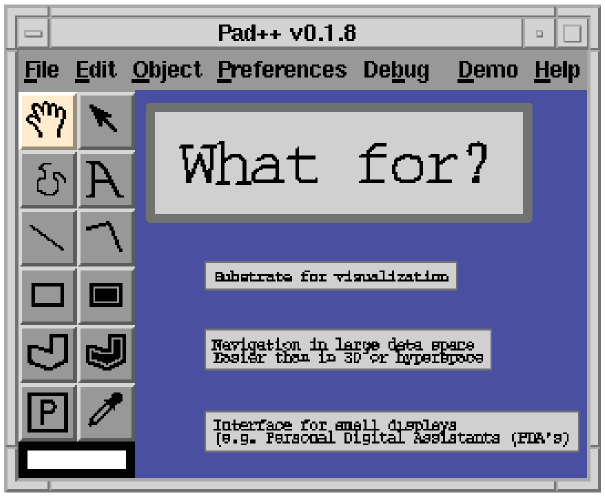
For example, here we see a distant view of an outline for a presentation about Pad++ itself.

Click on the next button to go on.



When you zoom in, the major headings become legible.

Click on the next button to go on.



What for?

Substrate for visualization

Navigation in large data space
Easier than in 3D or hyperspace

Interface for small displays
(e.g. Personal Digital Assistants (PDA's)

When you click on a piece of text, Pad++ zooms in so that that text fills the screen horizontally and appears near the top of the screen vertically.

Home Contents Prev Next

Interaction with the outline view

To move through a presentation shown in the outline view, you can simply click on successive pieces of text.

Alternatively, the 'n' and the 'p' keys move to the next or the previous item in the presentation using smooth animated zooms.

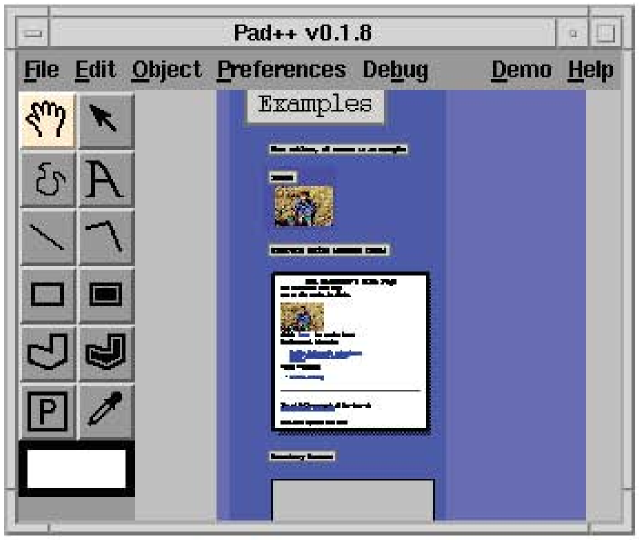
Of course, at any time you can use the mouse to pan and zoom to another part of the presentation - great if you want to quickly skip a few sections or go back to a previous section.

Click on the Next button below to read on.

Home Contents Prev Next

Hypermedia presentations using outlines

Outline documents can include things other than text: images and interactive objects can be incorporated in the outline; you could even include a whole document nested within the outline, drawn at a smaller scale - here you see an HTML document and an image nested within the outline:



2. A file browser

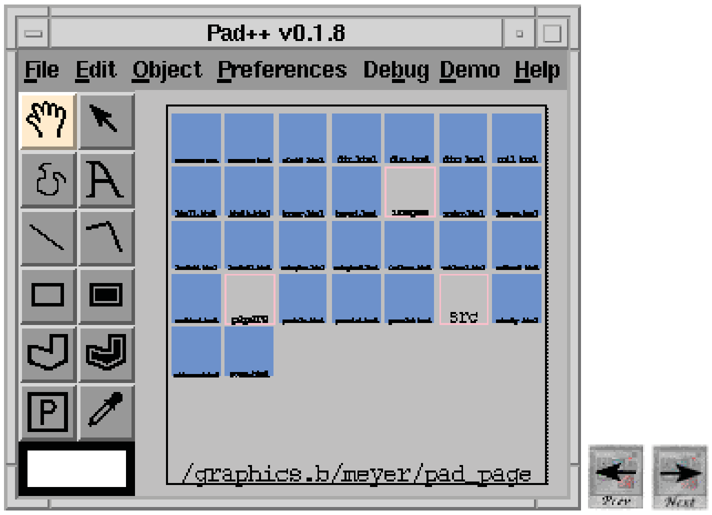
Many hierarchical structures are suitable for visualisation within Pad++. The second application we will look at is a file browser.

An important concept in the zoomable coordinate system is that objects have an absolute physical location. The metaphor is not one of viewing a small part of the data through a window, as in MS-Windows. Instead, all of the data is placed directly on the Pad++ surface, and you navigate through the data by panning and zooming. In this sense, Pad++ is a 2.5 dimensional virtual reality.

For example, in the Pad++ file browser, each file is shown as a small rectangle with a label.

Click on the Next button below to read on.

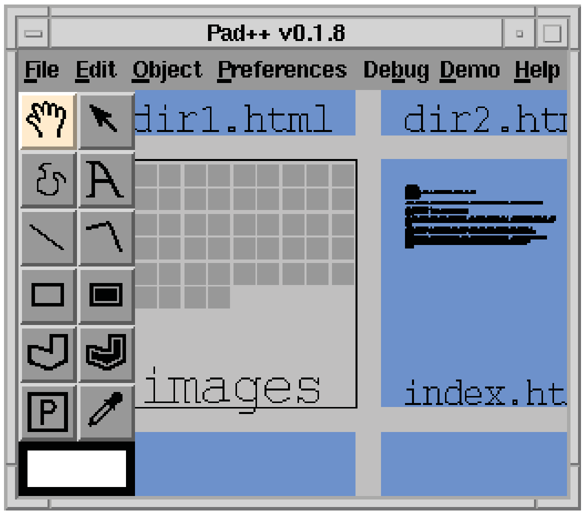
Home Contents Prev Next


Pad++ v0.1.8
File Edit Object Preferences Debug Demo Help
src
/graphics.b/meyer/pad_page
Prev
Next

Above you see the file browser looking at some of the files that constitute this document.

