Jazz: An Extensible 2D+Zooming Graphics Toolkit in Java

Benjamin B. Bederson, Britt McAlister
Human-Computer Interaction Lab, Institute for Advanced Computer Studies
Computer Science Department
University of Maryland, College Park, MD 20742 USA
+1 301 405-2764
bederson@cs.umd.edu, brittmc@cs.umd.edu

ABSTRACT

Jazz is a new general-purpose toolkit that supports applications using zooming object-oriented 2D graphics. It is built entirely in Java using Java2D, and thus runs on all platforms that support Java 2. It supports zooming, internal cameras, and lenses in a similar style to Pad++, but does so in a general purpose manner without a specific focus on zooming. Jazz is primarily a "scenegraph" for 2D graphics that is analogous to Sun's Java3D and SGI's OpenInventor in their support for 3D scenegraphs.

This paper describes Jazz and discusses the issues of using a scenegraph for 2D graphics. We discuss the Jazz architecture, and how applications can build on top of it.

Keywords

Zoomable User Interfaces (ZUIs), Animation, Graphics, User Interface Management Systems (UIMS), Pad++, Jazz.

INTRODUCTION

The world of toolkits for building graphical applications can broadly be divided in two: those toolkits designed for two-dimensional (2D) graphics, and those designed for three-dimensional (3D) graphics. Considering the number of 2D versus 3D graphical applications in use today, a disproportionate amount of energy has gone into building tools that support 3D graphics. This is largely due to the complexity of 3D graphics and its computational requirements. However, today's 2D graphics applications are becoming more complex as well.

One of the primary differences between 2D and 3D graphics toolkits is that 3D systems frequently consist of a renderer (the package that takes basic geometry and paints the pixels on the screen to represent that geometry) and a scenegraph (the package that helps an application to manage the data structures representing their geometry.) 2D graphic toolkits on the other hand are generally lacking in scenegraph support, and instead tend to focus exclusively on the renderer.

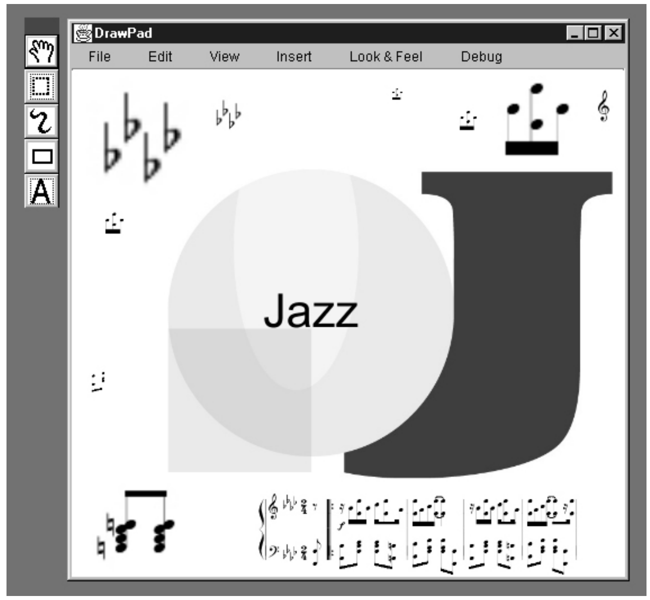

Figure 1: Screen snapshot of demo drawing program using Jazz

This focus on scenegraphs only for 3D systems can be seen in a majority of commercial graphics packages. For example, Silicon Graphics Inc. (SGI) has supported 3D scenegraphs through their OpenInventor software for many years, as do all of the 3D modeling packages that are used for animation and architecture, etc [4]. Even Java's Java3D package supports a scenegraph and renderer [1].

When looking at 2D packages, however, we usually find just renderers. The Mac toolkit, Microsoft Win32 API, X, and even Sun's new Java2D package have just 2D renderers without support for the data structures that all applications must maintain to use these renderers. Currently, there are a small number of 2D systems that have some kinds of scenegraphs, but not the general-purpose kind that we envision. The Tk Canvas supports object-oriented 2D graphics, but has no hierarchies or extensibility [15].

Jazz is open source software according to the GNU Library Public License, and is available at:

http://www.cs.umd.edu/hcil/jazz

InterViews supports a hierarchical structure for GUI widgets, but not general 2D graphics.

It could be argued that 2D graphics data structures are not that complex, and therefore do not need special support. However, we do not believe this, and think that to build a 2D graphical application is in fact quite complex. A principle reason is that computers have limited processing power. Writing graphical applications that perform well is difficult. It is usually necessary to be careful about what is rendered, and to only render the portion of the scene that is modified. This is especially true for animated interfaces which are becoming more common. Today, each application is left on its own to manage rendering. However, with Jazz, several efficiency mechanisms are supported.

The reasons for a 2D scenegraph include more than just speed. There are also several structural tasks that most graphical applications have to accomplish. For instance, it is common to group objects together, or even to create hierarchical groups of groups, and then move those groups around. Layers are becoming increasingly common with the ability to hide and show them on demand, or change the ordering of the layers. Nearly every application requires the ability to *pick* objects – that is, determine which graphical element is under the pointer. In addition, laying out graphical elements as well as saving them is also generally important.

