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

[9] 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.

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

[19] 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.

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

- **Spatial Indexing:** Objects are stored internally in a hierarchy based on bounding boxes which allow fast indexing to visible objects.
- **Clustering:** Pad++ automatically restructure the hierarchy of objects to maintain a balanced tree which is necessary for the fastest indexing.
- **Region Management:** Only update the portion of the screen that has been changed. When modifying objects, this means all places an object is visible (i.e., within multiple portals) must be updated. Linked with refinement, this allows different areas of the screen to refine separately.
- **Refinement:** Render fast while navigating by using lower resolution, and not drawing very small items. When the system sits still 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.
- **Image Caching:** Store zoomed images in a special cache. Since magnifying images is computationally expensive, we cache them, and then use the cache when rendering the image if it doesn't change size.
- **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 interrupted 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 if they become visible again.

CONCLUSION

We implemented Pad++, a zoomable graphical interface substrate, focusing on efficiency and extensibility. We are using Pad++ to explore new interaction mechanisms made possible by zooming. By implementing several efficiency mechanisms acting in concert, we are able to maintain high frame-rate interaction with very large databases.

AVAILABILITY

We intend to make Pad++ freely available for non-commercial use. See “http://www.cs.unm.edu/pad++” for current information (the Pad++ project home page).

ACKNOWLEDGEMENTS

This work was supported in part by ARPA grant N660011-94-C-6039 to the University of New Mexico.

We would like to thank several members of the NYU Media Research Laboratory with whom we are collaborating on this project. This includes Ken Perlin, Jon Meyer, David Bacon, and David Fox. We also would like to acknowledge members of the Computer Graphics and Interactive Media Research Group at Bellcore, including George Furnas and Kent Wittenburg.

REFERENCES

[1] Benjamin B. Bederson and James D. Hollan, *Pad++: A Zooming Graphical Interface Widget for Tk*, in Proceedings of the 1994 TCL/TK Workshop, 73-84.

[2] Benjamin B. Bederson, James D. Hollan, et. al., *Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics*, Journal of Visual Languages and Computing (in Press).

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

[4] 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.

[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] 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.

[7] Bay-Wei Chang and David Ungar, *Animation: From Cartoons to the User Interface*, in Proceedings of 1993 ACM User Interface and Software Technology Conference (UIST'93), pp. 45-55.

der, 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++ may be extended entirely with Tcl scripts (i.e., no C/C++ code). This provides a 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 automatically makes them 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.

While object type extensions may consist of compound objects, there must be a single control object that is used to access the others. This is often a portal looking onto the other objects on an unmapped Pad++ surface. For example, standard widgets such as buttons and entries are defined this way. These are similar to other Tcl/Tk extensions called MegaWidgets except that these act on items within the Pad++ widget, rather than on standard Tk widgets.

Extensions are defined with the following commands, where <extension> refers to the name of the extension (such as ‘button’), and the words inside braces are the command’s arguments. Note that some of these commands are required to follow certain return argument conventions.

- padcreate_<extension> {*PAD*}

  This procedure gets called when <extension> is created with the create command. It takes a single argument, *PAD* which is the name of the Pad++ surface to create the object on. The only requirement of this function is that it must return the id of the single object to be referred to for this extension.

- padconf_<extension> {*PAD id option ?value?*}

  This procedure gets called with option-value pairs when <extension> is created and when it is configured with the itemconfigure command. It takes three or four arguments. *PAD* is the name of Pad++ surface the object is on. *id* is the object’s reference id. *option* is the option being configured. If *value* is not specified, then this function must return the current value of this option. If *value* is specified, then this function should change this option to the specified value, and return the current value.

  The other requirement of this function is that if option is not specified (i.e., it is called with too few arguments), it must return an error with a return value of the list of legal options.

- padcmd_<extension> {*PAD option ?args...?*}

  This procedure gets called when <extension> is executed as a Pad++ command. This allows an extension object to define arbitrary sub-commands. For example, executing “pathName <extension> invoke myWidget” would call this procedure with *PAD* bound to “pathName”, *option* bound to “invoke”, and the first argument of *args* bound to “myWidget”. Note that this command may be defined without creation and configure commands in order to make generic command extensions without widgets. This command has no return requirements.

