Implementing a Zooming User Interface: Experience Building Pad++

Ben Bederson[†]
University of Maryland
Computer Science Deptartment
Human-Computer Interaction Lab
3171 A.V. Williams Building
College Park, MD 20742
bederson@cs.umd.edu

Jon Meyer
New York University
Computer Science Department
Media Research Lab
719 Broadway, 12th Floor
New York, NY 10003
meyer@cs.nyu.edu

Abstract

We are investigating a novel user interface paradigm based on zooming, in which users are presented with a zooming view of a huge planar information surface. We have developed a system called Pad++ to explore this approach. The implementation of Pad++ is related to real-time 3D graphics systems and to 2D windowing systems. However, the zooming nature of Pad++ requires new approaches to rendering, screen management, and spatial indexing. In this paper, we describe the design and implementation of the Pad++ engine, focusing in particular on rendering and data structure issues. Our goal is to present useful techniques that can be adopted in other real-time graphical systems, and also to discuss how 2D zooming systems differ from other graphical systems.

Keywords

Zooming User Interfaces (ZUIs), Real-time Computer Graphics, Animation, 3D Graphics, Windowing Systems, User Interface Management Systems (UIMS), Pad++.

Availability

Pad++ is available for non-commercial (educational, research and in-house) use from
`http://www.cs.umd.edu/hcil/pad++`

Introduction

For several years, we have been investigating an alternative user interface paradigm based on zooming. In our approach, users navigate over a single large information surface. Documents can be placed on the surface at any position, and also scaled to any size. Navigations (including pans, zooms and hyperlinks) smoothly animate the view so the requested document appears at the right position and size.

Zooming User Interfaces (ZUIs) are exciting to us because they present a possible solution to problems which plague previous approaches to user interfaces. ZUIs present information graphically and exploit people's innate spatial abilities. Detail can be shown without losing context, since the user can always rediscover context by zooming out. ZUIs use screen real estate effectively, and have great potential even on small screens. One way of thinking about ZUIs is that all the information you need is there if you look closely enough.

To explore these kinds of interfaces, we built a zooming graphics engine called Pad++, which brings together a unique combination of features, and supports smoothly animated zooming of large datasets using off-the-shelf PC-class hardware.

[†] Ben Bederson carried out this much of the work presented in this paper at the University of New Mexico, Albuquerque, NM 87131

In this paper, we describe the design and implementation of the Pad++ engine, focusing in particular on rendering and data structure issues. Our goal is to present techniques that can be adopted in other real-time graphical systems, and also to show some of Pad++'s unique characteristics and describe how 2D zooming systems differ from other real time graphical systems. We have written elsewhere about applications we have built using Pad++ [1][2][3].

Background

Zooming interfaces have a long intellectual history. Over 30 years ago, Ivan Sutherland showed the first interactive object-oriented 2D graphics system [4]. This visionary system, called Sketchpad, demonstrated many components of today's interfaces. It even provided rudimentary zooming: every drawn object could be scaled and rotated. Fifteen years later, the Spatial Data Management System (SDMS) [5] used zooming as an integral part of the interface. SDMS was the first system to use zooming through a two dimensional as a metaphor for finding information. SDMS supported two semantic levels - an "Overview" and a "Zoom into Icon" level. However, SDMS was implemented with custom hardware and consisted of several machines and displays integrated into a room with all of the controls organized around a single large chair.

Eleven years later, a system called Pad[6] was developed and presented at an NSF workshop in 1989. Pad integrated zooming into a single program that ran on inexpensive hardware. Pad was programmable, and supported interactive text editing, drawing, documents and portals (views onto different areas of the work surface.) It ran in black and white, and was entirely based on bitmaps, so drawings and text to became pixelated as users zoomed in. Pad ran on Sun 3 computers, which do not provide enough processing speed to support continuous smooth zooming, so the interface only allowed users to zoom in and out by powers of two. Subsequently, we built Pad++, a direct but substantially more sophisticated successor to Pad [7][8].

Zooming has been a component of other interface research as well, although not as a primary focus. Several researchers have investigated full 3D interfaces, and these interfaces have implicitly used zooming, since when a user moves close to an object, that object appears bigger. A system developed at Xerox PARC called the Information Visualizer [9] made extensive use of 3D, and showed several applications which took direct advantage of the difference in scale available in 3D.

We are aware of three other implementations of ZUIs. Two are specialized commercial products for World Wide Web navigation. They both use radial layouts to represent hierarchies of information [10], [11]. The third, Tabula Rasa [12] is a Scheme based implementation created by David Fox in his Ph.D. thesis research at NYU.

Other related systems include Self [13], ARK (Alternate Reality Kit) [14] and Xerox Rooms [15]. Like virtual window managers such as fvwm [16], these systems present the user with a 2D planar data surface which is much larger than a single screen, and which the user navigates by panning or by clicking on an iconic map. However they don't support smooth continuous zooming.

Requirements

Conceptually, ZUIs present the user with a zoomable view of a large information surface. The surface is populated by graphical objects – some of which are simple shapes (e.g. lines, images, text items), whereas others may be procedural (e.g. charts, animated objects, multiscale objects). The ZUI manages rendering and interaction with objects on the surface. It also handles resource allocation (colors, fonts, etc.) In this sense, a ZUI is very similar to a graphical window management system like X Windows, except that:

The size of a ZUI data space is not limited by the size of the screen, but instead by the precision of the numeric format used to store coordinates.

ZUIs must scale up to handle tens of thousands of objects, whereas windowing systems need only support a few thousand windows (this figure is naturally constrained by the limited resolution of raster displays.)

Objects are typically not rectangular, and they may be semi-transparent. Windows are expected to be both opaque and rectangular.

Coordinates are floating point, rather than integer based.

There may be many visible views of the surface through *portals* (we discuss portals in more detail later on, but briefly, they are objects on the data surface which show other areas of the surface, and allow interaction with the remote area through the portal.) Windowing systems typically manage a single view.

Rendering is double-buffered by default, since unbuffered pans and zooms produce a distracting screen flicker. Windowing systems let applications control screen pixels more directly.

Windowing systems typically provide applications with a blank window to paint on, and rely on the application itself to draw the contents of each window. This approach is well suited to a wide range of applications, since it gives applications a great deal of flexibility – each application can decide how much effort to spend organizing rendering, performing culling and clipping, handling resources, etc. The downside of this approach is that, in many cases, the code for culling, clipping, colormap handling, resource management, window updating and event handling is duplicated from application to application.

In Pad++, because objects can be non-rectangular and semi-transparent, and also because of portals, handling rendering, culling and clipping is more complex than it is under a windowing system. Consequently, by default, Pad++ performs these operations on behalf of the application. If applications have specialized needs, they can use procedural objects, which let the application handle rendering and clipping issues explicitly.

To help understand these differences, we started constructing a list of basic technical requirements that we felt our zooming user interface should meet. This list has evolved over time, since some of the requirements emerged from experiments involving Pad++. Even so, many of the items on our requirements list have been there since the outset of the project. The requirements list represents the technical problems that Pad++ aims to solve, so we present it here to underpin subsequent discussion. It is not meant as a formal definition of the requirements of a ZUI.

Zooming Interface Requirements:

Maintain and render at least 20,000 objects with smooth interaction: The number 20,000 is somewhat arbitrary, but we felt that this would give us the freedom to implement the kinds of interfaces we were imagining. Maintaining smooth real-time interaction is crucial. The entire metaphor is based on animation. If the system becomes slow and jerky, the metaphor dies. Frame rates of anything less than 10 frames per second are unacceptable. (Here, and in the rest of the paper, we use the term *object* to mean a graphical entity that is manipulated as a whole by the system. This might be a simple object such as a poly-line segment, or a compound object such as an HTML page composed of many characters, line segments and images).

Animate all transitions: All screen changes, whether a change of view or an object movement, should be animated. Since the interface is based on navigating through a surface, it is important to give as much feedback as possible to users about where they are within this space.

Use off-the-shelf hardware: We wanted to make this zooming engine widely available, and to have a wide range of people using Pad++. Therefore, we were obligated to use readily available hardware.

In particular, we haven't used graphics accelerator cards or alternate input devices. Our reference platform is a 200Mhz Pentium Pro running Linux.

Support high quality 2D graphics: Since Pad++ is a graphical user interface system, it is crucial that the graphics are high quality. The engine must support good fonts, high quality images, transparency, rotation, and other graphical effects, matching or exceeding the capabilities already found in windowing systems.

Provide rapid prototyping facility: ZUIs are new, and much of our work involves experimenting with variations of interfaces. As such, it is important for us to be able to quickly modify visualizations and applications. This requirement led to effort connecting Pad++ to various scripting languages so we could create applications through an interpreter.

Support rich dynamics: In addition to zooming, our envisioned ZUI system would include the following features. Objects should be able to have different visual representations at different sizes. That is, zooming into an object should be able to automatically show more detail. We call this *context-sensitive rendering*. It should be easy to animate an object within the space or to animate the view. It should also be possible to create *lenses* – objects which when dragged over other objects changes the visual representation of the object seen within the lens. Finally, we required *layers,* which provide a mechanism to easily change the visibility and drawing order of groups of objects.

Support rich navigation metaphors: In addition to text, images, vector graphics, and hierarchical groups, ZUIs require a few special object types to support navigation. In a ZUI, objects sit at a specific place in the flat space, and yet it is sometimes necessary to be able to view two objects at the same time that are far apart. We use *portals* to solve this problem. Portals are objects on the information surface that look onto another part of the surface. Portals can be used to implement lenses. Also, when the view changes, objects on the screen normally move with the view. Sometimes it is useful to have an object stay in the same position relative to the screen. We support this through *sticky* objects.

Support standard GUI widgets: To build a complete interface system, a ZUI must also support user interface widgets to match those found in existing GUIs. Buttons, sliders, scrollbars, menus, etc. must all be accessible in the same zooming environment as the other graphical objects.

Offer a framework for handling events: Writing applications with zooming, layers, hierarchical groups, portals and sticky objects can be tricky. The event model must be rich enough to gracefully deal with all of these object types, and also simple enough so it is easy to use.

Run within existing windowing and operating system: The ZUI must offer ways to work with existing applications as well as with new zooming applications. It should support established standards, such as X Windows, UNIX, and Microsoft Windows 95/NT, so that users can continue to use their current applications as well as zooming applications.

Having sketched the basic requirements of our ZUI, we next look at implementation issues. We will first look at rendering, then at visible object determination, and finally at the overall structure of the Pad++ software.

Part I: Rendering in ZUIs

A large part of the work required to build a ZUI involves developing a zooming renderer. Before looking in detail at how the Pad++ renderer works, we first present ZGA (Zooming Graphics Accelerator), an imaginary hardware graphics accelerator designed to support zooming user interfaces. The ZGA feature list was constructed by looking at existing graphics hardware and eliciting the features we felt were applicable to ZUIs (based on the requirements list we presented earlier). We describe ZGA here so as to offer a reference point to compare other graphics platforms against. After outlining the features of ZGA, we then go on to describe existing 3D and 2D graphics systems, showing how they overlap with the feature set of ZGA, and where they differ.

Our imaginary ZGA card features:

Text

High quality antialiased text which can be transformed and scaled rapidly

Support for a wide range of fonts, include Type1 and TrueType

International character set support

Lines

A rich set of line drawing styles, including rounded ends, bevels, mitering, and dashes

Scaleable line width and semi-transparent lines.

Images and Movies

Hardware-accelerated image scaling, preferably using filtering to produce smooth results

Support for MPEG and QuickTime digital movies which can be scaled to any size and played at 30 frames a second

Images are maintained and rendered in standard system memory (not specialized video memory) to support paint applications, and applications that present many images at once.

General

Two 24 bit color buffers (for double buffering), a 24 bit depth buffer, an 8 bit alpha buffer for transparency, and a 32 bit accumulation buffer for special effects.

Floating point coordinate system with support for affine transforms

Clipping, including clipping to arbitrary 2D polygons

Fast rasterization of arbitrary 2D polygons

Level-of-quality control over rendering routines for text and images (so the ZUI can trade quality for speed when system resources become overburdened).

Double buffering hardware which supports partial redraws and hardware pans

In hardware terms, the ZGA graphics board is not outlandish. In fact, at first glance, many of these features are present on 3D graphics boards. A natural step is to try to capitalize on the graphical power of modern 3D graphics hardware when building ZUIs.