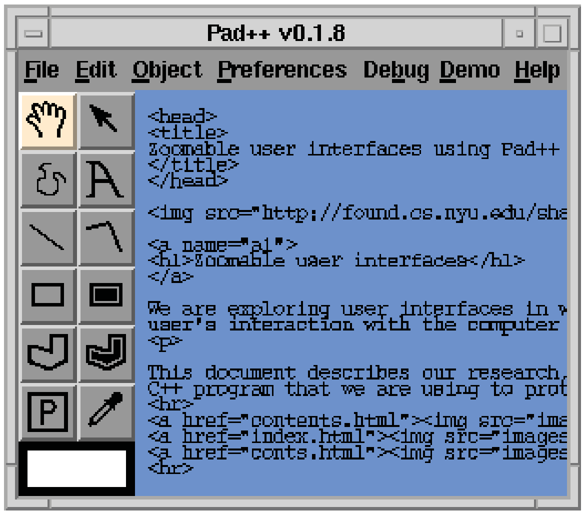
Colors are used to denote file type - text documents are shown using a solid blue rectangle, whereas the 'images' folder is shown as an unfilled rectangle outline.

Click on the next button to go on.

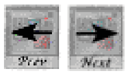


To see more detail, simply zoom in. Here, we have zoomed in for a closer view of the 'images' folder and the 'begin.html' document. At this distance, you can't make out individual pictures or words, but you can see the outlines of files within the 'images' folder and the shape of the text in the HTML file.

Click on the next button to go on.

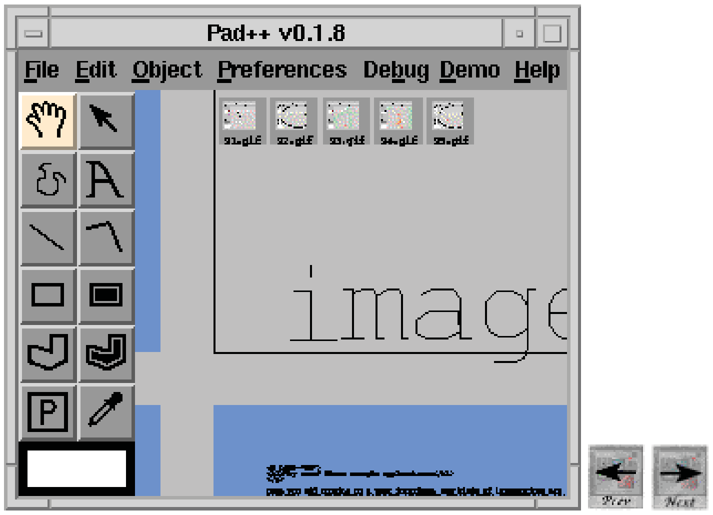


```
<head>
<title>
Zoomable user interfaces using Pad++
</title>
</head>
<img src="http://found.cs.nyu.edu/sha
<a name="a1">
<h1>Zoomable user interfaces</h1>
</a>
We are exploring user interfaces in w
user's interaction with the computer
<p>
This document describes our research,
C++ program that we are using to prot
<hr>
<a href="contents.html"><img src="ima
<a href="index.html"><img src="images
<a href="conts.html"><img src="images
<hr>
```



To read the contents of the 'begin.html' file, zoom in further.

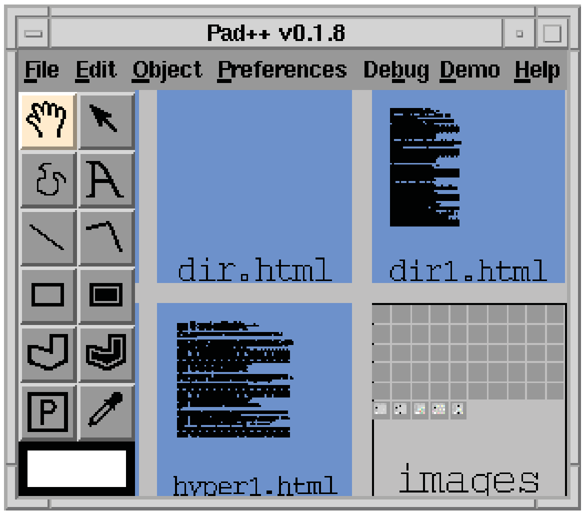
Click on the next button to go on.



Now we have zoomed out and panned across.

You can see the pictures within the 'images' folder, and some of the contents of another file. Of course you could zoom in further to see the images in greater detail.

Click on the next button to go on.



Zoom out again. The file browser performs 'lazy loading'; it only loads the contents of the files that you zoom in on. When you zoom out again, you can see which documents you have seen and which are still unloaded.

Home Contents Prev Next

Semantic zooming in the file browser

The file browser illustrates what we call 'semantic zooming' - objects can render themselves differently when viewed at different sizes. So when a file is very small, it is shown only as a rectangle. A little larger and the label appears. Larger still and the contents are visible.

Through semantic zooming, many of the mode switches required in traditional windowing systems can be removed. The user does not have to double click on a file icon to switch into a mode where its contents are visible - instead they learn and reuse the simple principal that, to see more detail, you look more closely.

Lets move on to the third application in Pad++...

Click on the Next button below to read on.

Home Contents Prev Next

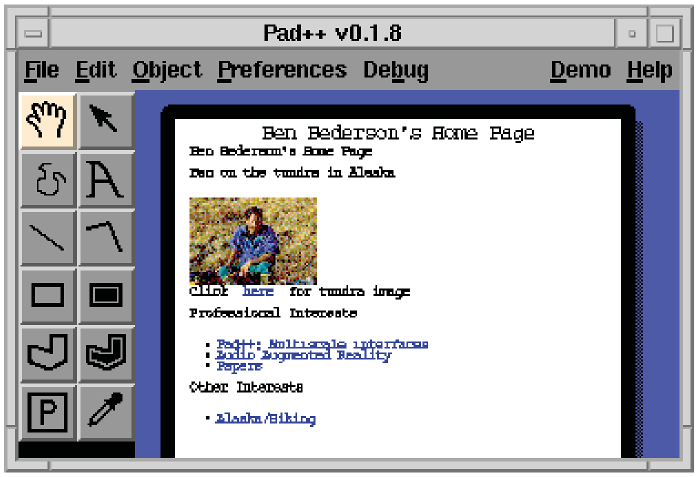
3. An HTML browser

A zooming version of Mosaic and Netscape? This is not such a strange idea. The third example Pad++ application we will look at is a simple Pad++ web browser.

Although the web browser we show here is still fairly primitive, you could imagine a much more advanced version. Does the idea of using a zoomable Web browser excite you? If so, why not email us at *pad-comment@cs.unm.edu* and tell us your thoughts.