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 objects can be animated with either render or timer callbacks. Finally, panning and zooming is animated under user-control, defined by Tcl scripts.

All automatic animations use slow-in-slow-out motion [7]. This means that the motion starts slowly, goes quicker in the middle, and ends slowly - resulting in smoother feeling animations. This does not affect the time the animation takes because time is effectively stolen from the middle to put at the ends. User-controlled animations are specified precisely by the user, and there is no distortion of the motion speed.

Efficiency

In order to keep the animation frame-rate up as the dataspace size and complexity increases, we implemented several standard efficiency methods, 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.

Briefly, the implemented efficiency methods include:

Searching Protocol), and the second is intercepting the event as it passes through the portal (PortalIntercept events).

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. Every item has a list of items which catch events for it. This list of items is known as the list of *event catchers*. When an item doesn't have an event handler, its event catchers are checked. If none of the event catchers have an appropriate event handler, and if the event went through a portal, then the portals and their event catchers are checked for event handlers. Finally, the Pad++ surface itself is checked. 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. Event catchers (and associated tag objects)
4. Portals (and associated tag objects and event catchers)
5. Pad++ surface that object is on

- 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 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.

When a message is sent within a render callback (see below), the message is automatically sent through the list of portals that the current object is being rendered through. This allows the Event Searching protocol to apply to messages. This portal list is overridable.

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 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 call the renderItem function at any point to render the object. 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. We call these *procedural objects*.

  Procedural objects can be 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 ren-

Portals can also be used to create indices. Creating a portal that looks onto a hyperlink allows the hyperlink to be followed by clicking on it within the portal - which changes the main view. This however, will probably move the hyperlink off the screen. We can solve this problem by making the portal (or any other object for that matter) *sticky*, 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 you don't lose the portal index, and thus, it can continue to be used.

Lenses

Designing user interfaces is typically done at a low level, deciding on user interface components, rather than on the task at hand. If the task is to enter a number into the computer, we should be able to place a generic number entry mechanism in the interface. However, typically, the specific number entry widget, such as a slider or dial, is decided on, and it is fixed in the interface.

We can use lenses to design interfaces at the task level. For example, we've designed a set of number entry lenses for Pad++ that can change a generic number entry mechanism into a slider or dial, as the user prefers. For example, by default the generic number entry mechanism might allow 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 the look and interaction of components seen through them. They can be dragged around the Pad++ surface examining existing data. For example, some data might normally be depicted by some 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 1).

We think this can be a useful teaching aid as it helps to make the notion that there can be multiple representations of the same underlying data more intuitive. 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.

Semantic Zooming

Once we make zooming a standard part of the interface, many parts of the interface need to be re-evaluated. 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 abstraction 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 (we implemented by dragging a lens over the digital clock) is similar - it doesn't show the second hand or the minute markings when zoomed out.

IMPLEMENTATION

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. Some of the significant differences between Pad++ and the Canvas widget are summarized here.

Events

As with the Canvas, 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, that item's event handler gets 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`. 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.

  This might be used in a drawing application where pressing a button on a tool palette causes the left mouse button to have a different function. With modes, all the event handlers can be defined once, and pressing buttons on the tool palette simply changes the Pad++ surface mode.

- Event Searching Protocol

  There are two mechanisms that allow portals to change the way users interact with items *through* portals. The first is by inheritance of events (Event

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.

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. We are not pursuing 3D worlds because we feel that two dimensional interfaces have much to offer and have not yet been fully tapped. Some recent work at Xerox Parc [6][16], though, shows some of the potential of 3D interfaces.

Pad++, based on a spatial information structure, is based on a long history of related work. Many ideas in this area are based on work by Sutherland [20], where he demonstrated the first interactive graphical computer system. Perhaps the first work to try to tap people's natural spatial abilities was Donelson [8], where he developed an interface based on interaction with an entire room linking voice and gesture to control access to spatially situated data.

The use of new metaphors to motivate interface research has directed several researchers. Hill and Hollan have been exploring the notion of history-enriched digital objects [12][13][15]. This is the notion that an object's representation should be a natural by-product of normal activity. This is similar to the physics of certain materials that show wear associated with use. Such wear records a history of use and at times 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.