3D Graphics Hardware

Over the last twenty years, there has been a considerable level of investment in technology to support real-time 3D interactive environments for use in virtual reality, visual simulations, games, information modeling and visualization, and other areas. This work has lead to the development of a number of software standards, such as OpenGL and VRML, as well as to cheap off-the-shelf 3D hardware.

These interactive 3D systems offer considerable graphical power. A high-end system can draw millions of polygons a second, and includes hardware support for anti-aliasing, double-buffering, texturing, lighting and other effects. A home PC with a modest 3D graphics card can support scenes containing rich textures, lights, thousands of polygons, and heart-pumping interactivity (as demonstrated by current games such as DOOM and its successors).

To what extent can alternative 2D systems such as ZUIs benefit from the 3D hardware/software now available? At first glance, real-time 3D systems appear to face many of the same technical challenges as a 2D ZUI such as Pad++. Both must maintain high frame rates, perform animated navigation, handle scene management, clipping, and event propagation, and deal with large numbers of objects. It seems natural to base a ZUI implementation on a 3D graphics API, such as Inventor or OpenGL [17]. An advantage of this approach is that the ZUI can take advantage of hardware acceleration when it is available.

In practice, while 3D programming interfaces such as OpenGL present a good starting point, they have a number of design traits that make their use in ZUIs challenging. For basic 2D graphical elements (such as polygons, lines, images and text), 3D APIs are often either overly rich (and hence wasteful of limited system resources) or have feature gaps (which require a slower software-based solution). Both traits require additional coding on the part of the programmer to produce good 2D results. We discuss some of these challenges below. The discussion is based upon our understanding of the OpenGL rendering API. Other 3D programming interfaces may differ.

Text, images and lines

Current 3D graphics cards can draw large numbers of polygons and lines a second. They often support features such as alpha blending (for transparency), clipping and antialiasing in hardware. However, these systems are optimized for rendering convex 3D polygons (notably triangles and quadrilaterals). Rasterizing arbitrary 2D polygons (for which there are well-known hardware optimizations) is not generally considered part of the 3D API. For example, rendering 2D non-convex polygons in OpenGL must be done by tessellating the polygons into triangles and quads, and then rendering the resulting geometry. This is not as efficient as scan converting the 2D polygons directly.

3D APIs tend to provide only a limited set of line styles. OpenGL offers some basic line drawing primitives, including thick lines and line stipples. However, the thick-line support in OpenGL draws multiple segment lines as a sequence of single segment lines. There is no support for controlling how lines are joined (such as mitering or beveling the joints). There is also no guarantee that pixels within a thick multi-segment line will only be drawn once. On the hardware we've tried, semi-transparent multi-segment lines look unattractive. The limitations with lines in OpenGL can be overcome by rendering line segments as polygons, although this makes line drawing more costly and also complicates the implementation.

For text, consider that a single page of text represents over 100,000 polygons. This is already near the limit of what a typical desktop computer can render at interactive rates. Rendering several pages of text at once (for example in a zoomed out view) by drawing each polygons would be very slow. It is possible to draw each typeface into texture memory, and then use texture mapping as a way to generate scaled text in 3D [18]. For systems with hardware accelerated texture mapping, large quantities of text can be quickly handled using this technique. Unfortunately, texture memory is usually a scarce resource, so only a fairly small set of fonts can be supported in texture RAM. Also, this only works well for small point sizes – for large characters, rendering from polygon outlines produces cleaner results. In practical terms, implementing a rendering engine that can draw large quantities of readable text in many typefaces and sizes is not trivial.

Texture mapping can also be used effectively to scale 2D images. Images rendered from texture RAM can be drawn quickly at any scale and orientation. 3D graphics cards often provide filtering hardware for scaling textures, which produces good looking results. However, as we mentioned above, texture memory is currently a scarce resource in 3D graphics hardware. Handling many images at once (or handling editable images) requires a good "texture residence" mechanism, which is not trivial to implement. For ZUIs, it is desirable to be able to zoom images held in normal process memory. OpenGL does have 2D image drawing capabilities which support scaling, but on all the platforms we have tried the image scaling is integer based, and doesn't work well for high scale factors.

3D graphics hardware availability

None of the challenges listed above prevent us from implementing ZUIs using OpenGL. However the number of current machines that provide hardware acceleration for OpenGL graphics is still small. If ZUIs are to become popular, they must also run reasonably effectively on hardware that is in people's homes today. This means that they must run well on VGA-level graphics cards that only support 8-bit graphics. Eventually, graphics cards with feature sets similar to hypothetical ZGA card will be built. Until then, we must rely on techniques for supporting ZUIs on cheap VGA cards.

2D Graphics Hardware

In the last section we looked at some of the problems of using a 3D graphics system to implement ZUIs. What about using existing 2D graphics hardware instead?

Of course, existing 2D graphics systems are designed to handle 2D graphical elements such as lines, text and images. Surely they will be ideal for zooming user interfaces also?

In reality, 2D graphics APIs make heavy use of caching. They are designed to draw graphical elements repeatedly at a fixed scale. For example, the data structures provided in most windowing systems for fonts and images cache device-level information about how the font or image appears at a given scale. Generating this information for the same font or image at different scales is too slow to achieve continuously animated zooming.

Despite these limitations, we built Pad++ using X windows, a 2D graphics system, in order to give us a wide base of computers to run on. We utilize a number of techniques designed to overcome the shortcomings of X Windows, and meet our goal of smoothly animated pans and zooms. We discuss how Pad++ handles fonts and images below, and also discuss our approach to screen management. We are currently working on writing a Windows 95/NT version of Pad++.

Text in 2D

We have experimented with a number of techniques for drawing text in Pad++. One approach is to use the font capabilities built into the underlying windowing system. Unfortunately, most windowing systems utilize bitmap fonts, which are hard to scale continuously. Worse, each font takes up many kilobytes of memory. The overhead of generating a bitmap font at any given size is a significant number of milliseconds, because the windowing system must generate raster images for all the characters in the font at the given size. Once the font has been rasterized, rendering characters in the font is quick. However, the memory requirements for a large number of fonts at all possible scales are prohibitive, especially when you consider the very large font sizes possible in Pad++ (for example, when a character is scaled up to fill the whole screen).

Microsoft Windows offers TrueType fonts, which are stored in outline form and are therefore scaleable, but generating high quality characters (especially for smaller font sizes) from outline fonts is hard, and outline fonts are also slower to render than bitmap fonts. Windows applications still rely on generated bitmaps for rendering large quantities of text, and rendering high quality small text.

In Pad++, to produce fast zooming fonts, we initially used a simple line font. In this font, each character consists of one or more multi-segment lines, with fewer than 10 segments per character. Lines are quick to render, so this approach is fast, but the characters must be hand-designed and have few curves. The characters look unattractive compared to bitmap based fonts.

As a next step, we obtained an Adobe Type 1 font decoder and attached it to the polygon renderer in Pad++. This generates only a single non-convex polygon per character, but is still too slow for large quantities of text, and the characters suffer from aliasing artifacts (e.g. uneven stems, holes and "pimples") when they are small, which makes small text hard to read. (see Figure 1). There are outline font renderers that overcome the limitations of our simple polygon renderer, but they are slower.

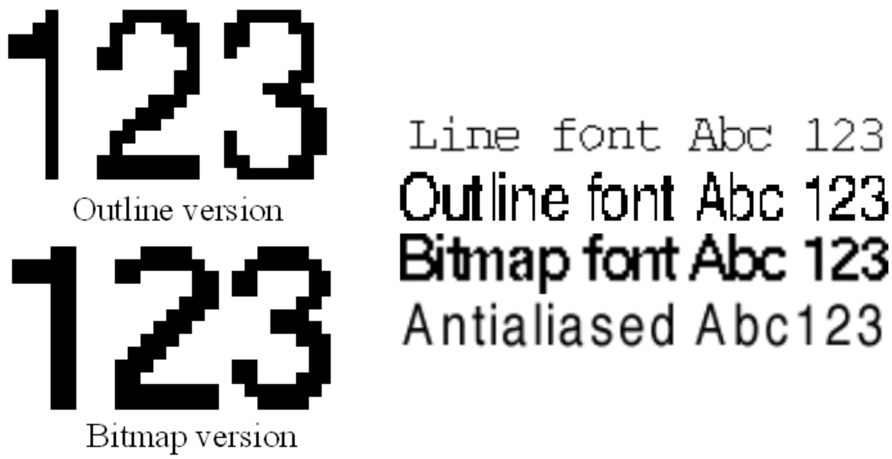

Figure 1: Various font rendering styles. Pad++ uses line fonts, bitmap fonts and outline fonts. It does not currently support antialiased fonts.

To address the speed problem, we implemented a lightweight font cache mechanism. A *fontcache* contains 96 cells, representing the ASCII character set. Each cell holds a 100x100 pixel bitmap. The fontcache mechanism remembers what character size and typeface appears in each cell bitmap. To render a character, say the 'A' character, from a given typeface, the font cache mechanism looks in cell 65 (the ASCII code for 'A') and checks if the character drawn in the cell is the right typeface and size. If it is, then the bitmap is copied directly to the screen. If not, then the cell is cleared, the polygon form of the character is obtained and rendered into the cell, and then the cell is copied directly to the screen.

In Pad++, all the fonts used with the renderer share two global fontcaches. It would be possible to allocate a separate fontcache for each font, though this is expensive since each fontcache requires 96 bitmaps that are 100x100 pixels (implemented with a single 9600x100 bitmap, approximately 100 kilobytes in size).

The fontcache mechanism accelerates rendering of small-sized text, which contains repeated uses of a character in the same font. Since for English text documents and program code this is a frequent occurrence (see Figure 2), the speedup effects of the fontcache for many documents are appreciable. On our reference platform, a sample text document took 367ms to render (2.7 frames per second) with no font cache. With the font cache enabled, the same document took 68ms (nearly 15 fps) – a speedup factor greater than 5. On other systems, even greater speedups have been observed. Also note that the overhead of changing the font size and typeface is also low, since only the characters that are actually rendered are placed in the cache.

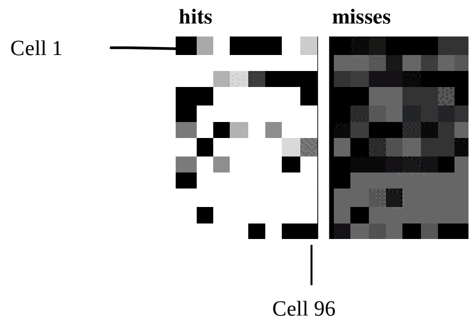

Figure 2: Fontcache hit/miss statistics for a piece of C++ program text. The left image depicts cache hits for character cells in the cache. The right image indicates cache misses for the same cells. The hit/miss count is reflected by brightness. Overall, there are more misses than hits. Most misses are in the higher numbered cells, representing characters ‘a’ – ‘z’.

An improvement to the fontcache might be to provide more cells in the cache for frequently used characters, observing that E is more frequent than T, which is more frequent than A, etc. We haven’t explored this approach fully.

The fontcache lets Pad++ zoom text quickly, but the text is still hard to read at small sizes. Our solution for certain fonts (in particular Times and Helvetica) at small sizes is to switch to X bitmap fonts during refinement (Refinement is discussed below in the section on *Screen Management*.) As we mentioned earlier, loading bitmap fonts is slow, but during refinement slow rendering is less important. This technique is only used for a few fonts (to keep memory usage down), and is not used for rotated text.

Note that if we use a bitmap font (or other windowing system font), we must discard the default spacing metrics for the font and instead compute the spacing for each character explicitly, using floating point coordinates, based on the font metrics of the outline font. If we don’t do this, text tends to “jiggle” as it is zoomed, and the outline and bitmap fonts do not line up consistently. This is because bitmap fonts come only in sizes of one pixel increments which is too coarse for smooth zooming, and because the font metrics in bitmap fonts is usually hand-tuned to produce more readable (but less mathematically accurate) letter spacing.

Eventually, we hope that there will be more hardware support for outline fonts and font antialiasing. Until then, text will remain a challenge for ZUIs, and multi-solution approaches such as the one adopted by Pad++ will be necessary to achieve reasonable text performance.

Images in 2D

In this section we look at how Pad++ zooms images. We do not consider storage issues, or multiscale image representations, but discuss only how to render a moderate size image held in RAM.

The image scaling required in Pad++ is basically a constrained version of texture mapping. 3D graphics systems not only scale images, they also have to rotate and shear the image to account for perspective transforms, and map them onto complex polyhedra. The Pad++ requirements are much simpler, since images are only transformed using translation and scale.