Click on the Next button below to read on.

Home Contents Prev Next



Ben Bederson's Home Page

Ben Bederson's Home Page

Ben on the tundra in Alaska

Click here for tundra image

Professional Interests

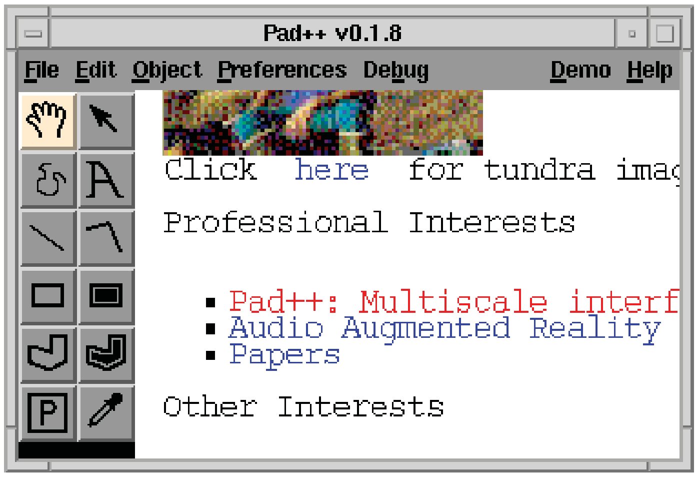
- Pad++: Multiscale interfaces
- Audio Augmented Reality
- Papers

Other Interests

- Alaska/Biking

Here is a screen snapshot showing Pad++ displaying an HTML document.

Click on the next button to go on.



Click here for tundra imag

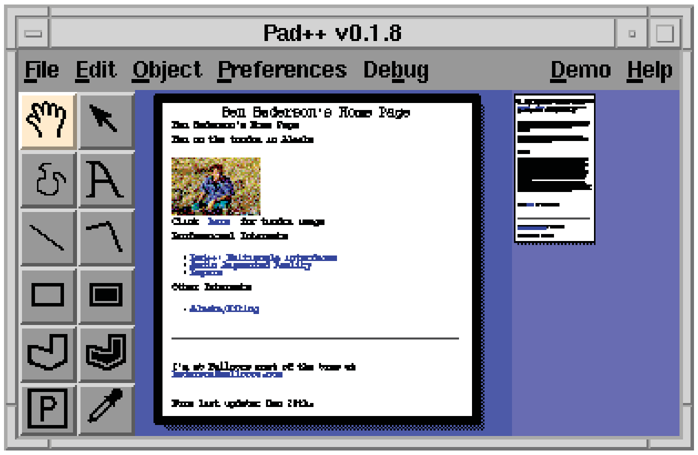
Professional Interests

- Pad++: Multiscale interf
- Audio Augmented Reality
- Papers

Other Interests

Here is a zoomed in view of the document. Hotwords are shown in blue - positioning the pointer over a hotword changes its color to red. In this snapshot the pointer was over the 'Pad++: Multiscale interfaces' link.

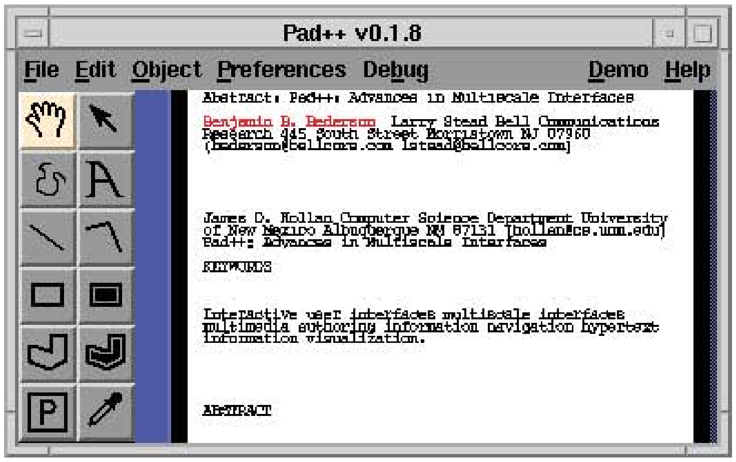
Click on the next button to go on.



When you follow a link, the relevant document is loaded into Pad++ and placed on the surface to the right of the original document, at a smaller scale. Here you can see the 'Pad++: Multiscale interfaces' document loaded beside the home page.

The Pad++ HTML browser will lay out sub-documents in two columns next to the parent document. Because Pad++ is zoomable, there is always enough space between those two columns for placing further documents reached from those sub-documents!

Click on the next button to go on.



Abstract: Pad++: Advances in Multiscale Interfaces

Benjamin B. Bederson Larry Stead Bell Communications Research 445 South Street Morristown NJ 07960 (bederson@bellcore.com lstead@bellcore.com]

James D. Hollan Computer Science Department University of New Mexico Albuquerque NM 87131 [hollan@cs.unm.edu] Pad++: Advances in Multiscale Interfaces

KEYWORDS

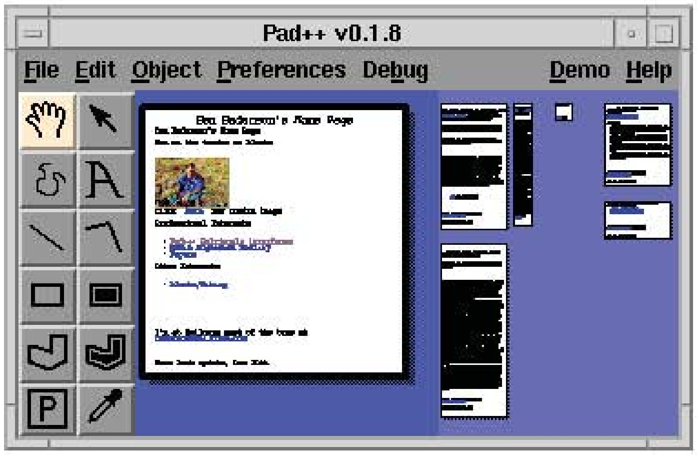
Interactive user interfaces multiscale interfaces multimedia authoring information navigation hypertext information visualization.

ABSTRACT

Simply zoom in on the sub-document to read it.

Preserving history in the HTML browser

You can see the history of the user's interaction implicitly in the layout of the documents on the Pad++ surface. Here's what the Pad++ surface looks like after we've done a little browsing. Four documents were accessed via Ben's home page. From the first of these, another two documents were visited. Just zoom in to view the documents.