We also believe that supporting multiple representations of data is important. That is, we want the same underlying model to appear differently on the screen depending on the context in which it is viewed. This context can include zoom (sometimes called *semantic zooming*) or the viewport that the model is viewed within (sometimes called *lenses* or *filters* [19].) Actually, the context could be anything else as well, from author, to the time the data is viewed.

More generally, we are interested in exploring interfaces that directly consider cognitive factors. While the venerable Information Visualizer [8] and Self [16] systems were motivated by an understanding of human cognition, that is not a principle motivating factor of most toolkits. Thus, we are building a toolkit that we feel will be generally useful for 2D graphics, but specifically useful for exploring novel interface ideas. Finally, our particular interest is in building Zoomable User Interfaces (ZUIs) that have the above mentioned complexities as well as others. We are interested in building fully zoomable information spaces.

So, why haven't scenegraphs [6, 7] for 2D graphics become more common? It is hard to be sure, but we speculate that there are several reasons. One is that the range of 2D applications is greater than for 3D applications, and a general-purpose solution will not be broad enough. 3D applications are more stereotypical, and can be well-modeled with a hierarchy of triangles. In addition, many 2D applications started out as text-based, and became more graphical. Scenegraphs are not as useful for text-based applications. A third reason is that 2D applications are often designed to run on low-end systems. 3D applications, however, typically run on higher-end systems, and can afford the luxury of abstraction. Finally, most 3D applications are spatially organized while many 2D applications are organized conceptually.

What is Jazz?

Jazz takes many of the structural elements common to 3D graphical systems, and creates a scenegraph for 2D graphics. By using a basic hierarchical scenegraph model with cameras, Jazz is able to directly support a variety of common as well as forward-looking interface mechanisms.

This includes hierarchical groups of objects with affine transforms (translation, scale, rotation and shear), layers, zooming, internal cameras (portals), lenses, semantic zooming, and multiple representations.

What makes Jazz unique is the small number of basic elements that work together to support this large feature set. While zooming has been one of our motivations for building Jazz, we think that its simple model will prove useful for builders of non-zooming applications as well.

We anticipate that Jazz will be useful for a broad array of applications. From drawing programs, and electronic presentations, to information visualization – we believe that Jazz's combination of extensibility, object orientation, hierarchical structure, and support for multiple representations will enable easier building of 2D graphical applications.

WHY ANOTHER ZUI TOOLKIT?

There have been several implementations of Zoomable User Interfaces (ZUIs) recently that address many of the above mentioned issues. This includes the original Pad system [17], and more recently Pad++ [5, 6, 7]. There also have been some other systems developed by individuals for research purposes [10] as well as some commercial ones that have not been widely accessible [2, 3].

Pad++ is the ZUI that we have the most experience with, and is the ZUI that has been most widely used. Thus, one question is why we are starting fresh, rather than extending Pad++.

The biggest problem with Pad++ is that it is closely tied to X and Tcl/Tk [15]. Much of the Pad++ core had design decisions made due to the shortcomings of the application language (Tcl/Tk). Several features were implemented in the core that should have been left to the application because Tcl/Tk was too slow. In addition, since Pad++ was designed originally to be a prototyping tool, the Pad++ core grew to become quite large, and supported many general-purpose features. This tended to make it easy to do basic things, but harder to do more complex things.

The second major issue is that the Pad++ class structure has several significant design problems, and is difficult to extend. For example, an application can't have multiple top-level views onto a single surface. Also, the Pad++

coordinate systems, hierarchical groups, and anchor points while feature rich, are very confusing and hard to use programmatically. And, in Pad++ the renderer is tightly coupled to the object system; both of which are bound to events. This makes Pad++ difficult to port to either different operating systems, or different graphics systems in a clean manner.

Finally, we believe that Java will play a large part in the future of application programming. Connecting Pad++ to Java, while technically feasible, would be difficult because it would require a wrapper around every function. This would be difficult to maintain and we would likely have similar language conflicts as existed in C++ and Tcl/Tk. In addition, there is a fundamental inefficiency in that Java application programmers would typically want to create a Java object that would have to be associated with each C++ object. This would lead to complexities relating to dangling references to C++ objects from Java when a C++ object is deleted.

Impact of a 2D Scenegraph

A standard scenegraph architecture for 2D graphics offers a straightforward solution to most of the issues just raised. However there are costs as well. Let us look at some of the tradeoffs that come about with the use of a scenegraph.

Advantages:

- **Complexity:** Scenegraphs scale nicely, and handle complex scenes well.
- **Abstraction:** Scenegraphs decouple the components of the system, making it easier to improve the renderer, switch to different hardware, make platform-specific tweaks transparently, etc.
- **Reusability:** Scenegraphs allow novice programmers to use professionally implemented algorithms, and to avoid implementing many common features.
- **Interactivity:** Scenegraphs make it easier to implement things like selection and picking.
- **Aliasing:** Scenegraphs make it easy to reuse data in multiple places.

Disadvantages:

- **Footprint:** A general solution such as a scenegraph will likely use more memory than a custom solution.
- **Efficiency:** It is typically more efficient to write a custom solution than to use a general-purpose scenegraph.
- **Restrictions:** Even with the most open-ended designs, a scenegraph is likely to place some restrictions on the application, which may be avoidable with a custom solution.