The easiest image scaling algorithm magnifies images using pixel replication and shrinks them using pixel decimation. A simple version for this is:

```
Image zoomImage(Image src, float scale) {
    Image dst = new Image(src.width * scale, src.height * scale);
    for (int y = 0; y < dst.height; y++) {
        for (int x = 0; x < dst.width; x++) {
            dst.data[y][x] = src.data[(int)(y / scale)][(int)(x / scale)];
        }
    }
    return dst;
}
```

However, a literal implementation of this does not produce fast results. Before becoming discouraged and looking for a more sophisticated technique for handling real time image scaling, consider how this code can be optimized. Examining this implementation at the machine code level, every iteration contains approximately 10 instructions:

1 Loop increment
1 Loop test
2 Divides
2 2D array conversion (source)
2 2D array conversion (dest)
1 Memory lookup
1 Memory storage
⇨ Total ~10 instructions

An optimized version of this algorithm takes advantage of the fact that each row is mapped the same way. We can compute a lookup table for mapping the X coordinates of the first row, and reuse it for all the rows in the image. Further optimization can be achieved by "unrolling" the inner loop – so that the overhead of incrementing the loop variable and testing is greatly reduced. In C++, by using pointers instead of 2D arrays, the algorithm runs faster still. The scaling code now looks something like:

```
    // MAKE A PRECOMPUTED TABLE FOR MAPPING ROW X VALUES
    int table = new int[dst.width];
    for (int x = 0; x < dst.width; x++) {
        table[x] = (int)(x / scale);
    }

    // SCALE THE IMAGE
    for (int y = 0; y < dst.height; y++) {
        long *srcPtr   = src.data[y];
        long *dstPtr   = dst.data[(int)(y / scale)];
        int  *tablePtr = table;

        // SCALE THE ROW (UNROLLED)
        for (x = 0; x < dst.width & ~7; x += 8) {
            dstPtr[0] = srcPtr[tablePtr[0]];
            dstPtr[1] = srcPtr[tablePtr[1]];
            dstPtr[2] = srcPtr[tablePtr[2]];
            ...
            dstPtr[7] = srcPtr[tablePtr[7]];
            dstPtr += 8;
            tablePtr += 8;
        }
        // FINISH THE ROW (-NOT- UNROLLED)
        for (; x < dst.width; x++)
            *dstPtr++ = srcPtr[table[x]];
    }
}
```

A rough machine instruction count for the inner loop of this version is:

2 Memory lookup
1 Memory storage

⇨ Total ~3 instructions

This is about 3 times faster than the original version of the code, and produces fast real time image zooming with reasonable performance. With this algorithm Pad++ can zoom an 800x600 pixel image on a 200 MHz Pentium Pro at 30 frames per second.

We've ignored all the special cases: stippling images (for semi-transparent images), color dithering for simulating 24 bit color on 8 bit graphics cards (in Pad++ dithering is done during refinement), images with transparency masks, etc. To keep the image renderer fast, rather than adding statements to the inner loop code, we duplicate the scaling algorithm for each of the various combinations, leading to sixteen versions of the same code, each with minor variations. This is unaesthetic but fast.

Decimation/Replication does not produce the best-looking image scaling. One problem is that the pixels tend to ripple as the image is scaled. MIP-mapping and bi-linear or tri-linear filtering, such as is performed in hardware by high-end 3D cards, produces much smoother results. In the future, this style of hardware is likely to become readily available on home PC's. Hopefully, simple 2D image scaling operations will also be supported in hardware by these cards.

Rotation

To save the overhead of adding a general-purpose transformation to the renderer, we add special code to rotate each object type. This code off-loads rotation computation from the renderer to the time of rotation. Thus, the Pad++ renderer maintains a transformation stack of just translation and scale, but does not include rotation. Polygonal objects are rotated by transforming the points. Images are rotated by computing a new rotated image from the original. Text, however, is rotated at render time, and thus rotated text is slightly slower than non-rotated text.

Screen Management

Consider a 1280x1024 pixel screen. If the user is inserting a single text character in a small text object visible on the screen, it is unacceptable to redraw the entire screen-full of information for just this minor change. Rather than redrawing the entire display, it is better to focus rendering resources on only the areas of the display that have changed. To do this, applications must maintain information about what areas of their display are out of date and perform rendering operations clipped to these areas. We call this technique screen management.

Existing windowing systems do determine when windows need refreshing (e.g. because they have been moved or resized), and they also clip rendering operations to windows (so changes to one window do not effect other windows). In this sense, windowing systems carry out primitive screen management on behalf of applications. But windows are assumed to be rectangular and opaque (X Windows includes an extension to support non-rectangular windows, but there is a performance penalty for using it), and windowing systems still leave it up to the application to manage the contents of each window. To handle large windows effectively, applications must implement their own screen management schemes.

To help applications carry out screen management, all the common windowing systems also provide clipping primitives that use Shape algebra [19], which lets applications clip 2D graphic operations to arbitrary polygonal shapes. Shape algebra is useful because it supports efficient clipping and boolean operations. Using Shape algebra, applications can implement sophisticated screen management schemes.

In a ZUI such as Pad++, screen management is further complicated by three factors. Firstly, objects can be non-rectangular and semi-transparent, so a change to an object may require re-rendering the objects in front of and behind it. Secondly, rendering is double-buffered, so objects can never just directly modify pixels on the screen, but must coordinate with the double-buffering mechanism. Thirdly, because of portals, a change to an object may require several different parts of the screen to be redrawn.

Pad++ uses shape algebra in conjunction with a "damage and restoration" scheme to keep the display up to date.

Damage and Restoration

The Pad++ screen management system is based on a painting metaphor. The display is treated as a "painting", and changes to objects visible on the display are seen as "damage" to the painting. A "restorer" is an object whose contract is to fix up damage. Each restorer maintains a Shape specifying what area of the display to work on, and an integer level indicating what refinement level to work at (more on refinement in a moment). At any time, a number of restorers may be working on different areas of the painting, each at different refinement levels. The system ensures that no two restorers are working on the same portion of the screen – newer restorers always receive priority over older restorers (their Shape is subtracted from the Shape stored in earlier restorers).

When objects change, they register damage on the Pad++ surface. Objects register damage in object coordinates by calling the *Damage* routine, passing it a bounding box. When damage occurs, Pad++ searches for a restorer with refinement level 0 to register the damage with, creating a new restorer if necessary. Once a restorer is obtained, Pad++ adds the bounding box to the Shape record within the restorer. In principle, it is desirable to maintain damage information in floating point surface coordinates. In practice, because Pad++ uses the Shape algebra utilities provided by the underlying windowing system, damage coordinates are always converted to integer screen coordinates before being added to the Shape.

Notice that some changes to objects may generate multiple calls to the Damage routine – for example, when an object is moved, it generates damage for both the old and the new bounding box of the object.

When the system becomes idle (when all the events in the input queue have been handled), any restorers in the pending queue are removed from the pending queue and "run" – causing the appropriate regions of the display to be re-rendered. Applications can also force pending restorers to be rendered, for example during animations.

Damage Through Portals

Portals complicate damage, since a single object may be visible on the screen in multiple portals at difference scales. The system must also handle the case of damage to an object which is visible through a portal looking at a portal looking at the object (and so on). Damage to objects visible through portals must be clipped correctly to the portal's screen bounds.

A simple approach is to perform an exhaustive depth-first search every time an object is damaged, checking each portal to see if the damage is visible. This is too slow in practice, and doesn't scale up to support large numbers of portals. (Notice that the damage recording mechanism must be fast. During an animation, many objects may be changing every frame, generating large numbers of damage requests. If every damage request involves a large search operation, frame rate will drop significantly).

A better approach is to maintain a data structure indicating, for each object, which portals can see that object. The damage routine can then just register damage with each portal directly. A problem with this approach is that maintaining the per-object portal lists is expensive. Every time an object is moved, the system must examine all of the portals to check if the object is visible (directly or indirectly) in the portal. The data structure must also be updated every time a portal's view is changed.

A good compromise is to have each view (i.e. each portal or top-level window) maintain a list of the portals visible within that view. Then, when damage occurs, the system can quickly determine which portals are visible on the screen and recursively check only those portals for damage. The visibility list is comparatively lightweight to maintain, since it only involves work when a view is changed, and not when other objects are changed.

A further optimization would be to divide portals into high-priority and low-priority portals. High-priority portals are used for areas which the user can actually interact with (main displays, editor windows, lenses, etc.), and which must be kept up-to-date. Low-priority portals are used for static non-interactive information displays (bookmarks, icons, snapshots, etc.). Whenever damage occurs, all the visible low-priority portals are scheduled for repainting after a time-out of one second, regardless of the visibility of the specific object in those portals. High-priority portals register damage recursively as described above. With this mechanism, high-priority portals are always kept up-to-date (but require more work to damage). Low-priority portals can become out-of-date, but they incur a smaller penalty during animations. We are currently implementing this last optimization.

Sticky Objects

In a ZUI, it is sometimes important to have objects stay in a fixed location on the screen, and do not pan and zoom with other objects on the surface. These *sticky* objects are useful for bookmarks, cut buffers, status lines and other interface components. How should these sticky objects be handled?

Ideally, a hardware solution such as overlay planes (found, for example, on SGI workstations) would be used for sticky objects. Overlay planes let two distinct frame buffers be mapped to a single window. The hardware includes a mechanism for indicating which buffer is visible in each pixel location (e.g. via a special transparent pixel value).

Overlay planes would let Pad++ treat sticky objects and non-sticky objects as two separate planes of information. However, not all systems support overlay planes, and overlay planes prevent sticky objects from being beneath or interleaved with non-sticky objects.

A second approach to handle sticky objects is to maintain two separate scene graphs for each surface – one representing non-sticky objects, the other representing sticky objects. The renderer and event manager must then consult both graphs. The drawback of this approach is that it introduces complexity into the code, since functions must be aware of the two different cases (sticky and zoomable). We tried this approach, but abandoned it once we realized how many duplications this would introduce to the code.

Our current approach is to implement sticky objects as normal Pad++ objects with a simple one-way constraint: whenever the main view of a surface is changed, all the sticky objects on the surface are transformed by the inverse of the view's change. Since the number of sticky objects is generally small (limited screen real-estate offers a natural incentive to keep the number of sticky objects down), this does not impact hugely on the cost of view changes. To the user, the effect is that all sticky objects maintain a fixed location on the screen. Internally, sticky objects are maintained in the same scene graph as non-sticky objects.

An advantage of this approach is that it allows objects to be sticky only in one dimension – e.g. objects that pan left and right, but always stay the same size (we call these *sticky-z* objects). The notion of constraints makes it easy to customize the specific behavior of the sticky object.

Level of Refinement

During pans and zooms, the main role of the display in a ZUI is to provide information as quickly as possible to orient the user – giving them visual cues indicating where in the space they are looking. But when the view is stationary (e.g. because the user is reading a document), text and diagrams should be as legible as possible. These two different modes of viewing introduce a design conflict, since producing legible text at arbitrary scales is a compute-intensive operation, and during pans and zooms you want to spend as little computation time as possible.

One solution to this problem is to render the contents of the display using fast low-quality algorithms during animated pans and zooms, and then use higher quality rendering algorithms to redraw the display when it has been stationary for a period of time.

To support this approach, restorers in Pad++ can work at different *levels of refinement (LOR).* At LOR 0, the restorer draws objects as quickly as possible. At higher LORs, restorers draw with increasing detail, although they take longer to complete. In practice, only LORs 0, 1 and 2 are used.

When a restorer has finished work, it can schedule another restorer to redraw the same region at a higher LOR – allowing areas of the display to be "refined" (see Figure 3). Restorers determine whether refinement is needed by querying each rendered object to see if it has more detail available. If an object has further detail available, its bounding box is added to a "refining restorer". At the end of the render, the refining restorer is scheduled to run. With this mechanism, a change to an object on one area of the screen may cause that area to iterate through a number of refinements. Changes to other areas of the screen do not interfere with this refinement process, but instead initiate new refinement processes. Damage to areas that overlap with a refining restorer cause that area to be removed from the shape managed by the refining restorer and rescheduled for rendering by a new restorer using LOR 0, potentially canceling the refining restorer if it becomes empty.

