Pad++: A Zoomable Graphical Sketchpad For Exploring Alternate Interface Physics[1]

Benjamin B. Bederson
James D. Hollan
Computer Science Department
University of New Mexico
Albuquerque, NM 87131
(bederson@cs.unm.edu, hollan@cs.unm.edu)

Ken Perlin
Jonathan Meyer
David Bacon
Media Research Laboratory
Computer Science Department
New York University
New York, NY 10003
(perlin@play.cs.nyu.edu, meyer@play.cs.nyu.edu, bacon@play.cs.nyu.edu)

George Furnas
Bell Communications Research
445 South Street
Morristown, NJ 07960
(gwf@bellcore.com)

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

Keywords
Interactive user interfaces, multiscale interfaces, zoomable interfaces, authoring, information navigation, hypertext, information visualization, information physics, user interface management system (UIMS).

Abstract
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 design of new computationally-based work materials.

[1]To appear in the *Journal of Visual Languages and Computing.*

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 just a title that is readable. Similarly, when stretching out a spreadsheet, 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.

Figure 1: A sequence of views as we zoom into some data.

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. The exploration of virtual 3D worlds is one alternative. It follows quite naturally from traditional direct manipulation approaches to interface design and involves similar underlying metaphors, although they are enriched by the greater representational possibilities afforded by moving to richer 3D worlds.

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 [22] 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 often referenced places. Frequently consulted papers are 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 [19][20]. 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 [26]. 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 readily and effectively access. 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 [16] 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 [21] and the recent work of moveable filters at Xerox [4][31].

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 effectively navigate. 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 [25] (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 a 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 kind of 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.)

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.

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 a 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.

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.

We have also implemented visual bookmarks as another navigational aid. Users can remember places they've 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).

Figure 3: Visual bookmarks let users remember interesting places they've been by showing miniature views of those places. Clicking on one of these miniature views animates the main view to the remembered location.

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. 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 isn't lost and can continue to be used.

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've 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.

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 widget, 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.

Figure 4: These lenses shows textual data as scatter plots and bar charts.

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 doesn't 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. Also, 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 won't 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.

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 [16]. 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.

Figure 5: Many different HTML documents loaded in Pad++. Their layout implicitly shows the history of the users interaction. The two views show the same tree focused on different nodes.

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 of 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.

Figure 6: A view of our filesystem.

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.

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.

Figure 7: A sequence views into the history of computers and interfaces.

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.

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.

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 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.

Figure 8: Pad++ help screen with oval document layout.

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 [17].

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

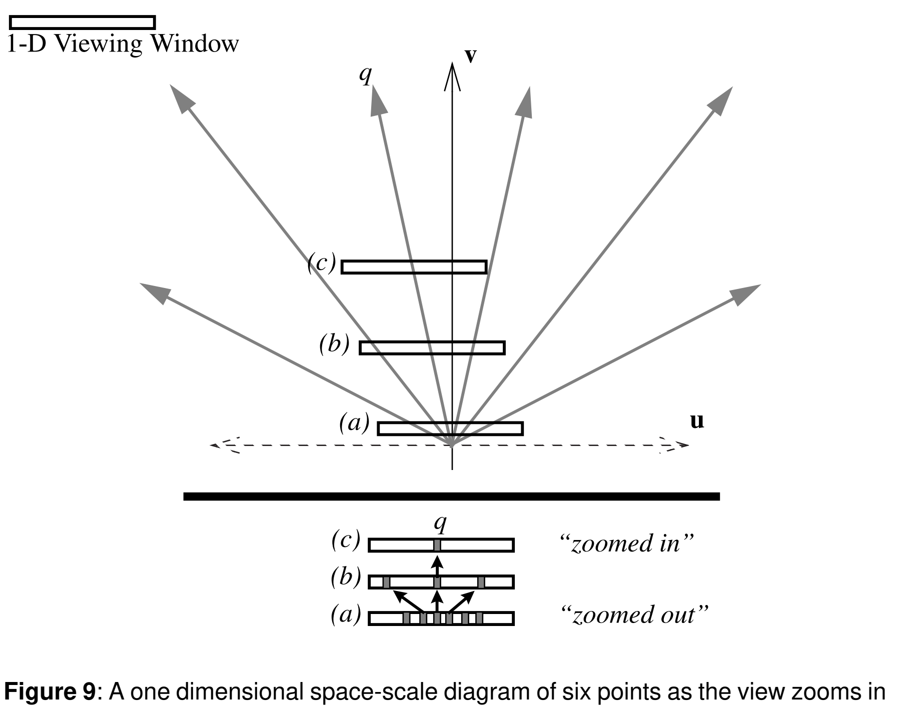

Figure 9: A one dimensional space-scale diagram of six points as the view zooms in from *(a)* to *(b)* to *(c)* around the point q.