DESIGN GOALS

The design of Jazz, like any system, includes many tradeoffs. Our primary high-level design goals, which influence the tradeoffs are:

- **Ease of adoption:** We must offer a clean, and understandable substrate for writing 2D object-oriented graphical applications, so that the barrier to adopting the technology is low. To this end, our motto with Jazz is "First build it right, then build it fast". We have strived to design Jazz so that others can use it quickly, and integrate legacy Java code.
- **Extensibility:** In order that Jazz is truly useful, it must be readily extensible by application builders to support their specific needs. We can not anticipate all possible uses of Jazz, and so our strategy has been to design it to be extensible in as general a way as possible. To this end, Jazz has no specific visual or interaction policy. Jazz comes with a default set of visual objects, but there is a well-defined mechanism for applications to define their own. Similarly, Jazz supports default selection, navigation, and other interaction mechanisms, but they are designed to be modifiable by applications.
- **Performance:** The system must run well on a wide variety of platforms, and also offer rich features such as animation and complex graphics. To this goal, Jazz's general policy is to favor speed over memory.
- **Scalability:** Jazz must support serious and complex applications. Towards this end, we were willing to give up the simplicity that Pad++ offered in order to support the long-term goal of non-trivial applications. However, Jazz has several utilities to help lower the start-up costs of building applications.

The name Jazz is not an acronym, but rather is motivated by the new music-related naming conventions that the Java Swing toolkit started. In addition, the letter 'J' signifies the Java connection, and the letter 'Z' signifies the zooming connection.

JAZZ ARCHITECTURE

Figure 2 shows the object hierarchy of Jazz's public objects that applications use. Figure 3 shows the object structure of a typical application with several objects and a camera. The basic elements of Jazz are summarized below.

- **Scenegraph**: Jazz supports an object organization based on a scenegraph paradigm including the ability to share graphical objects so that objects can appear multiple times in the scene. The scenegraph consists of a hierarchy of nodes that represent relationships between objects. Each node has a visual component that defines the visual object associated with that node. Alternatively, nodes can have a chain of visual components using the *Decorator* pattern [12].
- **Optimized Renderer:** Jazz uses the Java2D renderer, and is organized to efficiently support fast animation, high quality stills, very large images, and efficient screen updates.

- **Visual Components:** Jazz defines a base visual component type and several basic components, which are subclassed from that base type. Applications can define new object types by sub-classing the base type or the default objects.
- **Sticky Objects**: Objects that are "sticky" are associated with a specific camera, and do not move as the camera viewpoint changes. A typical sticky object would be a status bar that always stays at the bottom of the screen as the camera pans and zooms over the rest of the scene.
- **Cameras:** Scenegraphs are seen through cameras. A camera contains an affine transformation, which controls what part of the scenegraph is seen within that camera. A scenegraph can be seen through multiple cameras, each of which can navigate through the scenegraph separately. Cameras may be mapped to a top-level window (or Java widget), or to a printer. In addition, cameras may be nested. When one camera sees another camera, the internal camera acts like another view into the scenegraph. (We have used the word *portal* in the past to refer to these internal cameras [19].)
- **Drawing order:** Objects must have a specific drawing order. That is, each object is always either in front of or behind another.
- **Coordinate systems:** The coordinate system has the origin (0, 0) at the upper left, and X increases to the right and Y increases down. The camera and every visual component have an arbitrary 2D affine transformation that can be used to translate, scale, rotate, and shear the objects. When a camera's transformation is the identity matrix, the origin is at the top-left corner of the screen. Objects are defined relative to the coordinate system of their parent node, but can be translated, scaled, rotated, or sheared relative to that coordinate system.
- **Object Storage:** Jazz defines a new object storage mechanism similar to Java Serialization, but it is more stable over different versions of code. It is text-based and so can be hand-edited, and is well-defined so other applications (even ones written in other languages) can define objects that can then be read into Jazz. This object storage mechanism can be used without using the rest of Jazz.
- **Lenses:** A lens provides a mechanism for changing the way objects look through a specific camera. A lens can encapsulate any kind of semantic transformation – so while a simple lens may change the contrast of an image, a more complex lens may show some structure of an image.
- **Events:** Events are associated with individual objects. The same listener interface that is used in Java 1.1 are used per object within the scenegraph. (Currently, only