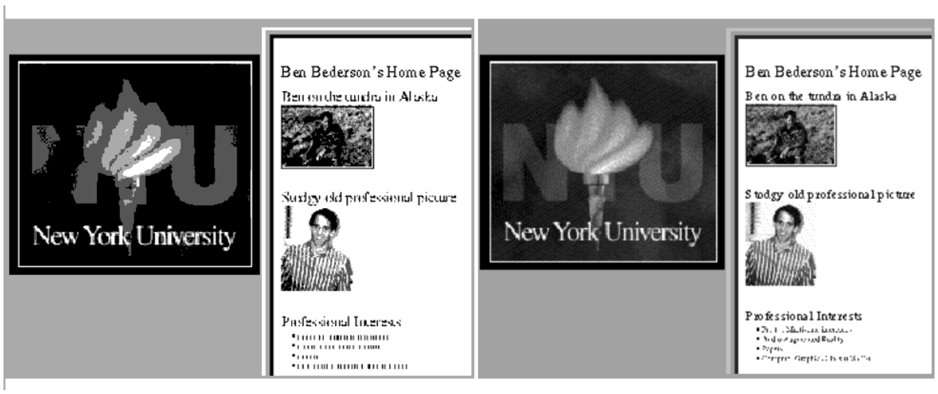

Figure 3: Before and after refinement. The left view shows the scene before refinement. The right view shows the same scene after refinement. Greeked text is replaced with fully rendered text. Images are refined using color dithering. Refinement is introduced using a dissolve effect, to reduce screen flashes

Refinements can be visually jarring to the user. For example, when switching from LOR 0 to LOR 1, small text alters its appearance from dashed lines ("greeked" text) to individual letters. The change in appearance can be distracting. To resolve this problem, Pad++ uses a dissolve transition to update areas of the screen as they are refined. Although this effect slows down the total time taken to refine a region, it also reduces screen flashing, so the overall impression is an improvement.

Interruption

Adding refinement to a system introduces a new problem. Refinements may take a considerable time to complete. For example, rendering a large image using color dithering takes nearly a second on a mid-range PC. During this time the user may decide to interact with one of the objects on the surface, or pan to a different location. Forcing the user to wait for refinements to complete is undesirable.

In Pad++, refinements are interruptible. Periodically during a refining render, the system checks for input events (keyboard or mouse button events). If an event is found on the input queue, the renderer finishes rendering the back buffer at LOR 0, leaves the front buffer unmodified, and schedules a new restorer to come back during idle time to redraw the region. Then the event is processed as normal.

Fast Panning

A second problem with refinement is that when the user pans or zooms the whole display, it jumps back to refinement level 0. This introduces a distracting flash to the display, although this is less distracting than the flash produced by successive refinements, since it is user-initiated, and not initiated by the system during idle time. Still, considering that the display may contain an entire page of text rendered at a high refinement level, it is desirable to preserve this image at high refinement level when the user performs a small pan (e.g. to read the bottom of the page).

Using a BitBlt operation, it is comparatively fast to shift the contents of the back buffer by a number of pixels. Then restorers can be scheduled to redraw the strips at the edges of the back buffer that are out of date. With this approach, much of the display contents can be retained for incremental panning operations. A secondary advantage is that, for complex scenes, there is a performance improvement (since only the objects in the exposed strips need to be rendered, and not the whole display).

There are three complications to consider. Firstly, if there are any sticky objects, the areas under the sticky objects must be damaged both before and after the BitBlt, so that the sticky objects are rendered in the new location. Secondly, any outstanding restorers must also have their Shape records updated to reflect the new screen coordinate system. Thirdly, no matter how careful we are, there will inevitably be some aliasing problems. For example, consider a pan by 1.1 “pixels” to the left. This is a legal operation in Pad++, which uses a floating point coordinate system. BitBlts, on the other hand, always move data a whole number of pixels. After ten successive pans by 1.1 pixels, the objects on the screen will have moved 10 pixels, whereas Pad will think they have moved 11. Tearing and other visual artifacts will become apparent.

To address this problem, after performing any BitBlt operations on the screen, Pad++ schedules a restorer to redraw the entire screen after one second of idle time.

Part II: Visible Object Determination

Having a powerful rendering engine is important, but rendering resources are always finite. For datasets containing only a few dozen objects this is unlikely to be a problem, but for datasets containing tens of thousands of objects a more intelligent approach is needed.

A basic implementation of a ZUI would simply draw every object every frame, and rely on the clipping mechanisms provided in the graphics hardware to avoid modifying pixels that are outside the current clipping region. However, clipping is compute-intensive, and as the number of objects increases this approach becomes too slow. For a ZUI that must scale up to handle tens of thousands of objects, the approach is unworkable.

A slightly better approach is to test each object for visibility, and only draw objects that are visible in the current view. This way, if an object is not within the current view (or outside the current clip region), it does not consume any rendering resources. In most applications only a small fraction of the total objects in a world are visible at a usable size in a given view, so this approach can significantly improve frame rates. Frame rates also improve if only a small fraction of a view needs to be redrawn.

Bounding boxes are often used to perform these coarse visibility checks. Testing if two bounding boxes intersect is cheap. A Pentium can performs hundreds of thousands of 2D bounding box intersection tests a second. So for medium sized datasets, performing bounding box checking alone is enough to yield good frame rates. Tabula Rasa [12] and the original Pad implementation [6] both use this approach. However, since the time spent doing visibility checks is linearly proportional to the number of objects in the world, as the number of objects increases, there will be a point where the system spends more time checking than rendering, and frame rates drop.

For very large datasets, a faster mechanism for determining which objects are visible within a given view is essential. Once such a mechanism is available, it is also useful for event processing, since determining which objects lie underneath a point is essentially a constrained version of the same type of query.

Common Approaches

There are several popular approaches to handle visible object determination. Windowing systems typically only manage a relatively small number of windows, and so don't need special purpose mechanisms for performing visibility tests. Windowing systems are also hierarchical by nature – top-level windows enclose sub-windows, which enclose sub-sub-windows, and so on. This enforced segregation means that windowing systems can quickly eliminate whole trees from consideration, further speeding up visibility checks. By comparison, ZUIs need to support datasets in which there is no rigorous enclosure hierarchy – such as you might find in a map.

ZUIs are more akin to 3D systems, in which objects can be distributed anywhere throughout space. The distribution of objects within space depends on the specific application, but there are several classes of applications that are instructive to look at. Two representative types of 3D systems are vehicle simulators and architectural walkthrough systems. Vehicle simulators typically have strict frame-rate requirements, and have objects distributed fairly uniformly through space, frequently lying on a large two-dimensional surface [20]. Architectural walkthrough systems on the other hand, tend to have a large number of objects concentrated in a small three-dimensional space [21]. In most 3D systems, there can be a fairly wide range of sizes of objects, as both large structures and fine details must be represented.

Object visibility in 3D systems is determined by what is called the *viewing frustum* [22]. This is the truncated pyramid of space that can be seen from the current view position. It is truncated by *near* and *far* clipping planes – so objects that are too near or too far from the view are clipped. Often, some kind of spatial indexing data structure is used to efficiently determine which objects overlap the viewing frustum. In addition, some systems use knowledge about the model to eliminate objects from consideration. For instance, in an architectural walkthrough of a room on one floor, no objects on any other floors are visible, and only a limited number of objects in other rooms on the same floor are visible [23].

Other 2D systems that maintain large numbers of objects and must quickly determine object visibility include integrated circuit design systems (VLSI) and geographical information systems (GIS). Both VLSI and GIS systems manage very large numbers of 2D objects and support limited forms of zooming. VLSI and GIS systems also contain objects with a wide range of sizes.

The visible-object determination needs of a ZUI are similar to these 2D and 3D systems. In all cases, the basic problem is to quickly determine the set of objects that overlap a specified view. The differences between 2D and 3D approaches to this problem are not great since most spatial indexing data structures can accommodate scenes of arbitrary dimensionality.

However, ZUIs impose two new constraints on the visible object retrieval system: small objects should be eliminated quickly, and objects must somehow be sorted by display-list order. We discuss these two issues below, comparing them with related issued in other graphical systems. Afterwards, we describe how spatial indexing is handled in Pad++.

Object Ordering

In a ZUI, objects need to be rendered in a specific order – this order determines which objects are in front of other objects, and which are partially obscured or behind other objects.

Unfortunately, spatial indexing data structures do not return the objects specified by a query in a guaranteed order. This is not a problem for 3D systems, which can render objects in any order and rely on a depth buffer (or "Z-buffer") to perform hidden surface removal. Order is also not a problem for GIS and VLSI systems because those systems are designed to avoid display order requirements through the use of *layers*. In these systems, objects of different types are put on different logical layers, and each layer is rendered in its entirety before any objects of another layer are rendered. Objects within a layer should not overlap because there is no ability to control the rendering order of objects within a layer. For instance, in most GIS systems, streets may appear on one layer with highways on another layer, and county boundaries on a third layer. When streets do overlap each other, the order in which they are rendered can not be controlled.

Two-dimensional windowing systems do require that windows get rendered with specified overlapping, but they do not use a spatial index. Rather, they maintain regions for each window specifying what portion of the window is not occluded, and only that portion of the window is rendered when the window contents change – thus maintaining the correct overlapping of windows on the screen. This approach is too costly when there are many thousands of objects, and doesn't handle semi-transparent objects well.

Small Object Elimination

Objects in a ZUI can differ in size by many orders of magnitude. In a zoomed out view, only the very large objects are visible, and many objects will be so small that they are less than a pixel in size. Spending time rendering these objects is wasteful, especially if rendering time is limited. Ideally, the visible object determination mechanism should quickly eliminate these small objects.

This requirement is similar for some other systems, but not identical. Windowing systems have no such requirement – they render every window unless it is occluded. VLSI and GIS systems do avoid rendering small objects, but as with sorting, use layers to avoid a per object solution. That is, rather than deciding whether or not to render each object by size, VLSI and GIS systems decide whether to render an entire layer depending on the magnification.

3D systems usually base culling efforts on object position rather than on relative size. Instead of culling objects that are in the current view but are very small, they are often optimized to eliminate objects that are positioned beyond the viewing frustum. Small objects that are within the viewing frustum still get rendered, or at best are checked on an individual basis. This is not sufficient for ZUIs.

Spatial Indexing in Pad++

Spatial indexing is the general term for a data structure that encodes spatial characteristics about a set of objects in n-dimensional space. These spatial characteristics are then used to provide efficient mechanisms for retrieving objects given spatial queries. One typical query is to retrieve all objects intersecting a given rectangle (for a two dimensional index.) This query can be used in a ZUI to return all objects visible in a given view, or to find all objects that overlap a single point in order to process events.

There are several widely used spatial indexing algorithms (A survey of spatial indexing algorithms and their applications can be found in [24][25]). Many are hierarchical and are based on partitioning space in smaller and smaller segments. Algorithms include R-trees [26], MX-CIF quad-trees [27], binary space partitioning trees [28] and k-d trees [29]. Each algorithm has maintenance (insertion and deletion) methods and query methods. Typically, there is a trade-off in time between these methods where increasing the efficiency of one introduces a higher cost for the other. Consequently, the choice of algorithm depends partially on whether the scene consists of static or dynamic objects. For static scenes, a higher maintenance cost is usually acceptable, whereas for dynamic scenes the maintenance costs become more critical.

In implementing Pad++, we chose to use an R-tree. An R-tree is a hierarchical structure based on *bounding regions*. Objects are contained in leaf nodes, and internal nodes contain regions that specify the bounds of its children. Each region has a specified minimum and maximum number of children. Regions can overlap, so while an object is only a member of a single region, a query can require looking at several regions at each level. R-trees are similar to quad-trees except that they partition space into regions based on the objects within the space, and not based on fixed-sized grids. R-trees are balanced so they have a guaranteed maximum depth.

We chose to use R-trees rather than the more advanced R*-tree [30]. R*-trees produce more efficient structuring of regions, but they take longer to maintain. Since visible object determination is not the biggest factor determining rendering speeds (see *Timing Results* below), we decided to choose an algorithm which offers effective indexing and very fast insertion and deletion times. This supports our goal to handle very dynamic data spaces, in which many objects move and resize over time. A basic R-tree appears to have lower maintenance costs than the other algorithms we examined. In addition, R-trees are amenable to efficient small object culling during queries.