Home Contents Prev Next

Other Pad++ features

We have seen three example Pad++ applications. Now we will look at some other Pad++ features: portals, hyperlinks and magic lenses.

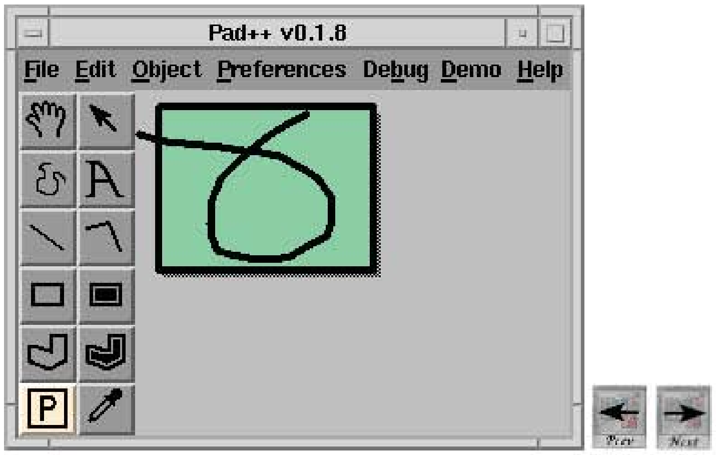
Portals - twists in Pad++ space

The Pad++ surface as an infinite zoomable space, with objects residing at specific *x, y, zoom* coordinates within that space.

Sometimes, however, you want to be able to compare two distant locations, or to look at a zoomed out view of a document together with a close up view.

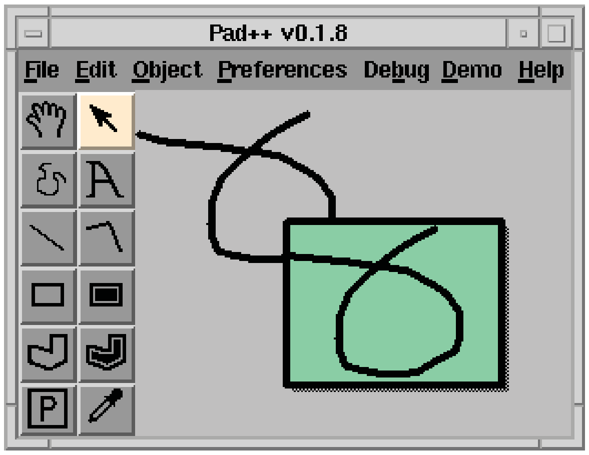
Portals allow you to do this - they introduce 'holes' in the Pad++ surface through which you look onto other regions of the Pad++ surface, or in fact onto other Pad++ surfaces.

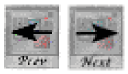
Home Contents Prev Next



When you initially create a portal it looks like a colored rectangle with a drop shadow. Here we have created a portal over a squiggle on the Pad++ surface.

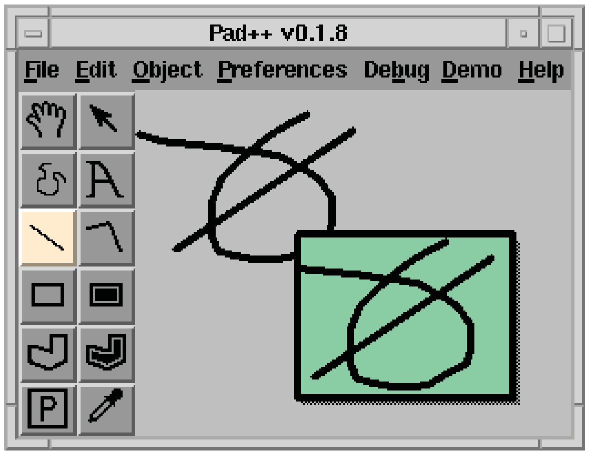
Click on the next button to go on.

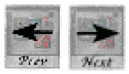




When you select the portal and move it away, it drags off a copy of what was underneath it.

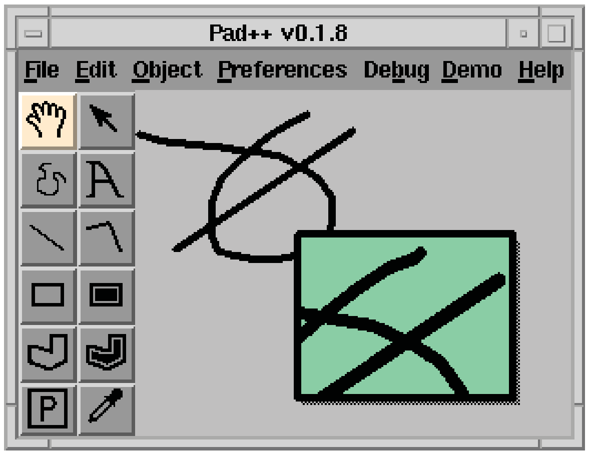
Click on the next button to go on.





In fact, however, it isn't a copy - its the 'real thing' - its another view onto the same part of the Pad++ surface. So when you draw a second squiggle next to the first, it appears in two places... on the Pad++ surface, and within the portal's view of the Pad++ surface.

Click on the next button to go on.

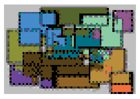


Each portal has its own independant view - you can pan and zoom inside a portal without changing the view outside the portal. Here we see the portal looking at a zoomed in view of the squiggle and line.

Portals vs Windows

A portal is not like a window, which represents a dedicated link between a section of screen and a specific thing (eg: a Unix shell in X-Windows or a directory in the Macintosh Finder). A portal is, rather, a view onto the infinite Pad++ surface; links to specific items are established and broken continually as the portal's view changes.

Also, unlike windows, portals can recursively look onto other portals - you can get some pretty interesting effects with portals:



We will discover more about portals later. First, lets look at hyperlinks.

Click on the Next button below to read on.

Home Contents Prev Next

Hyperlinks

We have seen that, when you move the pointer over text within the Pad++ outliner, the text changes color to red; clicking on the text causes Pad++ to pan and zoom until that text fills the screen horizontally and appears near the top of the screen vertically. The text is acting as a hyperlink to a location on the Pad++ surface.