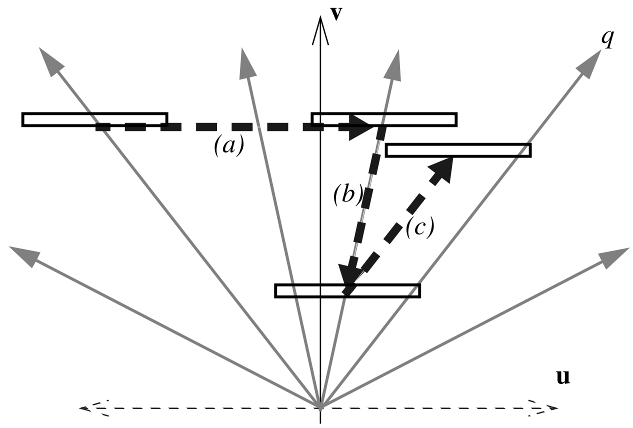

Figure 10: Basic pan-zoom trajectories are shown in the heavy dashed lines. (a) is a pure pan, (b) is a pure zoom (out), and (c) is a zoom around the point q.

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 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 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 *(x1, y1)* to *(x2, y2)* that goes outside such a parallelogram is non-

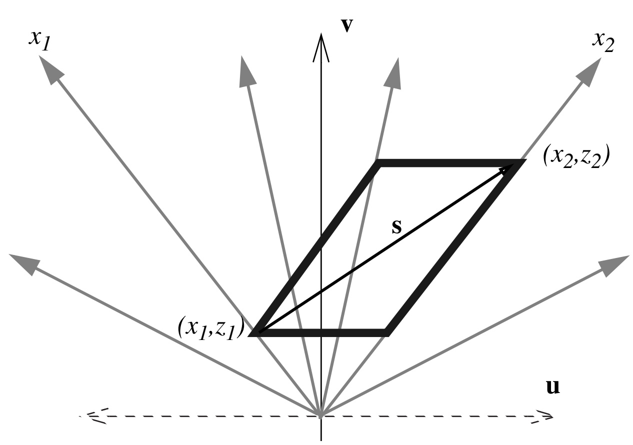

Figure 11: Solution to the simple joint pan-zoom problem. The trajectory **s** monotonically approaches point *(x2, z2)* in both pan and zoom.

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 [17]) 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

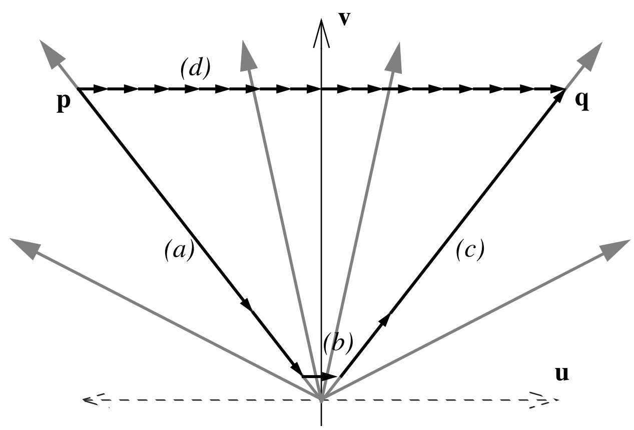

Figure 12: The shortest path between two points is often not a straight line. Here each arrow represents one unit of cost. Because zoom is logarithmic, it is often “shorter” to zoom out (a), make a small pan (b), and zoom back in (c), than to make a large pan directly (d).