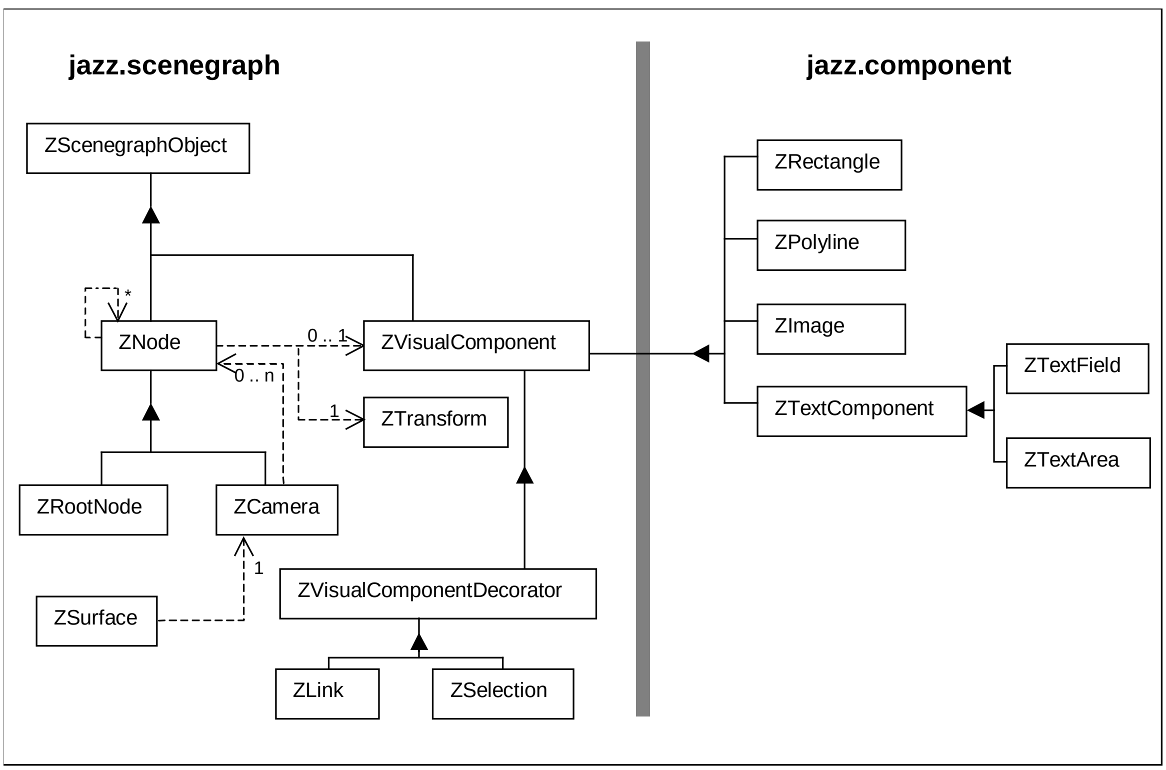

Figure 2: Static object hierarchy of Jazz showing the scenegraph nodes and the basic components, specified in slightly modified UML. Solid lines/arrows arrows represent inheritance. Dashed lines with open arrows represent an association. The number next to the open arrow specifies how many instances of the object is associated with the object pointing to it.

simple events are implemented.)

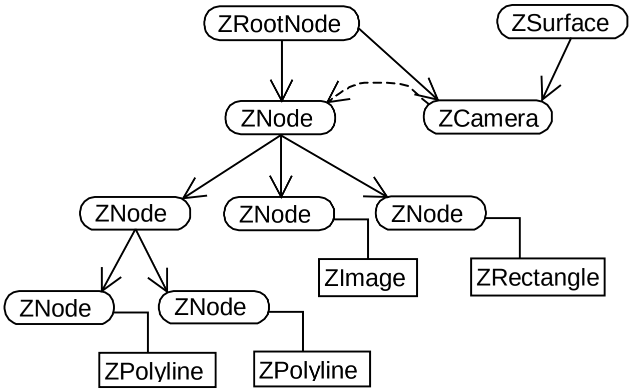

Figure 3: Run-time object structure in a typical application. This scene contains a single camera looking onto a layer that contains an image, a rectangle, and a group consisting of two polylines.

- **Layout:** Objects can be positioned and scaled with one-time layout procedures, or with active layout managers in a manner analogous to the Java AWT layout managers. (Currently, only simple one-time layout methods are implemented.)

One of the goals of Jazz is to support a rich set of functionality using a simple, understandable, and extensible design. Jazz supports all of the features that Pad++ does with just three primary concepts: cameras, nodes, and visual components.

Jazz has the following packages:

jazz.scenegraph	Primary package
jazz.component	Basic visual components
jazz.event	Event handler support
jazz.util	Utility classes
jazz.io	Generic I/O system

Nodes and Visual Components

The scenegraph package defines the data structure used to store all the objects. Each object (such as a rectangle) has two elements in a scenegraph, a node and a visual component. This separation exists to support hierarchical structuring of objects. Hierarchies can be useful in a scenegraph because they support hierarchical movement of objects. For instance, hierarchies can be used to implement "groups" and "layers" that are found in most drawing programs. An application using the Jazz scenegraph uses nodes to construct the hierarchy. Then, each node has a visual component, which specifies what gets drawn for that node.

There is a clear separation between what goes in a node, and what goes in a visual component. Nodes contain all object characteristics that are passed on to child nodes. This includes an affine transform, min/max magnification, and transparency (affine transforms support translation, rotation, scale, and shear). Each of these characteristics (e.g., rotating) affect the node's visual component and also affect that node's children.

The visual components are just visual. They have no structure, and do not even have a transformation that specifies where it should be rendered. The visual component specifies just how to render something, how big it is, and how to select it. Each node may be assigned a specific component, or else it will reference a "dummy" component that does nothing. The dummy component is implemented as a *Singleton* [12] and gives the guarantee that every node has a component, and thus applications are not forced to check if the component reference is null before accessing it.

Visual components are rendered in the order of their associated nodes. That is, if a node has five child nodes, then those five children's visual components will be rendered one after the other in the order of the child nodes. Changing the order of a node within a parent node will change the rendering order of the associated component.

Nodes can be "hidden", which means that neither it nor any of its children will be rendered or receive events. Visual components may be set to be "unpickable" which means they do not receive events.

Visual components typically define their bounds to help Jazz decide which objects are visible, and thus quickly cull objects that are not visible in a given view. Because nodes are hierarchical, the bounds are cached at each node in the current relative coordinate system. Objects that regularly change their dimensions, however, can define themselves as *volatile*. This tells Jazz not to cache their bounds, and instead to query the object directly every time the bounds are needed to make a visibility decision.

