Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java

Benjamin B. Bederson, Jon Meyer, Lance Good
Human-Computer Interaction Lab
Institute for Advanced Computer Studies, Computer Science Department
University of Maryland, College Park, MD 20742
+1 301 405-2764
{bederson, meyer, goodness}@cs.umd.edu

ABSTRACT
In this paper we investigate the use of scene graphs as a general approach for implementing two-dimensional (2D) graphical applications, and in particular Zoomable User Interfaces (ZUIs). Scene graphs are typically found in three-dimensional (3D) graphics packages such as Sun's Java3D and SGI's OpenInventor. They have not been widely adopted by 2D graphical user interface toolkits.

To explore the effectiveness of scene graph techniques, we have developed Jazz, a general-purpose 2D scene graph toolkit. Jazz is implemented in Java using Java2D, and runs on all platforms that support Java 2. This paper describes Jazz and the lessons we learned using Jazz for ZUIs. It also discusses how 2D scene graphs can be applied to other application areas.

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

INTRODUCTION
Today's Graphical User Interface (GUI) toolkits contain a wide range of built-in user interface objects (also known as widgets, controls or components). These GUI toolkits are excellent for building hierarchical organizations of standard widgets such as buttons, scrollbars, and text areas. However, they fall short when the developer needs to create application-specific widgets. Developers typically write these application-specific widgets by subclassing an existing widget and overriding methods to define new functionality. However, GUIs have become more sophisticated, and the level of functionality needed to implement a new GUI widget has increased. Beyond writing the code to draw the widget, the developer must also write code to handle events, drag and drop, selection, layout, keyboard navigation, keyboard focus highlighting, tool tips, context sensitive help, popup menus, accessibility, internationalization, animated scrolling, and so on. Implementing a fully functional application-specific widget is a daunting task.

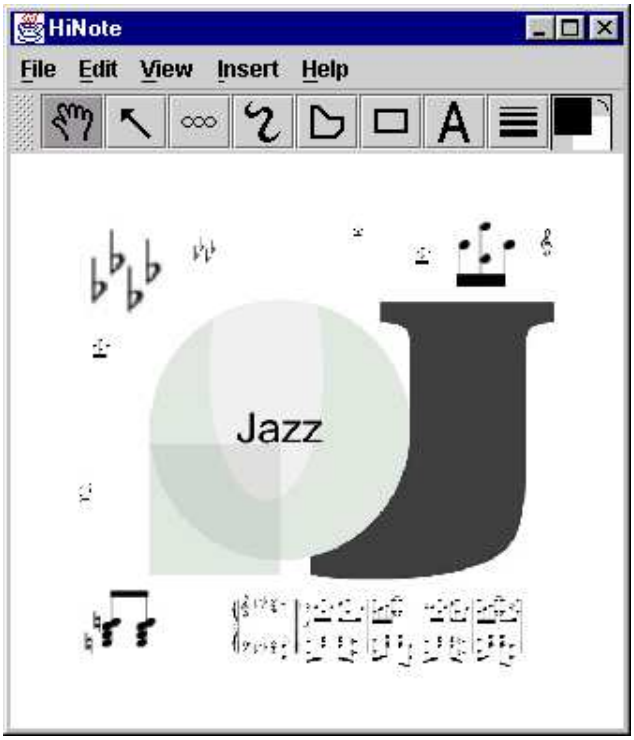


Figure 1: Screen snapshot of the HiNote application program, written using Jazz.

We believe that a significant problem with existing 2D user interface toolkits is that they follow a "monolithic" design philosophy. That is, they use a relatively small number of classes to provide a large amount of functionality. As a result, the classes tend to be complex and have large numbers of methods, and the functionality provided by each class is hard to reuse in new widgets.

To address code reuse, toolkit developers usually place generally useful code in a top-level class that is inherited by all of the widgets in the toolkit. There are several drawbacks to this approach. Firstly, it leads to a very complex hard-to-learn top-level class. (In Microsoft MFC, the top-level CWnd class has over 300 methods. The Java Component class has over 160 methods. Even Java Swing, a relatively new toolkit with a modern design, has a top level JComponent class with over 280 methods). Secondly, application developers are forced to accept the functionality

provided by the toolkit's top-level class – they cannot add their own reusable mechanisms to enhance the toolkit. Many application developers create their own custom toolkit so that they can have complete control over the capabilities of widgets in their application.