Object Elimination in Pad++

As we mentioned earlier, a ZUI typically has overlapping objects with sizes that differ by several orders of magnitude, so it is crucial to quickly eliminate objects that are much smaller than the query window. For Pad++, we chose to ignore all objects that are less than one pixel in both screen dimensions. We can not eliminate very large objects because even if they are huge, they may still get rendered and have a large visual effect. Since the hierarchy within R-trees is based on enclosure, we are guaranteed that every child of a node is no larger than the parent node. Thus, if we determine that a given node is smaller than a minimum size, we do not have to descend that portion of the tree. In our implementation, in addition to the rectangle we pass in for each query, we also pass in a minimum size. All queries use this extra parameter to cull sub-trees that are too small during a search.

Even eliminating all non-visible and small objects may not be enough. If the system is performing an animated zoom and attempting to maintain at least ten frames a second, what happens when the number of visible objects in a scene is too high, and there is not enough time to render every object?

A general approach is to render only a partial or fixed set of visible objects during animation frames, and then render the full set of objects during refinements. For example, the system can avoid rendering text labels during animated view changes, and only render them when the view remains stationary for a while. Alternatively, it could render only the first few hundred objects during an animation.

A drawback of this approach is that it doesn't adapt according to the hardware or data. Even for simple scenes containing only a few objects, the user is given an impoverished presentation of the data during the animation. Animations don't scale according to the hardware – yet people with faster computers expect to see more detail during animations.

Instead of adopting a simple, fixed approach to controlling frame rates, a better solution is to include adaptive mechanisms in the renderer. The idea is to monitor frame rates, and use the statistics about prior frame rates to control how much work is done during the current render.

There are several things that can potentially be controlled. The renderer can make decisions about what objects are rendered, culling more objects when frame rates are low. The renderer can also switch to lower-quality rendering algorithms (e.g. algorithms which have a lower level-of-detail) when frame rates drop.

In Pad++, in addition to using a spatial index to eliminate small and non-visible objects, we use frame rate statistics to control how text is rendered and how many small objects are rendered. If frame rates are low, the number of “small” objects (objects beneath a certain pixel size in their largest dimension) rendered each frame is reduced and the notion of what constitutes a “small” object is increased to be more inclusive. Similarly, if frame rates are low, the system switches to using “greeked” fonts for small text. If, during any frame, a time-limit is exceeded, the system can switch to an emergency mechanism – this renders the remaining large objects using the lowest level-of-detail possible.

Object Ordering in Pad++

As described earlier, a query must not only retrieve the objects that overlap the specified rectangle, but it must retrieve those objects in display-list order, so they can be drawn with the correct overlapping behavior.

The easiest solution to this problem (and the one adopted by Pad++) is to simply retrieve the objects from the spatial index unsorted, and then apply a sorting algorithm (e.g. QuickSort). However, sorting can become quite expensive if a large number of objects are visible. Figure 4 shows the time to retrieve and sort objects in a scene depicting a map of New York City. In this case with a scene of over 2,000 objects, the sorting time took longer than the retrieval when the number of objects to be sorted became greater than about 400.

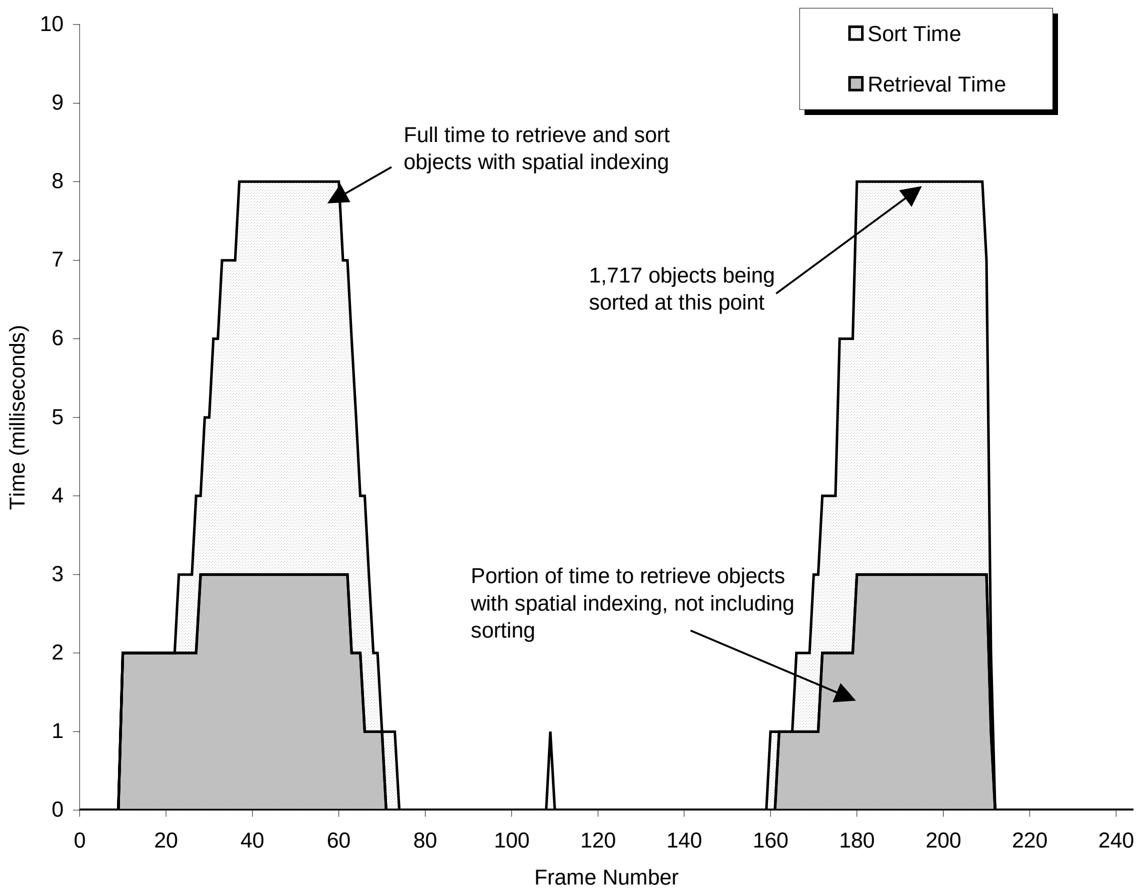

Figure 4: Time taken to retrieve and sort visible objects in a scene depicting a map of New York City containing 2,135 objects.

We attempted to reduce this sorting time by maintaining the list of objects in each R-tree leaf node in sorted order. We then performed a merge sort motivated by the fact that we had already done a fair amount of the sorting work since the objects at each node were sorted. Unfortunately, this approach did not result in sorting any faster than using the system QuickSort function on all the visible objects. It appears that the overhead of copying objects and maintaining extra lists required for a merge-sort consumes more time than is saved.

A hardware solution to this problem exists: using a depth buffer eliminates the need for sorting. Depth buffers are commonly found on many 3D cards, and also on our hypothetical ZGA card. To render the ZUI scene, the depth buffer is first cleared, then objects are drawn in arbitrary order but with a Z-depth corresponding to their drawing order number. The graphics hardware then takes care of clipping objects according to their depth. For this to work well, the depth buffer must have at least 16 bits per pixel to offer a reasonable number of depths.

Compromise solutions also exist. For example, given that the number of overlapping objects in any scene is likely to be much less than the total number of objects in the world, we could provide for only a fixed number of different drawing depths (perhaps a few thousand). Objects with the same depth would be rendered in an arbitrary order – for objects that don't overlap this is not a problem. Objects with different depths are rendered in order according to their depth, with lower depth numbers being rendered first. This lets users specify overlapping constraints when they are important.

Another possibility is to use binary space partition (BSP) trees [28] with the drawing order of each object representing its Z depth. This approach has the advantage of eliminating the sort entirely as BSP trees implicitly sort objects. However, maintaining BSP trees for dynamic scenes tends to be compute intensive.

We are currently experimenting with an indexed approach, which eliminates the need for sorting and is still reasonably efficient. We construct an array of flags, one flag per object. The flags are maintained in pages, each page containing 256 flags, as well as a *used* bit that is set whenever one of the flags in the page is set. The retrieval algorithm clears all the used bits, and then invokes the R-Tree spatial index described earlier to locate visible objects. For each visible object, the corresponding flag in the array is set. Whenever a page of flags is referenced for the first time, the page is marked as used and the flags in the page are cleared to zero. The rendering routine then steps through each of the pages. If a page is marked as used, then all the flags in the page are checked, and each object whose flag is set is rendered.

With this paging approach, if every object on a surface is visible the algorithm has a worst case running time of O(N), where N is the number of objects on the surface. This is much better than the QuickSort approach which is O(NlogN). If no objects are visible, the paging approach still has a performance overhead of O(N), since every used bit must be checked. But the costs involved are very low since there is only a single used bit to check for each group of 256 objects. By paging the used bits themselves, the overheads can be reduced still further, leading to good overall performance. The timing tests in the next section do not reflect this new approach which is still under development.

Timing Results

In this section we present the results of several timing tests of Pad++, showing the performance of rendering and visible object determination. Remember that the goal is frame times of at most 100 milliseconds (ms), or 10 frames per second (fps). All tests were performed on a 200 MHz Pentium Pro running Linux, with an 800x600 pixel graphics window. Window size is significant because each frame render involves both clearing the back buffer to the background color, and copying the back buffer to the window. The overhead for this clearing and copying on the reference platform is 4 ms.

Four different tests were performed, to compare hand-created scenes with computer-generated scenes, and to compare scenes of uniformly sized objects with scenes of objects of many different sizes. See the graphs in Figures 5–8.

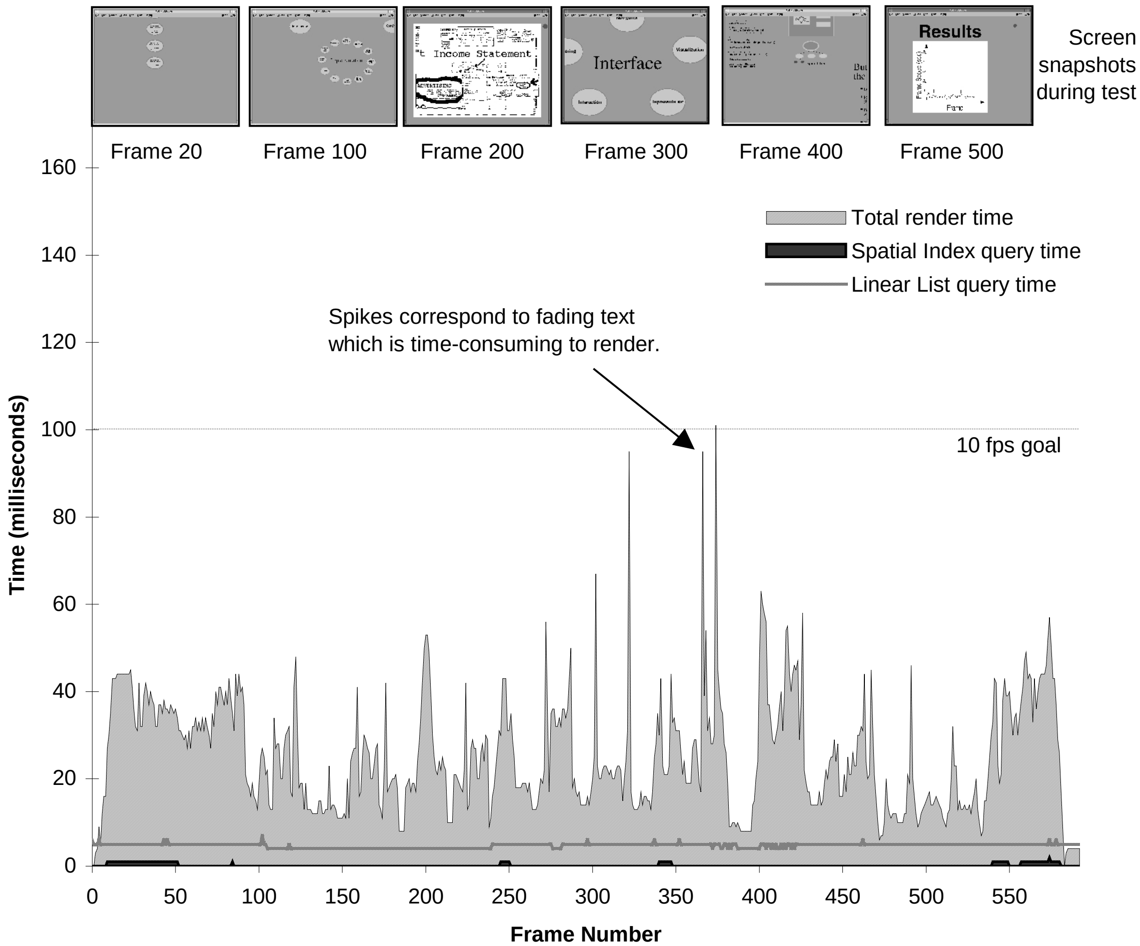

Figure 5: Times for 20 copies of a hand-created scene of a typical Pad++ document. Objects have very different sizes, and only a small number are visible at any given time. Total of 11,620 objects.