While a node must check all of its children for visibility, we plan on adding optional spatial indexing per node. This should help reduce searching time when a node has many children.

Visual component decorators can be used to associate extra characteristics with a visual component without forcing all visual components to define that behavior. For instance, Jazz defines a selection decorator, which draws a highlight around its component to indicate that the component is selected. Similarly, Jazz defines a link decorator, which is used to create spatial hyperlinks. The link decorator associates the destination of a spatial hyperlink with a visual component, but does so without modifying the component, and in fact the component doesn't even know about it. We also envision decorators that allow a single component to appear in multiple places by defining a decorator with multiple node parents. Jazz uses Java introspection to support the finding of specific decorator or component types by searching the chain. Figure 4 shows the data structure representing a more complex scene with a decorator chain.

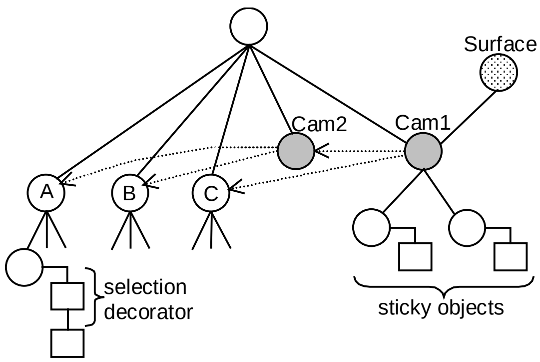

Figure 4: A more complex scenegraph. The top-level camera (cam1) sees the internal camera (cam2) and node C, and has two sticky objects. Cam2 sees nodes A and B. Node A has a child with a visual component decorator chain that selects the component.

Why the split?

This split between nodes and visual components may at first seem odd, but this separation solves a number of problems that plagued Pad++ as well as early versions of Jazz. For one, it separates the *structure* from the *content*. Visual components are interchangeable, making it possible to, say, replace all the circles w/ squares in a sub-tree of the scenegraph without affecting the grouping or position of objects.

It also greatly simplifies coordinate system issues. Because objects can be transformed hierarchically, and can even appear in different positions on the screen due to cameras (and aliasing decorator chains), it can be very confusing to understand what coordinate system an object or event is defined in. With our split approach, visual components are always defined in their own local coordinate system. The nodes define where the components end up by using affine transforms, which essentially define coordinate system mappings.

Finally, the split between nodes and components offers a more de-coupled design and simplifies modularity. Component types are defined independently from each other and from the rest of the scenegraph. As described in a later section, this enables applications to define visual components quite easily, or even extend nodes. Using a component decorator chain, each component can remain small while enabling more complex features (e.g., selection and hyperlinks) to use resources where needed.

Cameras

A camera shows a portion of the Jazz scenegraph. Cameras specify which portion of the scenegraph should be rendered within that camera using an affine transform. Multiple cameras are supported, and cameras can navigate the space independently of one another. Cameras can either be attached to top-level *surfaces* (see next section), or can be viewed by other cameras. In the latter case, they are called *internal cameras*, and act like windows within the world that look onto the world. In previous ZUI implementations, we called these "portals". Since cameras are nodes (they extend ZNode), they also can be transformed as objects via the node transform. This enables internal cameras to be moved within a scene.

Each camera controls which portion of the scenegraph it sees by specifying its *paint start points*. A camera renders itself by first rendering its background, then all the nodes in its paint start point list, and then finally the camera's visual component (if there is one). This approach lets an application build a single very large scenegraph, but control which portion of the scenegraph is visible and where.

Jazz does not specifically support traditional layers, but traditional layers can be implemented directly using the hierarchical scenegraph in combination with cameras. Traditionally, a drawing system contains objects which each sit on a single layer. The drawing system can specify which layers are visible, and can change the drawing order of entire layers. This is accomplished in Jazz by creating one hierarchical node in the scenegraph for each layer, and putting objects that should appear on a specific layer under the node that represents that layer. The layer may be turned off within all cameras by hiding the relevant node, or the layer can be turned off within just one camera by removing the appropriate node from the camera's paint start point list. Finally, the layer rendering order can be changed by changing the order of the layer nodes. Figure 4 shows how multiple cameras are set up.

Sticky Objects

Sticky objects are visual components associated with a camera that do not move as the camera viewpoint is changed. Jazz implements sticky objects by putting them under a camera node. The camera first renders the portion of the scenegraph it refers to through its paint start points *with* the viewpoint transform, and then renders its children (the sticky objects) *without* applying the view transform. The camera's children then do not move as the viewpoint changes. It is as if they are stuck to the camera's window.

This implementation of sticky objects solves a problem that we had with Pad++. Sticky objects in Pad++ were implemented in a general way by applying a constraint to those objects. Whenever the camera viewpoint was changed, the inverse transform of the view change was applied to the sticky objects, thus leaving them at the same place on the screen. This strategy worked, but because every view change resulted in two transforms being applied to the sticky objects, they often "jiggled" on the screen - especially when zoomed in. Jazz avoids this problem by simply rendering the sticky objects without applying the camera view transform to sticky objects. Figure 4 shows a scenegraph with some example sticky objects.