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 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

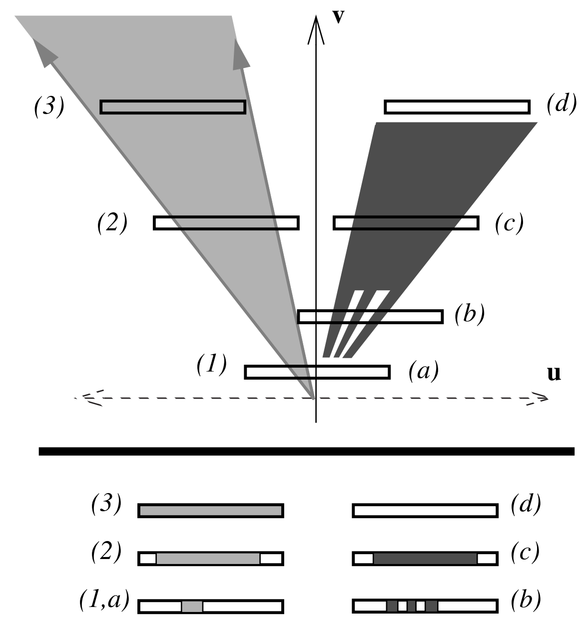

Figure 13: Semantic Zooming. Bottom slices show views at different points.

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.

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 [28]. 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.

Figure 14: A prototype zoomable choice widget.

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 postfix stack language whose simple structure allows execution times roughly 10 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 automatically 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 a 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. On a Pad++ surface with 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 new C++ code.

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:** Render 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:** Render items differently depending on how large they appear on the screen. If they are small, render them with lower resolution.
- **Region Management:** Only update the portion of the screen that has been changed. Linked with refinement, this allows different areas of the screen to refine independently.

- **Clipping:** Only render 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.

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 easily experiment 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 [25] 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, sliders, 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 previously mentioned, we use KPL for high-speed animations. We also are considering incorporating an alternative language, such as Scheme, for more general programming which needs high speed interaction.

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 (2, 1):

```
.pad create rectangle 0 0 2 1 -fill red -pen black
```

- Put item number 5 at the point (3, 3), make the object twice as big, and make the object anchored at that point on its northwest corner:

```
.pad itemconfig 5 -anchor nw -place "3 3 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 make scripts get 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 (1, 0) with a magnification of 5, and take 1000 milliseconds for the animation:

```
.pad moveto 1 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 object ids that contain the text "foo"

```
.pad find withtext foo
```

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:

- Modes

  Every event handler is defined for a specific *mode*. The mode is a simple text string and defaults to `all` if it is not specified. The Pad++ surface has a set of active event modes associated with it (that always includes the `all` mode).

Only those event handlers whose mode is currently active will be fired. This allows the creation of many different event handlers that are selectable by setting the Pad++ mode.

- Event Searching Protocol

  When an event hits an item and there are no event handlers defined for that item, there is a well-defined event searching protocol that specifies which other items will be searched for event handlers. If the event went through a portal, then the portals are checked for event handlers. The portals are checked from the bottom up, that is, in the reverse order that event went through the portals. To summarize, the searching order is as follows:

  1. Most specific object
  2. Objects associated by tag (“`all`” being last)
  3. Portals (and associated tag objects)

- PortalIntercept event

  Portals can intercept events as the events pass through them with the PortalIntercept event. PortalIntercept is a new event sequence recognized by the Pad++ bind command. PortalIntercept event handlers get called for every event that passes through a portal in top down order. They do not replace other event handlers, but instead get called before those handlers. A PortalIntercept command may execute any code, and then it can return a special value that can modify the event. The modifications include killing the event, stopping the event at the portal rather than passing it through, changing the list of active modes on the surface the event hits for this event, and changing the coordinates of the event.

- Passing Events

  When an event is fired, it is often useful for a handler to pass the event on to the next most general event handler. This is most commonly used to have a single event trigger the event handlers for specific items as well as classes of items.

Messages

Items can send arbitrary messages to other items or groups of items. This message sending facility is analogous to the Event mechanism, including the Searching Protocol and passing mechanism.

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.

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 more naturally arises 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 following a physics-inspired course in thinking about interface design. It derives, like most interesting interface ideas, from the seminal work of Sutherland [32] 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 [18] [34] in particular has taken a physics-as-interface approach to construction of dynamic interactive interfaces.

Smith's Alternate Reality Kit [29] [30] and languages such as Self [33] 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 smoothly integrate 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 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 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++ won't 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] Ronald M. Baecker, *Human Factors and Typography for More Readable Programs*, ACM Press, 1990.

[2] Benjamin B. Bederson, Larry Stead, and James D. Hollan, *Pad++: Advances in Multiscale Interfaces*, Proceedings of ACM SIGCHI Conference (CHI'94), 315-316.

[3] 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.

[4] 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.