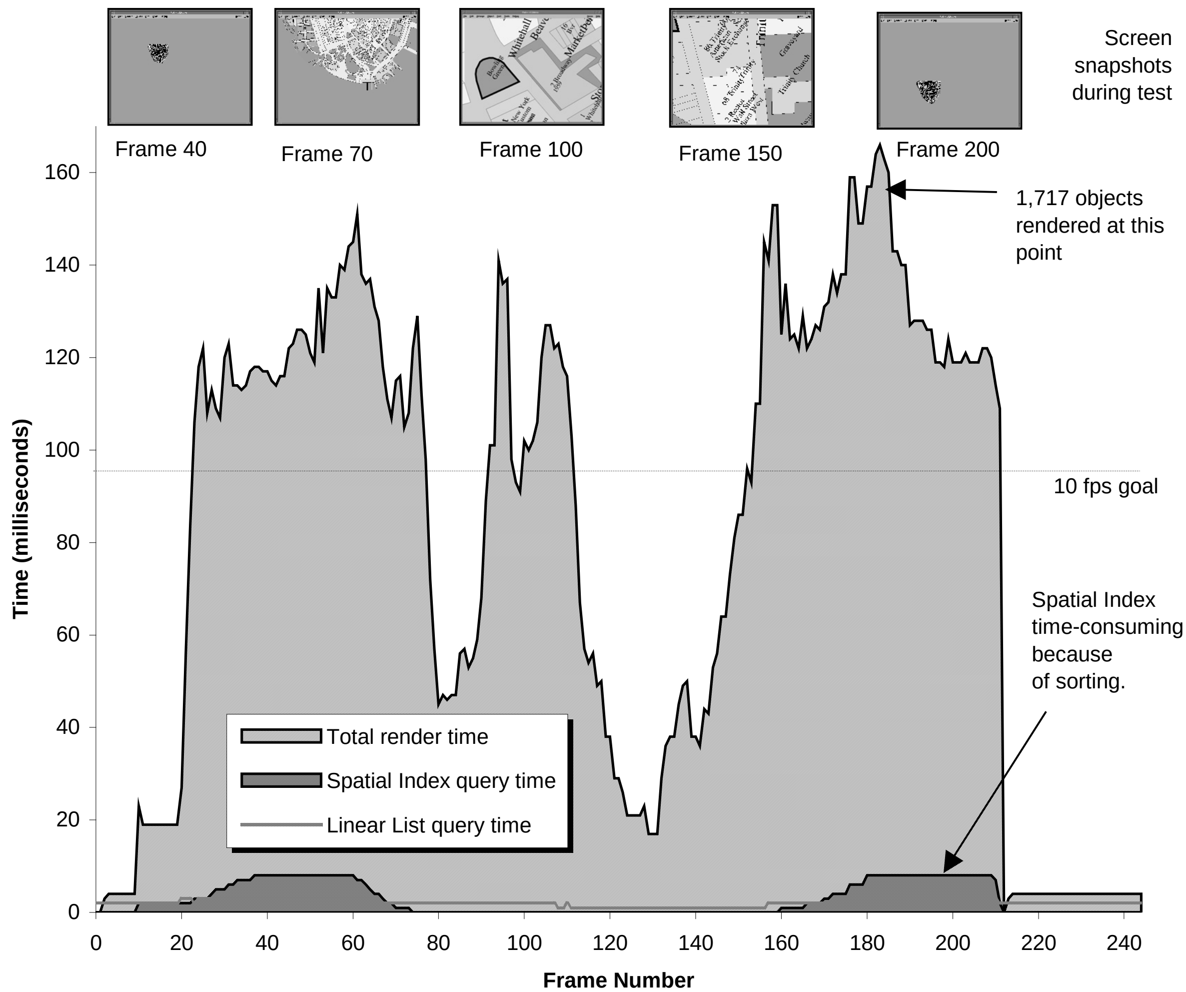

Figure 6: Times for hand-created scene of 2,135 objects where all objects are of similar sizes. The scene depicts a map of lower Manhattan.

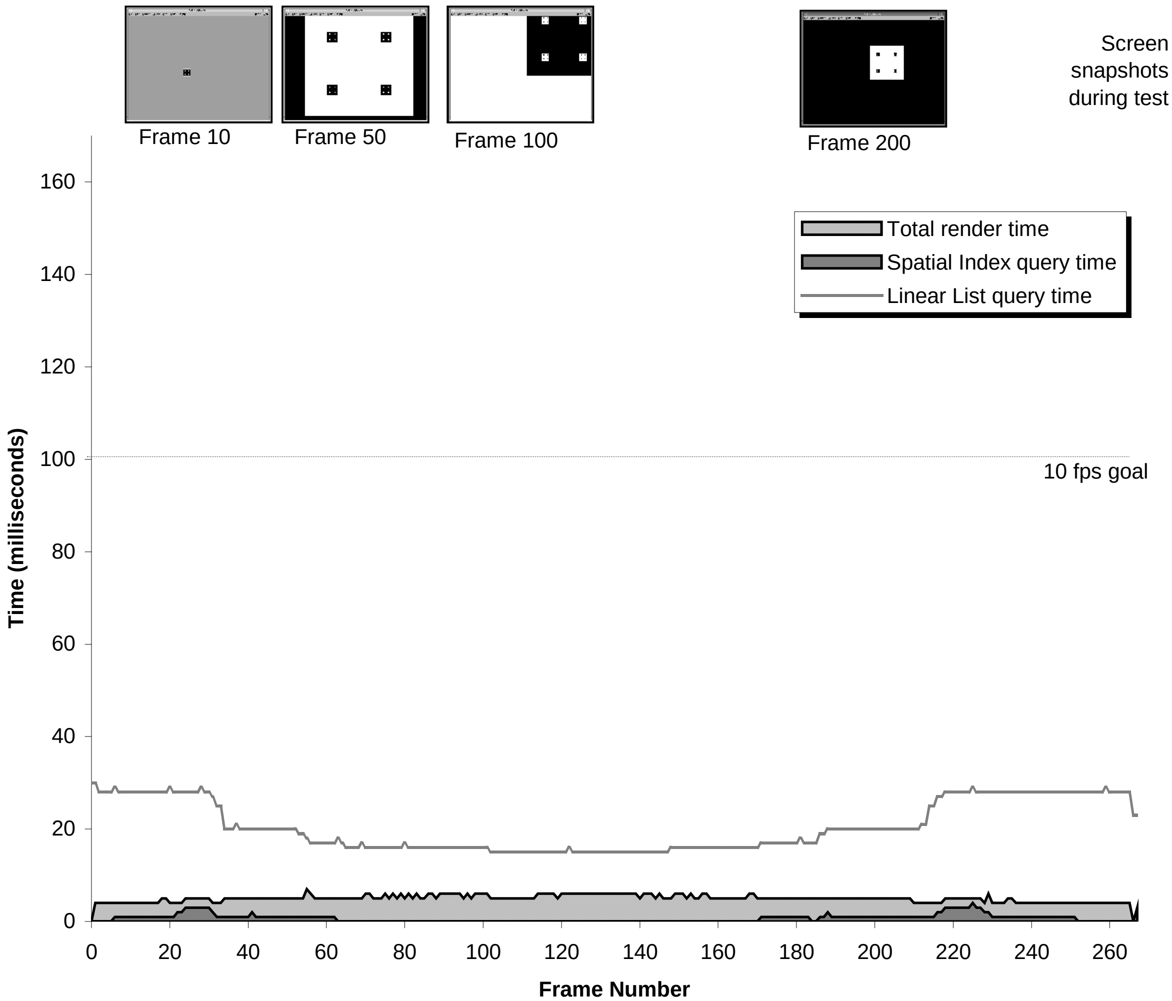

Figure 7: Times for a computer-generated scene of 21,849 objects with significant size differences between objects. The scene contains 7 levels of nested rectangle where each rectangle contains 4 others.

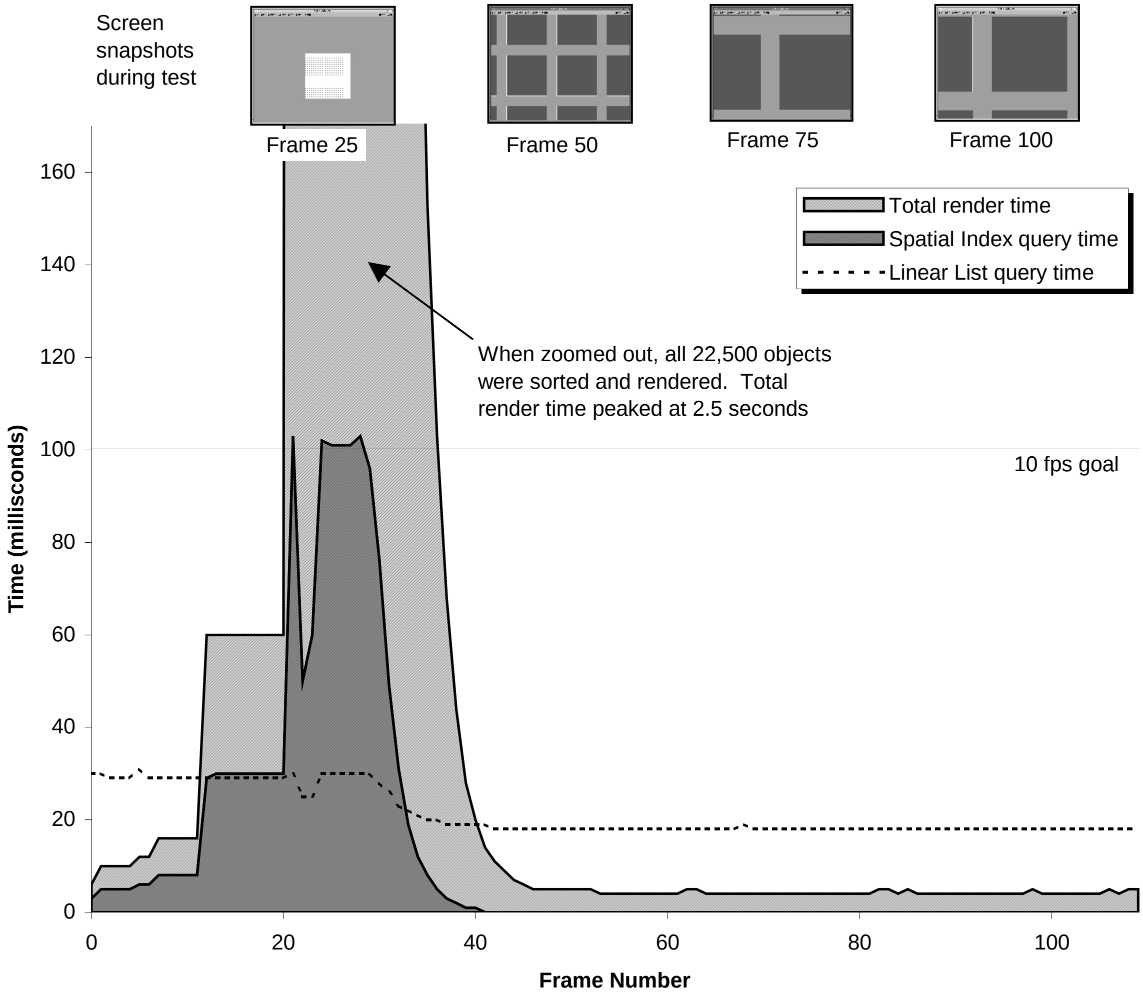

Figure 8: Times for a computer-generated scene of 22,500 objects with no size differences between objects. The scene contains an array of 150x150 identical rectangles. When zoomed out, all 22,500 rectangles are visible, and thus rendering times grow very high.

Each graph shows the total rendering time for each frame as the view is changed to move through all parts of the scene, zooming in and out, and panning to each area of the scene. The total rendering time is further broken into two parts: time spent querying the spatial index (and sorting the results), and time spent actually doing the drawing.