This motivating strategy has lead us to explore new methods for interacting with graphical data. As part of that exploration we have formed a research consortium to design a successor to Pad [18]. This new system, Pad++ [1][2][3][4], 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.

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 it, is rarely presented in ways that reflect either its rich structure or dynamic character.

We address the information presentation problem of how to provide effective access to a large structure of information on a much smaller display. Furnas [10] 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. More recent work has shown other approaches to address the local detail versus global context problem [11][16]. Eick visualizes large software packages by representing each line of code with just a few pixels on the display, using color to represent various kinds of information, such as author, date of creation, and number of modifications [9].

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 [14] and the recent work of moveable filters at Xerox [5][19].

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 is a useful substrate to *structure* information.

Portals

Portals are special items that allow you to look onto other areas of the Pad++ surface, or even other surfaces. Each portal passes events to the place it is looking. Thus, you can pan and zoom within a portal. In fact, you can perform any kind of interaction on the Pad++ surface through a portal. Portals can filter input events as they pass through the portal, providing a mechanism for changing the semantics of interactions with objects when viewed through a portal. Portals can also change the way objects look. When used in this fashion, we call them *lenses* (see below).

Portals can be used to duplicate information efficiently, and also provide a method to bring physically separate data near each other. Portals can be created near each other, each looking at places far away.

Advances in the Pad++ Zoomable Graphics Widget

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

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

KEYWORDS
Interactive user interfaces, multiscale interfaces, zoomable interfaces, authoring, information navigation, hypertext, information visualization, information physics.

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 some of the differences between Pad++ and the standard Tk Canvas widget.

INTRODUCTION

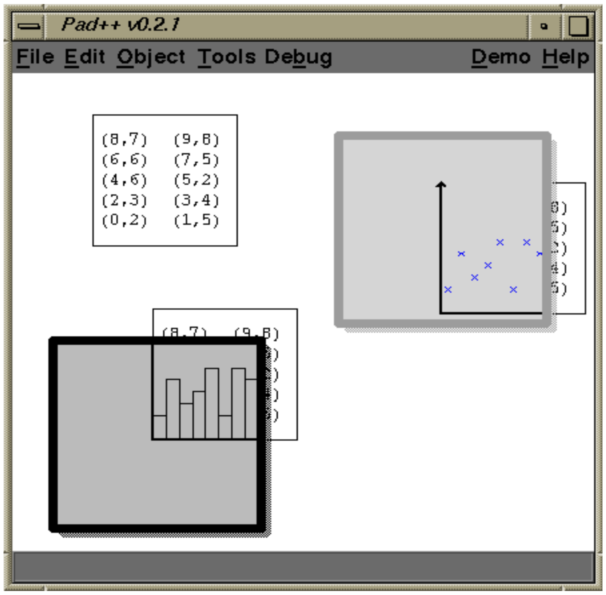
Imagine that the computer screen is made of a sheet of a miraculous new material that is stretchable like rubber, but continues to display a crisp computer image, no matter what the sheet's size. Imagine that this sheet is very elastic and can stretch orders of magnitude more than rubber. Further, imagine 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 visual 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 - perhaps those satisfying some search criteria.

Imagine also new kinds of stretching 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.

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 is an approach to navigate through a large information space via direct manipulation. By tapping into people's natural spatial abilities, we hope to increase users's intuitive access to information. Of course, traditional computer search techniques are also available, but they produce an automatic animation to the area with the desired data - bridging traditional and new interface metaphors.



(8,7)   (9,8)
(6,6)   (7,5)
(4,6)   (5,2)
(2,3)   (3,4)
(0,2)   (1,5)

Figure 1 These lenses shows textual data as scatter plots and bar charts.

The following paper was originally presented at the
Third Annual Tcl/Tk Workshop
sponsored by Unisys, Inc. and USENIX Association
Toronto, Ontario, Canada, July 1995.

Advances in the Pad++ Zoomable Graphics Widget

Benjamin B. Bederson and James D. Hollan
Computer Science Department
University of New Mexico
Albuquerque, NM 87131