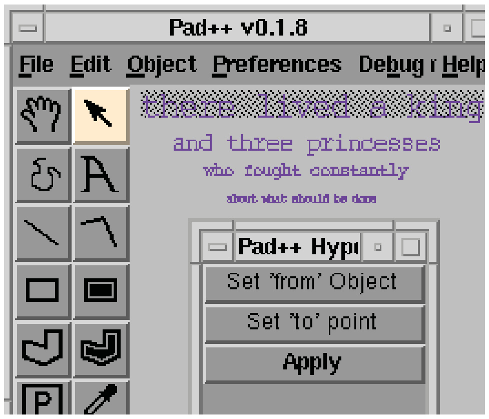
(Similarly, in the Pad++ HTML browser, some words are shown in blue. They are 'hotwords' - positioning the pointer over the word causes it to change color to red. When you click on the word, the appropriate document is placed on the Pad++ surface.)

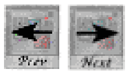
In Pad++, this concept is generalised. Any object an be turned into a hyperlink.

For example, suppose I am writing a zoomable narrative, and I want to make the text in the story active, so that users can click on lines of text to progress through the story. Read on to discover how to do this.

Click on the Next button below to read on.

Home Contents Prev Next

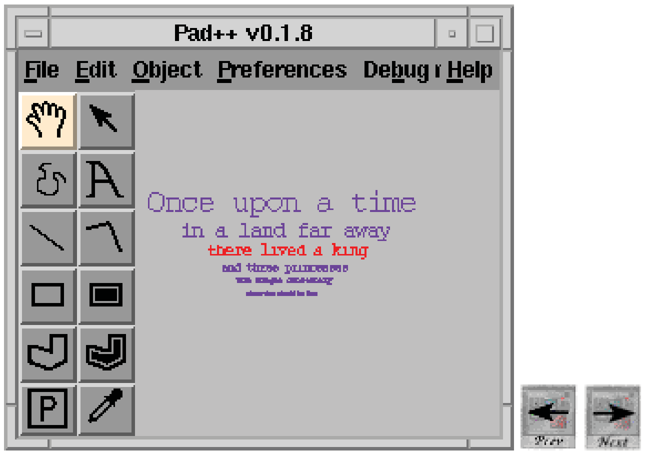




In this snapshot we are making the 'there lived a king' line of text a hyperlink.

From within the Pad++ draw application, bring up the 'Hyperlink' panel. Then set the 'from object' (the object you wish to make active), and the 'to point' (the place you want the hyperlink to take you to), then click on apply.

Click on the next button to go on.

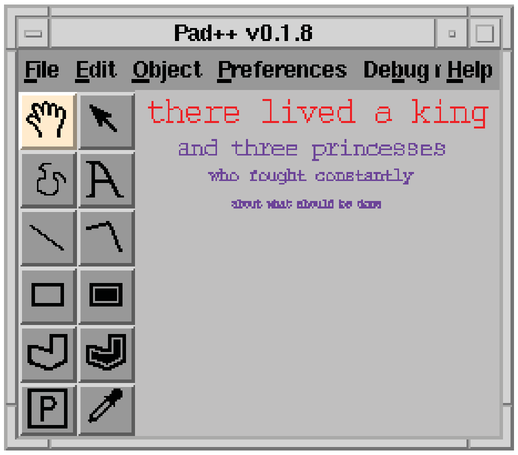


Once upon a time
in a land far away
there lived a king
and three princesses
[illegible]
[illegible]

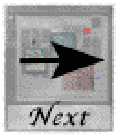
The text is now active. Here we have zoomed out and you can see the whole story.

The pointer is positioned over the 'there lived a king' line, and it has changed color to red to indicate that it is a hyperlink.

Click on the next button to go on.

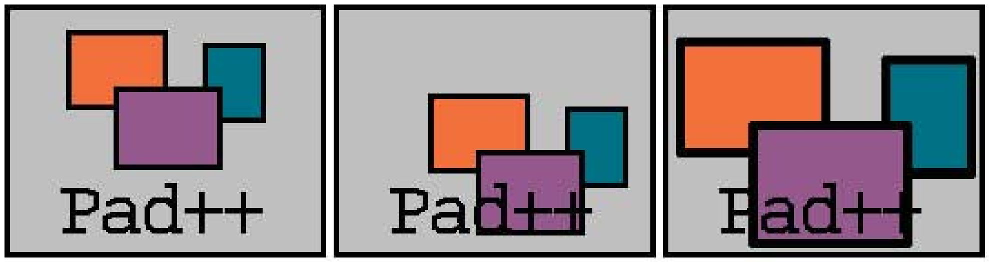


Clicking on the 'there lived a king' line performs an animated pan/zoom to the location that the author specified.



Sticky objects

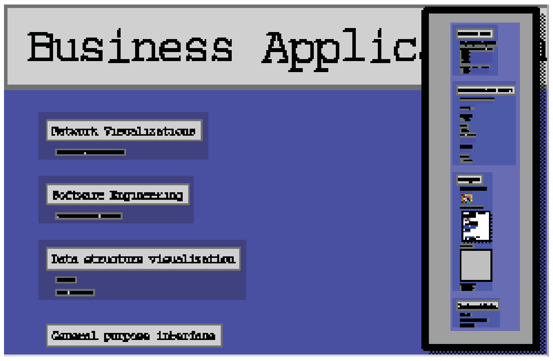
Pad++ objects can be made 'sticky' - so that they stick to the surface of the glass, rather than being on the Pad++ surface. When you pan and zoom the Pad++ surface, sticky object remain stationary. In these snapshots, the 'Pad++' label has been made sticky, and does not pan and zoom with other objects:



Sticky portals as indexes

By making a portal sticky and changing its view so that it looks at a hotword or an active document, you create automatic hyper-buttons and indexes which float above the Pad++ surface.

Here you can see an example of a portal acting as an index to the Pad++ outline: Clicking on text within the portal causes the main Pad++ surface to pan/zoom to show the text you clicked on, giving you rapid access to the outline.



Click on the Next button below to read on.

Portal filtering and 'magic lenses'

Portals can be used to filter what is seen through them. This concept goes by the name of 'Portal Filters' or 'Magic Lenses'.

When an object is seen through a portal, the portal can communicate with the object during rendering to change how the object appears.