For several years, we have been investigating Zoomable User Interfaces (ZUIs), which use zooming as a principal method of navigation. ZUIs have a number of unique requirements, such as the need for "semantic zooming" where more detail is displayed as the scene is zoomed in, and the need for multiple views of the same scene at different magnifications. In practice, implementing a general purpose widget for supporting ZUI applications is very hard [7].

In this paper, we report on our experiences developing Jazz[1]. Jazz is a new toolkit for developing ZUI applications. It can also be used to build many other kinds of 2D widgets.

Unlike prior GUI toolkits, Jazz is based on a "minilithic" design philosophy. In Jazz, functionality is delivered not through class inheritance but rather by composing a number of simple objects within a scene graph hierarchy. These objects are frequently non-visual (e.g. layout nodes), or serve to "decorate" nodes beneath them in the scene graph with additional appearance or functionality (e.g. selection nodes). Jazz therefore tackles the complexity of a graphical application by dividing object functionality into small, easily understood and reused node types. Those nodes can be combined to create powerful applications. The base ZNode class in Jazz has under 60 public methods (16 are related to events, 16 are related to the scenegraph structure, 8 are related to local and global coordinates, and the rest are for other functions such as painting, saving, properties, and debugging.)

We believe that minilithic scene graphs are an important mechanism for supporting custom 2D application widgets in general, and ZUIs in particular. While zooming has been one of our motivations for building Jazz, we think that its simple model will prove useful for non-zooming applications as well. In particular, we believe that Jazz's combination of extensibility, object orientation, hierarchical structure, and support for multiple representations will simplify the task of writing many application-specific 2D widgets.

In this paper, we first describe the unique requirements of ZUIs that led us to create Jazz. We outline related work and discuss the architecture of Jazz. We show how Jazz supports adding functionality by composition, and describe some applications we have built using Jazz. We conclude by describing some of our experiences building the Jazz toolkit, and outline future work.

[1] 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 is open source software according to the Mozzilla Public License, and is available at:
http://www.cs.umd.edu/hcil/jazz

REQUIREMENTS FOR ZUIS

Zoomable User Interfaces are a kind of information visualization application. They display graphical information on a virtual canvas that is very broad and has very high resolution. A portion of this huge canvas is seen on the display through a virtual "camera" that can pan and zoom over the surface.

ZUIs have unique requirements beyond those supported by standard 2D GUI toolkits. We list some of the requirements for the kinds of ZUIs we want to build below. Although these requirements reflect the complex nature of ZUIs, many non-zooming application-specific widgets have similar requirements:

1) The ZUI must provide support for custom application graphics that may be non-rectangular or transparent, as well as traditional interactive widgets such as buttons and sliders.
2) Large numbers of objects must be supported so that rendering and interaction performance doesn't degrade with complex scenes.
3) Objects must support arbitrary transforms and hierarchical grouping.
4) View navigations (pans and zooms) should be smooth and continuously animated.
5) Multiple representations of objects must be supported so that objects can be rendered differently in different contexts, for example, at different scales.
6) Multiple views onto the surface should be supported, both as different windows, and within the surface to be used as "portals" or "lenses".
7) Objects must be able to be made "sticky" so they stay fixed in one spot on the screen when the view changes. This is similar to a heads-up-display (HUD).
8) It must be possible to write interaction event handlers that provide for user manipulation of individual elements, and groups of objects.

The Jazz platform supports all of these requirements.

RELATED WORK

The influential InterViews framework [18] supports structured graphics and user interface components. Fresco [26] was derived from InterViews and unifies structured graphics and user interface widgets into a single hierarchy. Both Fresco and later versions of InterViews support lightweight glyphs and a class hierarchy structure similar to Jazz. However, these systems do not support large scene graphs well, or handle multiple views onto the scene graph.

They also do not support advanced visualization techniques such as fisheye views and context sensitive objects. Jazz adds new node types to the scene graph to support these additional features.

A number of 2D GUI toolkits provide higher-level support for creating custom application widgets, or provide support for structured graphics. The Tk Canvas [17] for example supports object-oriented 2D graphics, though it has no hierarchies or extensibility. Amulet [19] is an excellent toolkit that supports widgets and custom graphics, but it has no support for arbitrary transformations (such as scaling), semantic zooming, and multiple views.

The GUI toolkit that perhaps comes closest to meeting the needs of ZUIs is SubArctic [16]. It is typical of other GUI toolkits in that it is oriented towards more traditional graphical user interfaces. While SubArctic is innovative in its use of constraints for widget layout and rich input model, it has a monolithic design. In addition, it does not support multiple cameras or arbitrary 2D transformations (including scale) on objects and views.