Surfaces

Cameras are mapped to operating system windows through *surfaces*. The abstraction of a surface is important for two reasons. First of all, it encapsulates a Java `Graphics2D` class, which supports 2D rendering. A `Graphics2D` can come from either a window, a back buffer, or a printer.

Thus, with this mechanism, a Jazz surface can be used to display, print, or to render into a back buffer so an application can grab the pixels that were rendered.

The second purpose of a surface is to store the region management information. Jazz stores region management in the surface, and does so using the scenegraph global coordinate system rather than window coordinates for efficiency.

CREATING NEW VISUAL COMPONENTS

Jazz defines a basic set of simple visual components for basic geometries and text. However, we expect that many application designers will create their own to specify a new visual look, or complex custom components.

In order to define new visual components, an application must extend the `ZVisualComponent` class, and must define three functions. The new object must define how to paint itself, how big it is, and how to pick itself. Picking means that the object must specify whether a given rectangle is considered to intersect the object or not. Figure 5 shows the code necessary to define a simple circle visual component.

This standard picking is used just to select whether the pointer is over an object. Visual components are free to define their own more advanced picking methods, such as for selecting the text within a component, but that is separate, and not supported by the visual component interface.

Using this mechanism, visual components can be easily defined that wrap existing Java code which was written without consideration of Jazz. For example, it would be possible to take some pre-existing code that draws a scatterplot, and make it available as a Jazz visual component. To do this, a class similar to the circle defined in Figure 5 would use the bounds of the scatterplot, call the scatterplot's paint method, and then implement a simple pick method that returned true with an intersection anywhere within the bounds of the scatterplot.

MODEL BASED COMPONENTS

An application that renders a scatter plot might be designed in various ways. A straightforward approach would be to create a new component that extends `ZVisualComponent.` The class could store the data, and its `paint` method could render the plot. This is the simplest way, but has the problem that the data is coupled to the representation of the data. Thus, changing the data or the representation independent of each other would require knowing about the other – since it is all stored in one class.

An alternative approach would be to have the class reference an underlying data model. Whenever the data model changes, the scatter plot class would be notified, and would re-render itself based on the new data.

Because we envision many visual components relying on an underlying data model, we plan to define a very simple

```
         //
         // A basic circle visual component
         //
public class ZCircle extends ZVisualComponent {
    protected Ellipse2D.Float circle;

                          // Creates circle with center (x,y) and radius = r
    public ZCircle(float x, float y, float r) {
         circle = new Ellipse2D.Float(x - r/2, y - r/2, r, r);
         updateBounds();   // This results in computeLocalBounds getting called
    }
                          // Describe how to paint circle
    public void paint(ZRenderContext renderContext) {
         Graphics2D g2 = renderContext.getGraphics2D();
         g2.setColor(Color.black);  // Draw circle
         g2.fill(circle);
    }

                                   // Describe how big the circle is
    protected void computeLocalBounds() {
        localBounds.setRect((float)circle.getX(), (float)circle.getY(),
                            (float)circle.getWidth(), (float)circle.getHeight());
    }

                                   // Describe how to pick the circle
    public boolean pick(Rectangle2D r) {
        boolean picked;
        picked = circle.intersects(r.getX(), r.getY(), r.getWidth(), r.getHeight());
        return picked;
    }
}
```

Figure 5: Java code to create a new circle visual component

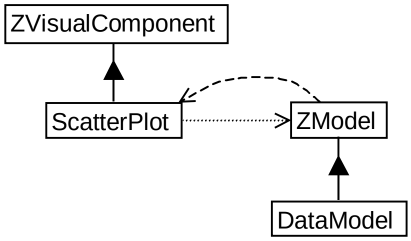

Figure 6: Basic model-based visual component

`ZModel` class whose primary purpose will be to generate events when a model has changed to notify the visual component of the change. Figure 6 shows the basic class structure of an application using this approach. (`ZModel` is not implemented yet.)

MULTIPLE REPRESENTATIONS OF OBJECTS

The basic approach for representing models is satisfactory, except that it does not show how we can have *multiple* representations of a single data model. That is, it doesn't support visually displaying an object differently in different contexts.

One of the basic things that Jazz supports is the ability to present different visual representations of data in different circumstances. Because many different kinds of context can be used to influence how and what gets drawn, we sometimes call this *context-sensitive rendering*. While an application can use any context whatsoever to control an object's rendering (such as author, or time), two especially common contexts are magnification and the camera being rendered within. We sometimes use the more specific term *semantic zooming* to refer to objects that change the way they appear based on the current magnification. When an object appears differently when viewed with different cameras, we sometimes use the term *lens* or *filter* [6, 19].

Jazz's standard visual components generally render themselves the same way every time. Although for efficiency, the standard visual components sometimes render themselves at lower resolution during interaction. However, an application can define a new visual component whose paint method depends on some context.

Standard software engineering approaches lead us to desire decoupled representations. That is, each visual representation should exist independently of the others. This allows the application builder to design new representations, and modify old ones without affecting the other representations. A clean decoupled design would support different classes for each visual representation of the data. One design that accomplishes this is to make a special visual component that acts as a *Proxy* [12] for another visual component, and can delegate between them.

Such a delegator is fairly straightforward to build. It maintains a list of ancillary visual components and exactly one of them is active at a time. It then defines its `paint`, `pick`, and `computeLocalBounds` methods to call the active visual component.