Each graph also shows the time it takes to perform a simple linear visible object check (i.e. checking every object for visibility every frame). This lets us compare the effectiveness of the spatial index with a simple linear approach.

Intuitively, the time taken by the linear approach should be constant, since every object is checked in every frame. In practice, for visible objects the system also performs small-object culling, so the total time taken in each frame varies slightly according to the number of visible objects (i.e. visible objects take two checks, whereas objects that are not in the view require only one).

To evaluate the cost of maintaining the spatial index, we performed a test where we moved many objects at once. For the scenes used by Figures 7 and 8, we moved all the objects, with and without spatial indexing. The timing results are shown in Table 1.

	Number of Objects	**Linear List** Time to move every object (seconds)	**Spatial Index** Time to move every object (seconds)	**Cost Ratio** Spatial Index : Linear List
Scene from Figure 7	21,849	32	53	1.66
Scene from Figure 8	22,500	38	43	1.13

Table 1: Time taken to move many objects, comparing linear list approach with spatial index approach.

We also tested the time it takes to build the index in the first place. For the first scene, building the index took 1.75 seconds, so about 12,485 objects can be inserted a second. The second scene, inserting objects into the index took 1.82 seconds, or 12,363 objects per second.

Analysis

These results collectively show that we are very close to meeting our goals for typical scenes of a ZUI. Many scenes met our frame rate goal of 10 fps, and the spatial index is effective. For most scenes, the spatial index is no slower than a simple linked list, and for scenes with many objects where only a small percentage of them are visible, the spatial index cuts down on visible object determination time substantially. For all scenes, the time spent detecting visible objects is a small fraction of the time spent actually drawing.

Regarding maintenance efficiency, Table 1 shows that maintaining the spatial index does impact object movement speed, but by a factor which is less than 2. With spatial indexing, we can move an average of 461 objects per second. With no indexing, we can move 633 objects per second (we suspect that rewriting the test in C instead of Tcl would increase both of these numbers substantially).

The one type of scene for which our implementation does not meet our rendering goals is where there are many objects of similar size. When zoomed out to a certain point, all of these objects become visible and the spatial index spends a lot of time sorting these objects, and the renderer spends even more time rendering all of these objects. Thus, our current implementation of Pad++ is effective for scenes that contain many objects at different sizes, but is not effective for scenes with many objects at similar sizes.

Part III: Pad++ Structure

Pad++ is an object-oriented graphical interface library which implements a Zooming User Interface (ZUI) as previously described. It is object-oriented in both its implementation and its use. All graphical elements in Pad++ are instances of objects and share certain functionality, such as the ability to be moved and resized.

This section describes the structure of Pad++, and how its different components are designed and implemented. We discuss the object hierarchy, and the implementation of unusual features.

Structure

Pad++ is implemented in C++, and provides application interfaces for several languages, including C++, Tcl, Scheme, Perl, and KPL (an in-house reverse-polish language for rendering). In addition, an interface to Java is under development. Most applications to date have been written in Tcl as that is the language we originally targeted, though we have also developed a procedural animation engine that runs within Pad++ and uses KPL [31]. While Pad++ now runs on Windows 95/NT, we originally designed and implemented Pad++ for UNIX with the X window system. It runs on many versions of UNIX, including Linux, SunOS, Solaris, IRIX, and FreeBSD. The Windows 95/NT version of Pad++ maps X windowing system calls to equivalent functions under Windows. Since in some cases this mapping is not straightforward, the Windows version is currently not as fast as the UNIX version. We are working on a renderer that uses the Windows features more directly.

One continuing goal has been to make Pad++ as portable as possible. Because people download, compile, and run Pad++ on all kinds of systems all around the world, we decided to use only the most commonly available and most reliable features of C++. For this reason, we decided not to use multiple inheritance or templates. In some places, this made the code more complicated, but we feel it was a necessary trade-off in order to minimize the time we spend supporting Pad++.

Pad++ consists of several components. At a high level, they are:

Renderer: The renderer performs all the rendering to the screen. It maintains a stack of transformations that specify translation and scale.

Event Handler: The event handler is responsible for processing all input events. It determines which objects receive events, it maps events through portals and it takes care of event grabbing (insuring that that mouse motion and release events go to the same object that received the associated press event.)

Surfaces: A surface represents a single flat data space where graphical objects exist. Objects can exist at any position and scale on the surface.

Views: Surfaces are mapped to the screen through views. A view specifies the position and magnification at which the associated surface is seen. There are currently two places where views are used: for windows and for portals. The extent of the surface that is seen is specified indirectly by the size of the associated window or portal.

Objects: Every graphical item on a Pad++ surface is derived from a base Object class. This class defines much of the behavior common to all objects. It controls where and at what size an object appears, the object's transparency, and the range of sizes at which it is visible. It also controls the object's stickiness and drawing order, as well as what layer it is on. Figure 9 shows the hierarchy of all objects deriving from this base Object class.

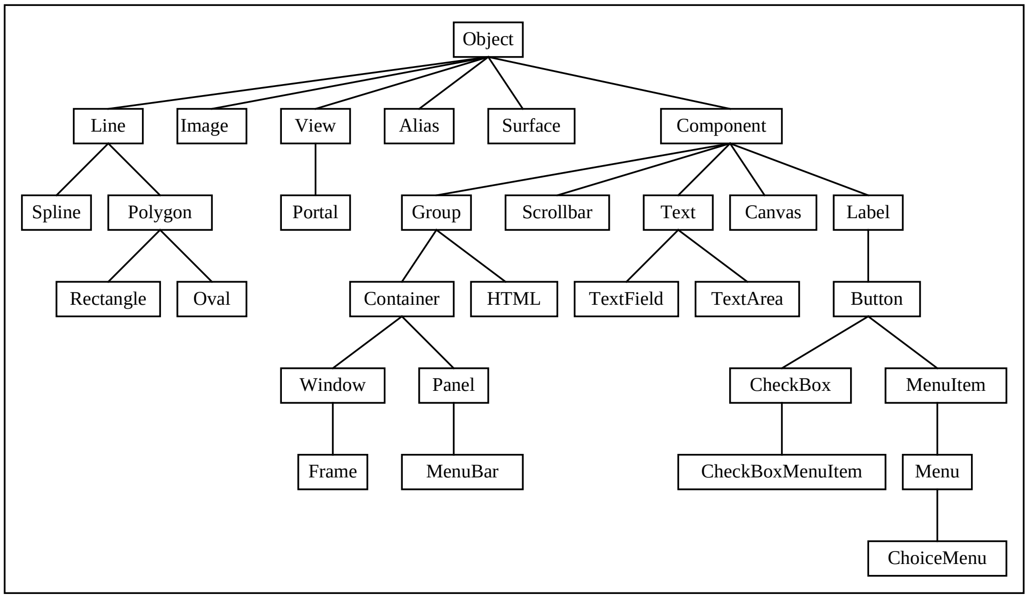

Figure 9: Pad++ Object Hierarchy.

This class hierarchy is somewhat complicated, partly because Pad++ is designed to be connected to both Tcl and to Java. We made a strong effort to be consistent with the programming and visual interface standards of both languages.

Surfaces, Views, and Portals

The Pad++ dataspace is structured around surfaces and views. A surface is a plane on which objects exist. The Pad++ environment supports multiple surfaces simultaneously. Each surface can be visible within top-level windows, or can be seen within portals. Objects exist on a surface within the surface's coordinate system. The Pad++ coordinate system uses a standard right-handed Cartesian coordinate system. The X axis increases to the right and the Y axis increases *up*. Note that this is different from many graphical windowing systems. All coordinates are specified in floating point units that by default correspond to the dimensions of a single pixel on the screen. Dimensions can also be specified in inches, millimeters, or points. This means that, when the top-level view is at a scale of 2.0, a line drawn on the surface from 0.0 to 100.0 is 200 pixels long.

Surfaces are mapped to the screen through views. Each surface that is mapped to a top-level window implicitly gets a view that controls what part of the surface is visible in that window. Views specify the visible portion of a surface with a point and a magnification. The point specifies the portion of the surface that will appear at the center of the view. The magnification specifies how much the surface should be enlarged or shrunk.

Portals are a special type of view. They exist on a Pad++ surface, and can be moved and resized like other objects on the surface, but they also specify a "lookon" – this specifies the surface that is visible through the portal. Portals can look onto the surface they themselves are on, or they can look onto any other Pad++ surface.

Portals are implemented with special rendering and event processing methods. A portal's render method sets the clipping region, and then renders the Portal's lookon surface, clipped by this region. A portal can render another portal within itself, but prevents recursive rendering of itself. Portals also process events specially. When an event hits a portal, if the event is on the portal's outer frame, the event is sent to the portal object itself. If the event is within the portal, the event is passed through to the surface the portal looks onto. When an object receives an event, it may query the event to determine the list of portals (if any) that it was passed through.

Sample Pad++ Code

Here is a short example of some Tcl code that gives a feel for how some of the simple features of Pad++ can be used. This example creates a surface, puts it within a top-level window, and then creates several objects with event bindings, and changes the view. In this example, '>' represents a UNIX prompt, and '%' represents a Tcl prompt. Commands typed in by the user appear in bold face. The resulting output is shown in Figure 10.

```
> padwish                ;# Start the Pad++ executable
% pad .pad               ;# Make a Pad++ surface
.pad
% pack .pad              ;# Map it to a top-level window with a view

                         ;# Create a rectangle. A unique integer is
                         ;# generated which is used to identify this
                         ;# rectangle in the future.
% .pad create rectangle 0 0 50 50 -fill red
3                        ;# System returns '3', uniquely identifying object

                         ;# Create some text and place it within rectangle
% .pad create text -text "Hello World!" -font "Times-12" -anchor sw \
     -position "10 10 1"
4                        ;# System returns '4', uniquely identifying object

                         ;# Create an image from a file whose bottom-left
                         ;# corner is at the same place as the
                         ;# top-right corner of the rectangle.
% .pad create image -image "pad.gif" -anchor sw -position "50 50 1"
5                        ;# System returns '5', uniquely identifying object

                         ;# Create event bindings so that moving the
                         ;# mouse over the rectangle highlights it,
                         ;# and clicking on the rectangle moves it
                         ;# 10 pixels to the right.
% .pad bind 3 <Enter> {%P itemconfig %O -pen blue}
% .pad bind 3 <Leave> {%P itemconfig %O -pen black}
% .pad bind 3 <ButtonPress-1> {%P slide %O 10 0}
                         ;# Zoom the view so that the center
                         ;# of the image is at the center of the
                         ;# view, and is 1.5 times larger than normal.
                         ;# The view change will be animated over a
                         ;# period of 1000 milliseconds.
% .pad moveto 200 150 1.5 1000
```

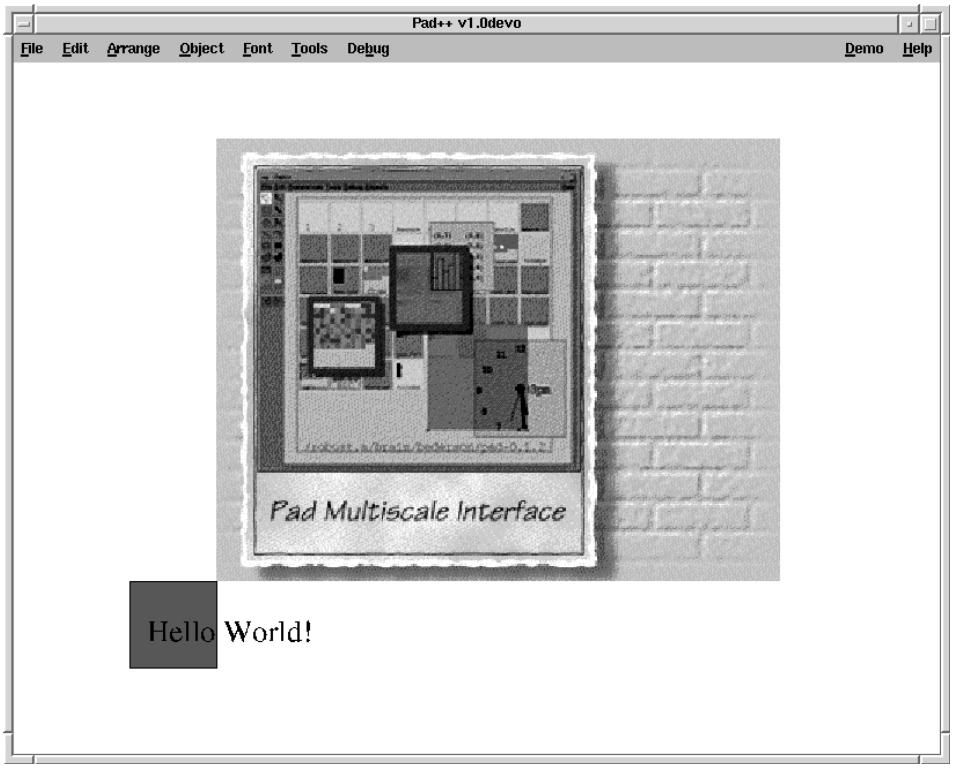

Figure 10: Snapshot of the Pad++ window after executing the Sample Pad++ code

Quantity of Zooming Space

Just how big is the zooming space of Pad++? Ideally, a ZUI would have unlimited space so objects could be put at any position and at any scale. While this might seem unnecessary, applications can easily use a lot of space. For example, a visualization system that depicted hierarchies through containment uses a lot of depth. If each level of the hierarchy were an order of magnitude smaller than the parent, dozens of orders of magnitude of zooming could quickly be used up. In addition to depth, the amount of breadth is also a concern. Just how far apart can objects be placed? Again, this may not seem crucial, but as soon as you zoom out, even objects that are very far apart are brought closer together visually. A user can very easily zoom out, pan a little, and then zoom in with the result that they've covered a very wide expanse of space. *Space-Scale Diagrams* provide an analytical tool that is useful for describing and analyzing objects in scale space [32].