[5] Alan Borning. Thinglab: A Constraint-oriented Simulation Laboratory, Technical Report SSL-79-3, Xerox Palo Alto Research Center, 1979.

[6] Alan Borning and R. Duisberg. Constraint-Based Tools for Building User Interface, ACM Transactions on Graphics, 1986, 5(4), 345-374.

[7] R. Brooks. *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, 1986, 2(1), 14-23.

[8] 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.

[9] 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.

[10] Muriel Cooper, Personal Communication, 1994.

[11] Steve Deerwester, Sue T. Dumais, George W. Furnas, Tom K. Landauer, and Ray Harshman. *Indexing by Latent Semantic Analysis.* Journal of American Society of Information Science, 41, 1990, pp.391-407.

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

[13] D. Ebert, et. al., *Texturing and Modeling*, A Procedural Approach, Academic Press, London, 1994.

[14] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr, *Seesoft: A Tool for Visualizing Line-Oriented Software Statistics*, IEEE Transactions on Software Engineering, 18 (11), 1992, 957-968.

[15] Kim M. Fairchild, Steven E. Poltrock, and George W. Furnas. *SemNet: Three-Dimensional Graphic Representations of Large Knowledge Bases*, in Cognitive Science and its Applications for Human-Computer Interaction, Lawrence Erlbaum Associates, 1988.

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

[17] George W. Furnas and Benjamin B. Bederson, *Space-Scale Diagrams: Understanding Multiscale Interfaces*, Proceedings of ACM SIGCHI'95, in press.

[18] Michael Gleicher, Briar: A Constraint-Based Drawing Program, In CHI'92 Formal Video Program, 1992.

[19] William C. Hill, James D. Hollan, David Wroblewski, and Tim McCandless, *Edit Wear and Read Wear*, Proceedings of ACM SIGCHI'92, pp. 3-9.

[20] William C. Hill and James D. Hollan, *History-Enriched Digital Objects*, in press.

[21] James D. Hollan, Elaine Rich, William Hill, David Wroblewski, Wayne Wilner, Kent Wittenburg, Jonathan Grudin, and Members of the Human Interface Laboratory. *An Introduction to HITS: Human Interface Tool Suite*, in Intelligent User Interfaces, (Sullivan & Tyler, Eds), 1991, pp. 293-337.

[22] James D. Hollan and Scott Stornetta, *Beyond Being There*, Proceedings of ACM SIGCHI'92, pp. 119-125. (also appeared as a chapter in Readings in Groupware and Computer Supported Cooperative Work (Becker, Ed.), 1993, 842-848.

[23] George Lakoff and Mark Johnson, *Metaphors We Live By*. University of Chicago Press, 1980.

[24] Mackinlay, J.D., Robertson, G.G. and Card, S.K., *The perspective wall: detail and context smoothly integrated.* In Proceedings of CHI'91 Human Factors in Computing Systems, ACM press, 173-179.

[25] John K. Ousterhout, *Tcl and the Tk Toolkit*, Addison Wesley, 1994.

[26] Ken Perlin and David Fox. *Pad: An Alternative Approach to the Computer Interface*, Proceedings of 1993 ACM SIGGRAPH Conference, 57-64.

[27] Ken Perlin, *An Image Synthesizer*, Proceedings of ACM SIGGRAPH '85, 19(3), 287-293.

[28] Ken Perlin, *Danse Interactif*, Video Proceedings of ACM SIGGRAPH '94, 28(3), 1994.

[29] Randy B. Smith, *The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations*, Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages, 99-106.

[30] Randy B. Smith, *Experiences With The Alternate Reality Kit: An Example of the Tension Between Literalism and Magic*.,Proceedings of ACM CHI+GI '87 Conference, 61-67.

[31] Maureen C. Stone, Ken Fishkin, and Eric A. Bier. *The Movable Filter as a User Interface Tool*, to appear in Proceedings of ACM SIGCHI'94.

[32] Ivan E. Sutherland. *Sketchpad: A man-machine graphical communications systems*, Proceedings of the Spring Joint Computer Conference, 1963, 329-346, Baltimore, MD: Spartan Books.

[33] David Ungar and Randy B. Smith, *Self: The Power of Simplicity*, Proceedings of OOPSLA '87 Conference, 227-241.

[34] Andrew Witkin, Michael Gleicher, and William Welch. *Interactive Dynamics*, Computer Graphics, 1990, 24(2), 11-21.