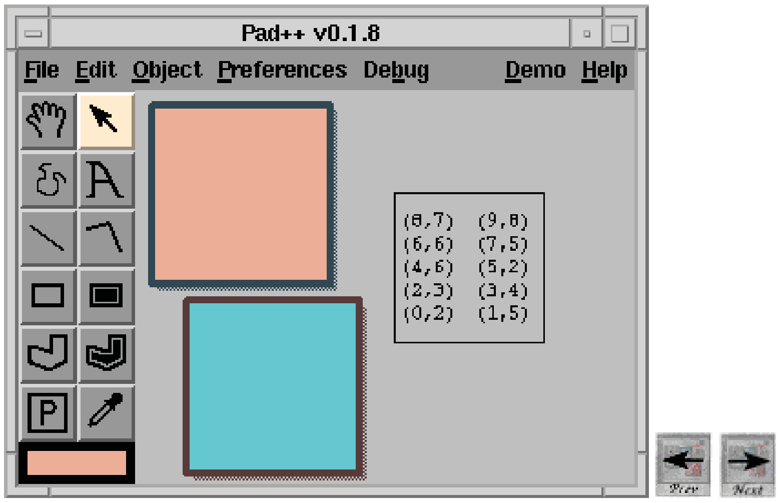
The portal can also filter user events (mouse clicks, etc.) which pass through it - either passing these events on unchanged, blocking them, or communicating with the object to decide how to interpret the event. So objects can behave differently when seen through portals.

By using these features, developers can create portals which act as magic lenses - lenses which change the nature of the data in unique ways. (the concept of portal filters and magic lenses was developed independently by Ken Perlin and David Fox in the original version of Pad and by Eric Beir, et. al. at Xerox PARC).

Lets look at some examples.

Click on the Next button below to read on.

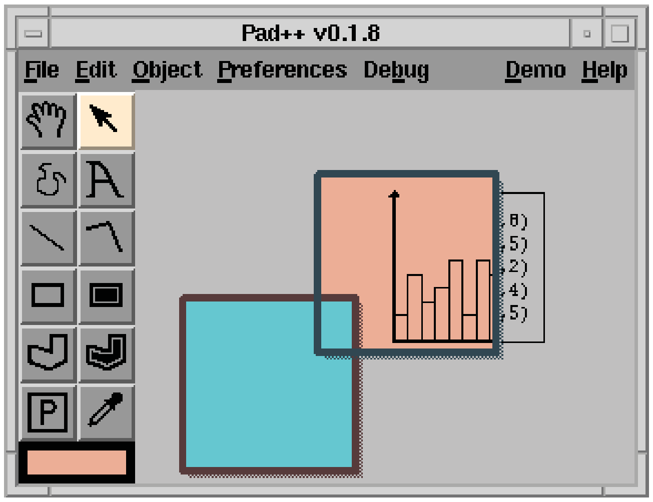
Home Contents Prev Next



(8,7)	(9,8)
(6,6)	(7,5)
(4,6)	(5,2)
(2,3)	(3,4)
(0,2)	(1,5)

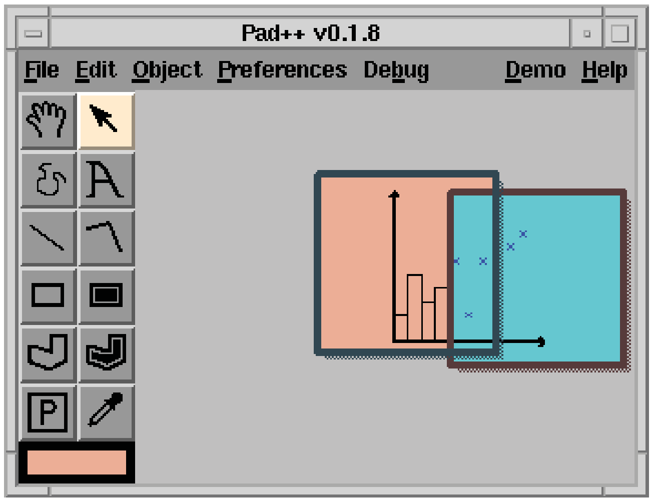
In the image above you can see a table of numbers and two portals.

Click on the next button to go on.



When you slide the orange portal over the table of numbers, it modifies how the table appears so that it uses a bar chart view.

Click on the next button to go on.

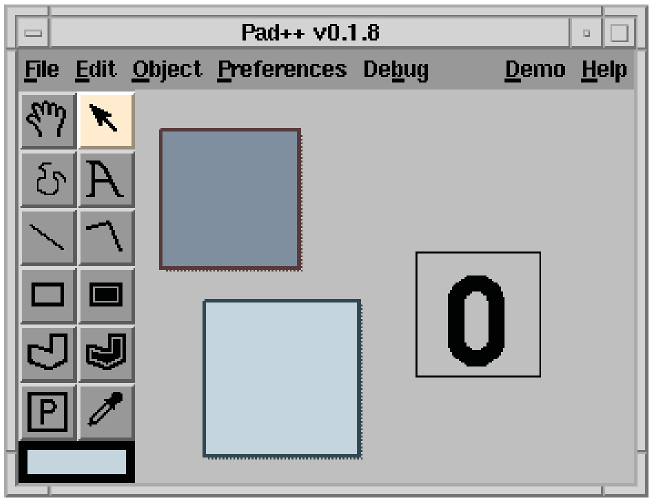


Slide the blue portal in place, and you see the table of numbers as a scatter plot.

Lenses such as these could make a good teaching aid, since they show students alternate views of the same data.

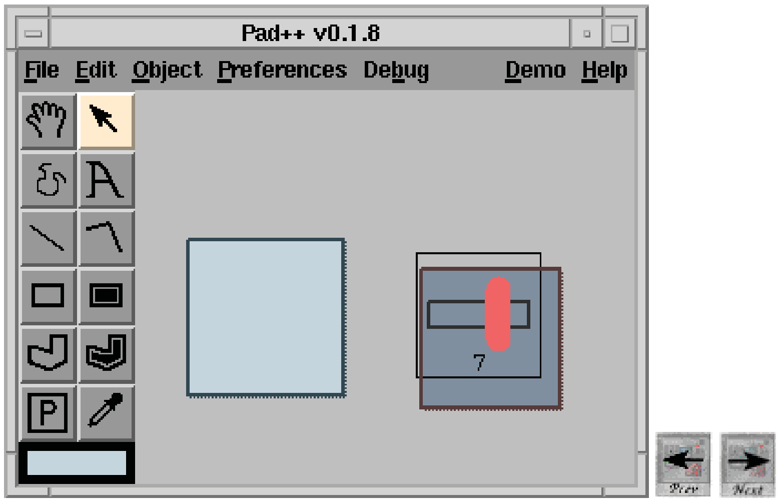
Lets see another example.

Click on the next button to go on.



In this example, the 0 digit shown above is a simple digital counter. When you position the pointer over the number and hit a number key between 0 and 9, the counter changes to show that number.