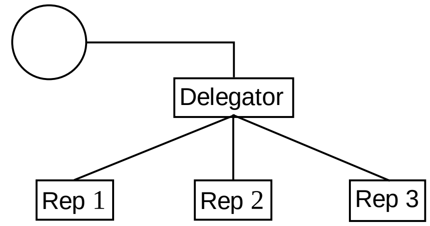

Figure 7: One way to implement semantic zooming in Jazz. The *Delegator* decides which representation to use to paint itself depending on the current magnification.

We implemented a simple delegator as a proof-of-concept. Our sample delegator supports semantic zooming by selecting a specific visual component to render based on the current magnification level. This approach has the property of having decoupled visual representations while keeping those representations together on the screen. Because they are all controlled by a single node, moving that node (by changing its transformation) moves each zoom level's representation together. We expect that this approach will work effectively for semantic zooming, lenses, and other forms of multiple representations. Figure 7 shows the basic structure of the scenegraph for our delegator that supports semantic zooming.

Jazz.IO

Jazz includes Jazz.IO, a sub-package that defines a new general-purpose mechanism and file format to save and restore Java Objects. To understand the need for Jazz.IO, consider the standard mechanism, Java Serialization.

Since JDK 1.1 Java has provided the `Serialization` interface for capturing and saving the state of object instances. Serialization is a useful extension to the Java stream classes. However, it comes with a mixed bag of features and liabilities. On the positive side, Serialization is a relatively fast encoding/decoding, compact binary format. Also, multiple reference and circular dependency issues are handled automatically.

Unfortunately, Serialization also presents many undesirable characteristics as well. Not all core Java classes implement the Serialization interface. One solution is to subclass the required core class and assign the subclass responsibility for saving the superclass's state. Unfortunately, several core classes, such as `TextLayout`, are declared final and cannot be extended. One solution to serialize these classes is to create a *Proxy* [12] class to wrap, encode and decode the final class.

However, this does not address all cases as some core classes do not implement the Serialization interface, and they contain private state information, but don't have public accessor functions (e.g., `LineBreakMeasurer`.)

Another difficulty with Serialization is the lack of version safety. Without the use of the `SerialVersionUID`, even very small changes to a class will render saved classes unreadable by the new class. Even with the

`SerialVersionUID`, any of the following changes to a class can result in version conflict:

- Deleting fields
- Moving classes up or down the hierarchy
- Modifying `writeObject()` or `readObject()`
- Changing the type of a primitive field
- Changing a non-static field to static
- Changing a non-transient field to transient

Primarily, it was the issue of version safety that led us to develop Jazz.IO. It was important to ensure Jazz users that their saved scenes would remain usable as new versions of Jazz were released. The options were to develop a version safe persistence mechanism or to provide a utility with each new version of Jazz that converted existing scene files to the new object format.

We chose the first option and developed Jazz.IO. Jazz.IO provides a well-defined text-based file format. While it is slower and files can be larger than with Serialization, files are more version resistant. And, because the file format is well-defined, it is straightforward to generate these files manually, or even from other languages. We have, in fact, already done this to populate a Jazz scenegraph with data generated from a Lisp-based language translation system.

Jazz.IO Architecture

Jazz.IO follows our design guideline regarding initial ease-of-use versus support for complex features. Jazz.IO requires slightly more work than Java Serialization for the simple case, however, Jazz.IO scales well and supports complex objects such as images.

Using Jazz.IO requires two classes and one interface. The first class, `ZObjectOutputStream` is almost identical in function to Java's `ObjectOutputStream`, and is used in the same manner. Jazz.IO handles the multiple reference and circular dependency issues automatically. The `ZParser` class is used to decode and instantiate the objects in a scene file. It returns a reference to a copy of the object originally written out.

All objects that are to be saved in the scene file must implement the `ZSerializable` interface (with the exception of supported core Java classes.) The interface defines four methods. The first is the `isSavable()` method which specifies if a particular object should be written out at all. The next method is the `ZWriteObjectRecurse()` method. This is used to determine which referenced objects should also be saved in the scene file. Then, `writeObject()` specifies how instance variables should be saved. The last method is used to read back in the objects. The parser instantiates each object in the scene file using the default, no argument constructor, and then calls the `setState()` method for each property to return the object to the pre-saved state.

CURRENT STATUS

Everything described in this paper is currently implemented in Java 2 and publicly available, except for complex lenses, object-based events, support for models, and the layout manager. In addition, we have only just begun the implementation of multiple representations of objects.

The Jazz distribution comes with a sample application called DrawPad that demonstrates some of the basic features. In addition, we are currently building two other applications. The first is a new version of KidPad [9, 14, 18] that provides collaborative storytelling tools for children. The second is an authoring tool for creating presentations. The authoring tool will build on our experience with PadDraw [6], while expanding the notion of editing in scale space [11] by offering simplified constrained tools [13].

We are still in the "do it right" state of implementing Jazz, and have not yet done serious performance tuning or analysis. Based on our informal observation, it appears to run 2 or 3 times slower than Pad++, but we are optimistic. Jazz currently can do animated full-screen zooms with moderate datasets running on a Windows NT 400MHz Pentium II machine. However, the system does slow down when we create numerous or complex components.