In Pad++, the view and object coordinates are stored with standard 32 bit floats that store roughly 7 orders of magnitude of resolution (+/-). So, we expect 14 orders of magnitude for zooming and panning. Again, this may seem like a lot, but to go from a dot to full screen is roughly 3 orders of magnitude (on a 1,000 pixel screen). Pad++ can do that about 5 times.

Care must be taken when implementing coordinate transformations in a ZUI. Unfortunately, the most simple and intuitive approach to implementing transformations can result in reduced zooming space. To understand this, note that each view has an offset and a magnification, which are stored as a triplet `(xview, yview, zoom)`. Every object has a position and a scale `(xoffset, yoffset, scale)`. In addition, objects such as polygons and lines also have coordinates (x_1, y_1, …). To apply a coordinate transformation from object to screen coordinates, we start by applying the transform for the current view (in these examples, we show only the `x` coordinate):

```
-(xview * zoom)
```

Now, we apply the object's transformation, leading to the expression:

```
(xoffset * zoom) - (xview * zoom)
```

However, consider when you zoom in and pan off to the side so that `xview` and `zoom` both become large, the quantity `(xview * zoom)` can overflow quickly. This results in a reduced zooming space – that is, the more you zoom in, the less you can pan (the effective space is pyramid shaped).

A better approach is to instead combine the two terms and compute transformations using:

```
zoom * (xoffset - xview)
```

Since the offset of the object counters the offset of the view for visible objects, the effects of overflow are reduced, and the result is a larger usable zooming space.

In implementation terms, computing transformations in this manner complicates matters. In the earlier equation, view and object transformations are carried out separately – an approach that lends itself to using a transform stack. Stacks are an elegant mechanism for graphical systems, which often have hierarchical structures (such as nested portals hierarchical groups) that are easily handled using recursion and stacks.

To implement this in Pad++, we keep a separate stack of view transformations and object transformations. This lets us combine the two terms separately as we compute coordinates.

An alternative solution to this problem is to use larger precision floating point numbers within the coordinate transformation system (e.g. store the coordinates using floats and perform transformations using doubles). One problem with this approach is that higher precision arithmetic comes at a performance cost – the result is increased space but slower rendering. If the coordinates are already stored in the highest precision format supported by hardware, this approach may not be feasible.

Conclusion

Pad++ is an implementation of a Zooming User Interface (ZUI). We achieved our goals of creating a substrate that supports rich zooming graphics with a large number of objects, and a consistent high frame rate for typical scenes.

ZUIs are strongly related to real time 3D graphical systems and 2D windowing systems. While we have reused existing real-time graphics techniques as much as possible, ZUIs present new and interesting challenges which distinguish them from their 2D and 3D cousins. Consequently, Pad++ represents a unique combination of features and implementation techniques.

There are still several areas where further research is required. Probably the most serious problem with Pad++ is its inability to maintain a high frame rate when many objects are visible simultaneously. Part of the difficulty here is that we have not solved the problem of visually presenting objects at many different levels of detail. While we do use level of detail to speed up rendering, object presentations do not change rapidly or extensively enough. Instead the system gets slow and animations are poor when too many objects are visible in a scene. In 3D systems, rendering lower-resolution models generally solves this problem. But this is hard to do in a reasonable fashion for text and images, which form the bulk of the visual content in a 2D system.

Finally, for Pad++ to scale up, we need to be able to handle scenes with a much larger number of objects. Currently, Pad++ holds all objects in RAM, and so the maximum number of objects is limited more by memory than anything else. Perhaps the best way to scale up the number of objects would be to link Pad++ to a persistent database, and keep only a small cache of objects in memory.

Acknowledgments

While Pad++ was primarily implemented by the authors of this paper, many other people have contributed to both the code and to our understanding of ZUIs. Our primary collaborator at the University of New Mexico is Professor Jim Hollan. Our primary collaborator at New York University is Professor Ken Perlin, the originator of Pad. Other students and staff at UNM that we have worked with include Hugh Bivens, Allison Druin, George Hartogensis, Ron Hightower, Mohamad Ijadi, David Proft, Laura Ring, David Rogers, Jason Stewart, David Thompson, David Vick, and Ying Zhao. People we have worked with at New York University include David Bacon, Troy Downing, Athomas Goldberg, and David Fox. Finally, people that have been involved with Pad++ in other places include Mark Rosenstein, Larry Stead, and Kent Wittenburg at Bellcore, and George Furnas at the University of Michigan. It is interesting to note that 6 out of 24 people that worked on this project are named David!

In regards to the Pad++ implementation, we particularly appreciate the help of David Fox who demonstrated how images could be scaled so quickly under X windows. We also appreciate Paul Haeberli from SGI who donated code for polygonizing Adobe Type 1 fonts.

This project has been primarily supported by generous funding from DARPA contract #N66001-94-C-6039 to which we are grateful.

References

1. Bederson, B. B., Hollan, J. D., Perlin, K., Meyer, J., Bacon, D., & Furnas, G.. “Pad++: A zoomable graphical sketchpad for exploring alternate interface physics,” *Journal of Visual Languages and Computing*, 7, 3–31, 1996.

2. Meyer, J., “EtchaPad – Disposable Sketch Based Interfaces,” *Proceedings Companion of Human Factors in Computing Systems (CHI’96)*, 195–196, ACM, New York, 1996.

3. Druin, A., Stewart, J., Proft, D., Bederson, B., and Hollan, J. D., “KidPad: A Design Collaboration Between Children, Technologists, and Educators,” *Proceedings of Human Factors in Computing Systems (CHI’97)*, 463–470, ACM, New York, 1997.

4. Sutherland, I. “Sketchpad: A Man-Machine Graphical Communication System,” *Tech Report #296*, MIT Lincoln Labs, Cambridge, MA, 1963.

5. Donelson, W. C. “Spatial management of information,” *Proceedings of Computer Graphics (SIGGRAPH ’78)* 203–209, 1978, ACM, New York, 1978.

6. Perlin, K., & Fox, D. “Pad: An alternative approach to the computer interface,” *Proceedings of Computer Graphics (SIGGRAPH ’93)* 57–64, ACM, New York, 1993.

7. Bederson, B. B., & Hollan, J. D., “Pad++: A zooming graphical interface for exploring alternate interface physics,” *Proceedings of User Interface Software and Technology (UIST ’94)* 17–26, ACM, New York, 1994.

8. Bederson, B.B, Hollan, J. D., Stewart, J., Rogers, D., Druin, A., Vick, D., Ring, L., Grose, E., Forsythe, C., “A Zooming Web Browser,” *Human Factors and Web Development*, Eds.: Forsythe, C., Ratner, J., and Grose, E., Lawrence Earlbaum, New Jersey , 1997.

9. Card, S. K., Robertson, G. G., Mackinlay, J. D., “The Information Visualizer, an Information Workspace,” *Proceedings of Human Factors in Computing Systems (CHI’91)*, 181–188, ACM, New York, 1991.

10. Perspecta, Inc., `http://www.perspecta.com`, 1997.

11. Merzcom, Inc., `http://www.merzcom.com`, 1997.

12. Fox, D., “Tab: The Tabula Rasa zooming user interface system”. URL http://www.cat.nyu.edu/fox/tab.html, New York University, New York, 1997.

13. Maloney, J. H. and Smith, R. B. “Directness and Liveness in the Morphic User Interface Construction Environment,” *Proceedings of User Interface Software and Technology (UIST ’95)* 21–28, ACM, New York, 1995.

14. Smith, R. B. “Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic,” *Proceedings of the Human Factors in Computing Systems and Graphics Interface Conference*, 61–67, ACM, New York, 1987.

15. Henderson, D. A. Jr. And Card, S. K. “Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface,” IEEE Transactions on Graphics, 5 (3), 211–243, 1987.

16. FVWM, `http://www.hpc.uh.edu/fvwm`, 1997.

17. Architecture Review Board (ARB), “OpenGL Reference Manual,” Addison-Wesley, Massachusetts, 1992.

18. Haeberli, P., Segal, M., “Texture Mapping as a Fundamental Drawing Primitive”, *Fourth Eurographics Workshop on Rendering*, (URL http://www.sgi.com/grafica/texmap/index.html), Cohen M., ed, Paris, France, June 1993.

19. Foley, J. D., van Dam, A., Feiner, S. K., Hughes, J. F., “Computer Graphics: Principles and Practice,” pp.992-996, Addison-Wesley, Massachusetts , 1990.

20. Deyo, R.J., Briggs, A., and Doenges, P. “Getting Graphics in Gear: Graphics and Dynamics in Driving Simulation,” *Proceedings of Computer Graphics (SIGGRAPH* '88), 24 (4), 317–326, ACM, New York, July 1988.

21. Airey, J.M., Rohlf, J.H., and Brooks, F. P. Jr. “Towards image realism with interactive update rates in complex virtual building environments,” *ACM SIGGRAPH Special Issue on 1990 Symposium on Interactive 3D Graphics*, 24 (2), 41–50, ACM, New York, 1990.

22. Angel, E., “Interactive Computer Graphics,” Addison-Wesley, 1997.

23. Funkhouser, T., Teller, S., Sequin, C., and Khorramabadi, D. “The UC Berkeley System for Interactive Visualization of Large Architectural Models,” *Presence: Teleoperators and Virtual Environments*, 5 (1), Winter 1996.

24. Samet, H. “The Design and Analysis of Spatial Data Structures,” Addison-Wesley, Massachusetts, 1990.

25. Samet, H. “Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS”, Addison-Wesley, Massachusetts, 1990.

26. Guttman, A. “R-trees: A dynamic index structure for spatial searching,” *Proceedings of the SIGMOD Conference*, 47–57, Boston, June 1984.

27. Kedem, G. “The quad-CIF tree: A data structure for hierarchical on-line algorithms,” *Proceedings of the Nineteenth Design Automation Conference*, 352–357, Las Vegas, June 1982.

28. Fuchs H., Kedem, Z. M., and Naylor, B. F. “On visible surface generation by a priori tree structures,” *Computer Graphics* 14 (3), 124–133, July 1980.

29. Bentley, J.L., “Multidimensional binary search trees used for associative searching,” *Communications of the ACM* (18), 509–517, ACM, New York, September 1975.

30. Beckmann, N., Kriegel, H. P., Schneider, R., and Seeger, B. “The R*-tree: An efficient and robust access method for points and rectangles,” *Proceedings of the SIGMOD Conference*, 322–331, Atlantic City, NJ, June 1990.

31. Perlin, K., “Layered Compositing of Facial Expression”, *Visual Proceedings of Computer Graphics (SIGGRAPH '97)*, 226-227, ACM, New York, 1993.

32. Furnas, G. W. and Bederson, B. B. “Space-Scale Diagrams: Understanding Multiscale Interfaces,” *Proceedings of Human Factors in Computing Systems (CHI'95)*, 234–241, ACM, New York, 1995.