Click on the next button to go on.

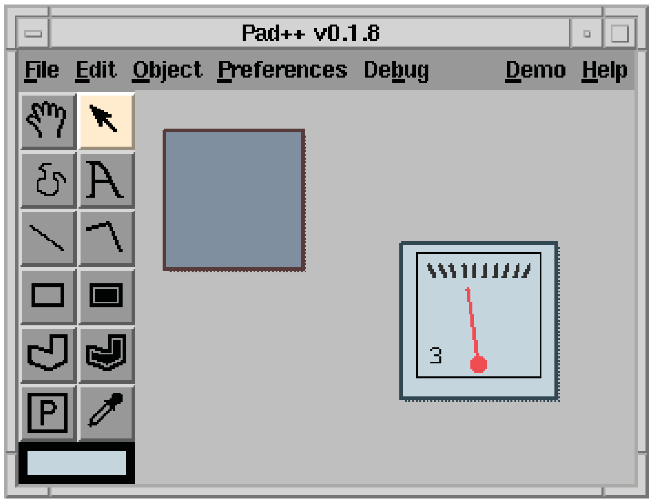


However, some people prefer to use a slider to set numeric values.

Here, the dark gray lens has been placed over the number, and the user has clicked and dragged the slider handle to change the number to 7.

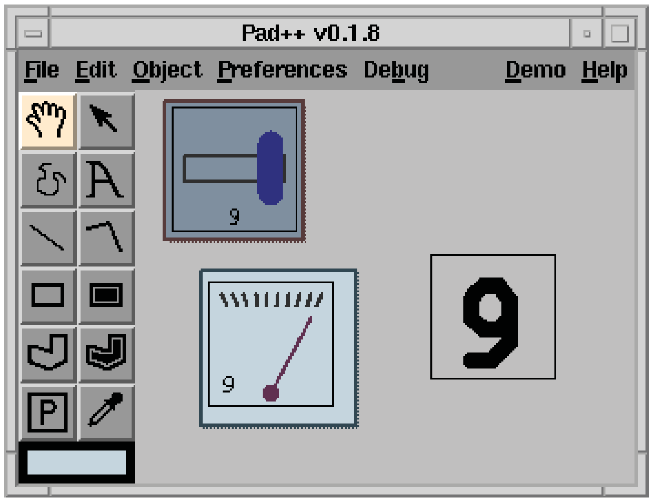
This shows how portals can change how an object behaves as well as how it appears.

Click on the next button to go on.



You may prefer to see the number as a guage. Use the light gray lens. Here we are setting the number to 3.

Click on the next button to go on.



With a little simple trickery, all three representations can be used at the same time - you can choose which version to interact with.

Home Contents Prev Next

Other uses for lenses

Whilst the example shown here are relatively trivial, you could create a much more complex lens.

Lenses are used for rendering and interacting with data in specialised ways. You could for example create a translation lens, which shows a piece of text in an alternate language.

More sophisticated lenses might be used for editing text, or for painting, or for modifying bitmap data. Instead of constantly switching between large applications and using the Import/Export options to move data between the applications, you simply pick up the relevant lens and slide it into place.

Click on the Next button below to read on.

Home Contents Prev Next

Conclusion

In Pad++ there is a sense of peripheral awareness. When objects are small or off to one side, you can still see them, and still make out a little of their contents, though they are not shown in all their detail. When you look directly at them and zoom in, all of the detail becomes available. Hence the Pad++ metaphor offers a new route for tapping into our natural spatial and geographic ways of thinking.

This metaphor is more powerful than the tradional (and now dated) view of a computer screen as a collection of small windows through which you peer at your data.

Click on the Next button below if you want to read answers to frequently asked questions about Pad++.

Home Contents Prev Next

Frequently asked questions

This document contains a list of answers to common questions about Pad++ and Zoomable User Interfaces.

Questions about the Pad++ consortium

Where did Pad++ come from?

Ken Perlin at New York University came up with the initial zoomable surface concept. He and David Fox implemented the first versions of Pad (described in the 1993 SIGGRAPH paper available here), which were the precursors to the current implementation, Pad++. Perlin and Fox's original work was supported in part originally by NYNEX, and then NSF.

Pad++ was designed by Ben Bederson and Jim Hollan while at Bellcore, and implemented by Ben Bederson. They both moved to the University of New Mexico where they developed the bulk of Pad++ in collaboration with Jon Meyer and Ken Perlin at NYU.

Now, Ben Bederson has moved to the University of Maryland, College Park where he is continuing responsibility for the further development of Pad++. Jim Hollan moved to the University of California, San Diego where he is building applications using Pad++.

Who is working on Pad++?

Pad++ research and development is supported in part by DARPA grant #N660011-94-C-6039. Work is being carried out at the University of Maryland, University of California, San Diego, and at the NYU Media Research Laboratory. Much of the initial work on Pad++ was done at the University of New Mexico.

Here are the links to folks currently working on Pad++:

- **University of Maryland, HCIL**
  Benjamin Bederson
  Allison Druin
  Tammara Combs
- **Univ. of California, San Diego, Cognitive Science Dept.**
  James Hollan
- **University of New Mexico, CS Dept.**
  Jason Stewart
- **NYU Media Research Laboratory**
  Ken Perlin
  Jonathan Meyer

And here are the folks who have worked on Pad++ in the past:

- **University of New Mexico, CS Dept.**
  Hugh Bivens
  George Hartogensis
  David Proft
  Laura Ring
  David Rogers
  David Vick

Ron Hightower
Mohamad Ijadi
David Thompson
Ying Zhao

- **The NYU Media Research Laboratory**
  David Bacon
  Troy Downing
  David Fox
  Athomas Goldberg
  Noah Wardrip-Fruin
- **Bellcore**
  Larry Stead

What research directions are you following?

The Pad++ group is developing Pad++ in many ways, such as:

- Creating Pad++ GUI components.
- Developing an internet based distributed Pad++.
- Performing Pad++ usability studies.
- Developing Pad++ applications for understanding large datasets.
- Porting Pad++ to Windows95/NT.
- Developing easy to use authoring tools.

How do I contact the Pad++ group

You can contact the Pad++ group at the following email addresses:

- pad-info@cs.umd.edu - Send your comments and information requests here
- pad-bug@cs.umd.edu - Report bugs here
- pad-chat@cs.umd.edu - Send questions/comments to all Pad++ users that have asked to be on this list

---

How do I get a Pad++ license?