We plan on addressing performance issues with three approaches. To begin with, Sun and other companies are working on faster JVMs and faster implementations of Java2D. This is actually more hopeful than it may first appear because it is likely that some vendor will create a version of Java2D implemented using a graphics library that can take advantage of 3D graphics acceleration hardware that is now becoming common in PCs and workstations alike.

Second, we have not yet optimized our code. For example, we need to be careful to avoid all memory allocation within paint methods. We also will add spatial indexing, load management, and several other standard scenegraph optimizations.

Finally, our experience with Java so far is that while it provides excellent powerful classes that are very general, this generality leads to inefficiency. Because Jazz uses only a subset of Java2D's rendering features, we are looking into implementing a portion of Java2D natively for a specific platform (Windows). Our expectation is that because we can get by with supporting a subset of the full Java2D functionality, we can implement it faster than the standard implementation. However, we have just started this investigation.

CONCLUSION

This paper describes the architecture of Jazz, a new Java toolkit that supports the development of extensible 2D object-oriented graphics with zooming and multiple representation. It is a descendent from previous Zoomable User Interfaces that one of the authors has built in the past. Jazz is unique in that it is designed in a general way to support several advanced interface features in a straightforward way with a small number of essential classes.

While Jazz is still young, we are optimistic about its utility. A basic design goal has been to provide an extensible architecture and well-engineered code so that this will be a platform on which to build a variety of 2D applications.

ACKNOWLEDGMENTS

We appreciate our previous collaborations with those involved with Pad++, especially Jim Hollan, Jason Stewart, Jon Meyer, Allison Druin, and George Furnas. Jason Stewart was also involved in early design discussions about Jazz's support for multiple representations of models.

We would also like to thank our fellow members of the HCIL, and especially the students in the seminar on ZUIs that had the patience to use early versions of Jazz and helped to identify its "features". Maria Jump also contributed to Jazz's development in recent months and we are grateful to her. We also greatly appreciate the careful reading of this paper by Jon Meyer, Bay-Wei Chang, Jason Stewart, and Allison Druin.

Finally, Bob Hummel at DARPA has been instrumental in supporting this work, and Jazz wouldn't exist without his support. This work has been funded in part by DARPA, and an equipment grant from Sun Microsystems.

REFERENCES

1. (Java [Web Page]. URL http://www.javasoft.com.
2. (MerzCom [Web Page]. URL http://www.merzcom.com/.
3. (Perspecta [Web Page]. URL http://www.perspecta.com/.
4. (SGI OpenInventor [Web Page]. URL http://www.sgi.com/Technology/Inventor/.
5. Bederson, B. B., & Hollan, J. D. (1994). Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. *In Proceedings of User Interface and Software Technology (UIST 94)* New York: ACM, pp. 17-26.
6. Bederson, B. B., Hollan, J. D., Perlin, K., Meyer, J., Bacon, D., & Furnas, G. W. (1996). Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics. *Journal of Visual Languages and Computing, 7,* 3-31.
7. Bederson, B. B., & Meyer, J. (1998). Implementing a Zooming User Interface: Experience Building Pad++. *Software: Practice and Experience, 28*(10), 1101-1135.
8. Card, S. K., Robertson, G. G., & Mackinlay, J. D. (1991). The Information Visualizer, an Information Workspace. *In Proceedings of Human Factors in Computing Systems (CHI 91)* ACM Press, pp. 181-188.
9. Druin, A., Stewart, J., Proft, D., Bederson, B. B., & Hollan, J. D. (1997). KidPad: A Design Collaboration Between Children, Technologists, and Educators. *In Proceedings of Human Factors in Computing Systems (CHI 97)* ACM Press, pp. 463-470.
10. Fox, D. (1998). *Tabula Rasa: A Multi-scale User Interface System*. Doctoral dissertation, New York University, New York, NY.
11. Furnas, G. W., & Bederson, B. B. (1995). Space-Scale Diagrams: Understanding Multiscale Interfaces. *In Proceedings of Human Factors in Computing Systems (CHI 95)* ACM Press, pp. 234-241.
12. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
13. Hochheiser, H. S., & Bederson, B. B. (1999). Authoring With Discrete Levels in Zoomable User Interfaces. *In Proceedings of User Interface and Software Technology (UIST 99)* ACM Press, (submitted).
14. Hourcade, J. P., Iyer, V., & Bederson, B. B. (1999). Architecture and Implementation of a Java Package for Multiple Input Devices (MID). *In Proceedings of User Interface and Software Technology (UIST 99)* ACM Press, (submitted).
15. John K. Ousterhout. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.
16. Maloney, J. H., & Smith, R. B. (1995). Directness and Liveness in the Morphic User Interface Construction Environment. *In Proceedings of User Interface and Software Technology (UIST 95)* ACM Press, pp. 21-28.
17. Perlin, K., & Fox, D. (1993). Pad: An Alternative Approach to the Computer Interface. *In Proceedings of Computer Graphics (SIGGRAPH 93)* New York, NY: ACM Press, pp. 57-64.
18. Stewart, J., Bederson, B. B., & Druin, A. (1999). Single Display Groupware: A Model for Co-Present Collaboration. *In Proceedings of Human Factors in Computing Systems (CHI 99)* ACM Press, (in press).
19. Stone, M. C., Fishkin, K., & Bier, E. A. (1994). The Movable Filter As a User Interface Tool. *In Proceedings of Human Factors in Computing Systems (CHI 94)* ACM Press, pp. 306-312.