For non-commercial (education, research, or in-house) use, all you need to do to use Pad++ is to fill in the online registration form. If you are interested in a commercial license, you should contact us at pad-info@cs.umd.edu. The Pad++ Consortium currently consists of UNM and NYU, and license can be negotiated from either institution.

How do I get on the Pad++ mailing lists?

When you download Pad++, there are two checkboxes to add you to the mailing lists. Or, you can send mail to pad-info@cs.umd.edu, and we'll add you manually.

There is an *announce* list which has very low volume. This list is just for us to send you mail about new Pad++ releases and other important information. This is a private list, and others can not use it, so you don't have to worry about getting much unsolicited mail. There is also a *chat* list which has moderate volume. This list is for the Pad++ user community to communicate with us and each other about ideas, solutions, requests, etc.

Questions about Pad++

What other online info is there on Pad++ or Zoomable interfaces?

There are several online papers about Pad++ and related issues here

Is Pad++ available?

Yes it is. You can register online and download it here.

What platforms does Pad++ run on?

Pad++ runs on just about any Unix platform. We have tested it on PCs running Linux, SGIs, and Suns (SunOS and Solaris).

Pad++ now also runs on Windows95/NT. This port is still early, and the Windows version is not very reliable. Also, it is slower because it is currently implemented with Tk's X emulator.

Why does the Windows version run slower than the Unix version?

The Windows version of Pad++ runs slower than the Unix version because we are using the Tk X emulator. In a future version, we will write a native Windows renderer, and expect that it will be at least as fast as the Unix version.

Is there a Pad++ FTP site

No, all public information about Pad++ is at this web site.

What are the plans for the Web browser?

We built a prototype Web browser in Pad++ in 1995. At that time, HTML was quite simple, and the idea of building our own better browser was quite tempting. We build what was at the time a cutting edge browser - with HTML 1.0. Since then, the web has advanced greatly, and we realized that we didn't want to get in the business of competing with the major Web browser companies.

Instead, we are taking advantage of the new features of HTML by building a new generation Web browser that piggybacks on existing commercial ones. The new Pad++ Web browser, called PadPrints, monitors what the browser is doing and what pages are being visited. It builds a map showing your history of interaction with Web. You can use your commercial browser normally, but at any time, you can go to PadPrints in a separate window to control the browser.

Are you going to port Pad++ to Java?

We are looking into connecting Pad++ to Java so that it will be possible to write zooming applications entirely in Java using the C++ core. However, this work still has a long way to go.

Another approach would be to build Pad++ entirely in Java. We have experimented with this, and it appears that at least for the time being, Java is too slow, and the AWT graphics package is not powerful enough. Perhaps when Java compilers are common, and Java2D is commonly available (and well-implemented), it will be possible, but that is still in the future. However, we have created a simple zooming Java applet to give a flavor of what the interaction feels like. See the link on the home page.

Questions about the approach

Its all very well, but why isn't Pad++ 3D?

There are some very attractive 3D systems out there (the Information Visualizer is one of them). However, although they may look impressive, 3D systems are typically hard to navigate using current display and pointer technologies. They also require high levels of computing resources. Its not obvious that a 3D interface is appropriate for a handheld device or for a cheap home computer.

Pad++, on the other hand, can be implemented very efficiently using a small home computer, and is easy to navigate using a mouse.

How does Pad++ relate to traditional windowing systems?

There are two answers to this - choose one you like:

1. Pad++ can be seen as a mechanism for creating applications which have a zooming component. For some applications, a zoomable interface may be more appropriate than a more traditional windows based approach - for example, zooming seems to be particularly effective for applications which need

to present large amounts of data in an intuitive and understandable manner. This especially true for data that is hierarchical in nature as information that is deeper in the hierarchy can be represented as being smaller on the Pad++ surface.

2. Pad++ could also be viewed as a way to implement an alternative windowing system based on zooming. Your whole desktop could be zoomable. This seems especially attractive for systems which have small screens, such as handheld computers (i.e. PDA's).

Technical questions about Pad++

What does the Pad++ code consist of?

Pad++ consists of a C++ core that provides zoomable objects. Applications can be written in C++ (this API is still under development), or with Tcl/Tk, a scripting language that implements the zoomable draw application which you have seen.

Pad++ runs on most unix machines including PC's running Linux, on workstations from Silocon Graphics, IBM and Sun, and probably on most other UNIX-like systems that support X11. Pad++ also runs on Windows95/NT, but this port is still slow and unreliable.

How do you get Pad++ to zoom so quickly?

In order to keep zoom animations fast, Pad++ employs several tricks:

Spatial Indexing

Pad++ uses an R-Tree internally to quickly determine which objects are visible in a given view.

Level of Detail

When things become too small to see, Pad++ ceases to render them. When Pad++ starts to get too slow, it renders medium sized things in a more ugly fashion. When the system is idle, things that have not been rendered fully are *refined* - see below.

Refinement

When trying to achieve a high frame rate, Pad++ use a reduced level of detail. Then, when the system is idle, successive refinements are performed to increase the level of detail.

Adaptive render scheduling

The system monitors render times, and adapts the rendering algorithm to maintain a constant frame rate during zooms and pans.

Pad++ dies as it starts up on Windows95/NT. Why?

There is a bug in the Windows95/NT version where it dies if there is a **space** in any of the directory names that Pad++ is installed in. So, if you installed Pad++ in the '*Program Files*' directory, it will die with this behavior. Try re-installing (or just moving Pad++) in the top-level directory, and it should work. Sorry about this - we'll try and fix this in the next version.

Building Pad++ on Solaris with gcc 2.8.1 hangs. Why?

We've had reports that building Pad++ on Solaris 2.5 with gcc 2.8.1 hanges when compiling generic/object.cpp. A workaround is to remove the -fPIC flag from the Makefile. This flag is only necessary when compiling shared libraries, but isn't necessary when just compiling an executable, so you should be able to remove it safely.

Another approach is to configure with:
configure --disable-load
and then add '-ldl' at the end of the link line

Why does my Tcl socket code hang when using Pad++?

When Tcl 7.6 performs a socket read in "line buffered" mode, it reads only until the new-line character. If the writer sends a zero-terminated string, Tcl 7.6 leaves the zero byte in the buffer. The next read, then, returns a string that begins with a zero byte; we interpreted this incorrectly as a read of a null string, when in fact it was a long string following a zero byte. This problem does not occur with Tcl 8.1, the latest release.