None of these 2D GUI toolkits adopt a scene graph structure that integrates structured graphics with user interface widgets. They are all are implemented with a monolithic design. So, while it may have been possible to extend an existing toolkit to add support for zooming, it would not have been possible to pursue a minilithic design which we felt was also an important research goal.

It is possible to build ZUI applications using existing 3D scene graph tools, such as OpenInventor [5]. That may work from a structural standpoint. However, we would then be restricted to using a 3D renderer. That is problematic because 3D renderers do not support 2D business graphics or standard user interface widgets well. Typical 3D renderers, such as OpenGL, support very efficient image and triangle rendering, but do not have direct support for high quality scalable fonts, 2D complex polygons, line styles, and other standard business graphics. We have discussed these issues in depth previously [7]. We are also interested in developing scene graph nodes that apply to 2D application domains. For this domain, many of the nodes found in 3D scene graph systems are not appropriate.

There are several prior implementations of Zoomable User Interfaces toolkits. These include the original Pad system [20], and more recently Pad++ [6, 7, 8], as well as systems developed by individuals for research purposes [13, 21][22], and a few commercial ZUIs that are not widely accessible [3, 4, 23; Chapter 6].

All of these previous ZUI systems are implemented in terms of a hierarchy of objects, and are therefore superficially similar to Jazz. However, like GUI toolkits, they all use a monolithic class structure that places a large amount of functionality in a single top-level "Node" class. For example, in Pad++, the top-level `Pad_Object` class has 235 methods, and supports fading, culling, spatial indexing, stickiness, layering, etc. We needed a cleaner and more flexible approach.

THE JAZZ TOOLKIT

Jazz is a new general-purpose toolkit for creating ZUI applications using zooming object-oriented 2D graphics. Jazz is built entirely in Java and runs on all platforms that support Java 2.

Jazz uses the Java2D renderer, and is organized to support efficient animation, rapid screen updates, and high quality stills. While we could have written Jazz using other rendering engines, such as OpenGL, we picked Java2D because of its clean design and focus on high-quality 2D graphics. As previously mentioned, OpenGL does not support business graphics well. In addition, using Java2D allows us to support embedded Swing widgets, which would be impossible with OpenGL.

Jazz borrows many of the structural elements common to 3D scene graph systems, such as Sun's Java3D [1] and SGI's OpenInventor [5]. By using a basic hierarchical scene graph 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.

Figure 2 shows a complete standalone Jazz program that displays "Hello World!" where the user can pan and zoom the view. Default navigation event handlers let the user pan with the left mouse button, and zoom with the right mouse button by dragging right or left to zoom in or out, respectively. Note that Jazz automatically updates the portion of the screen that has been changed, so no manual repaint calls are needed.

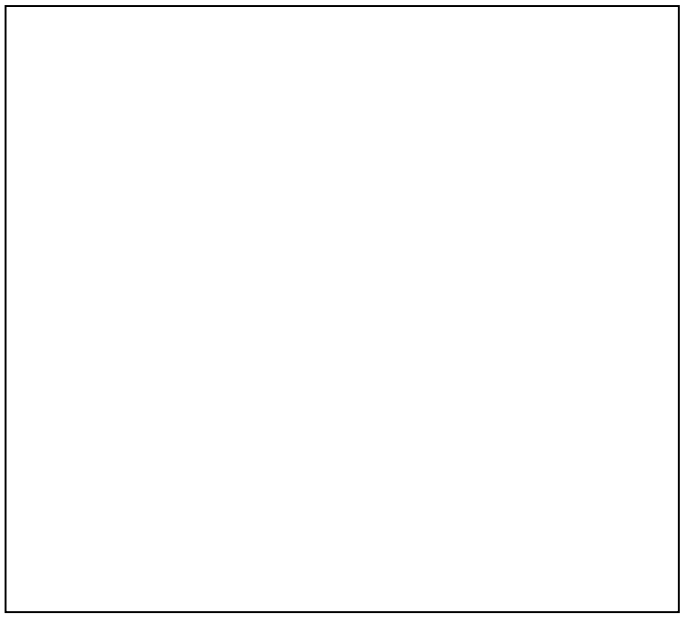


Figure 2: Complete Jazz "Hello World!" program that supports panning and zooming.

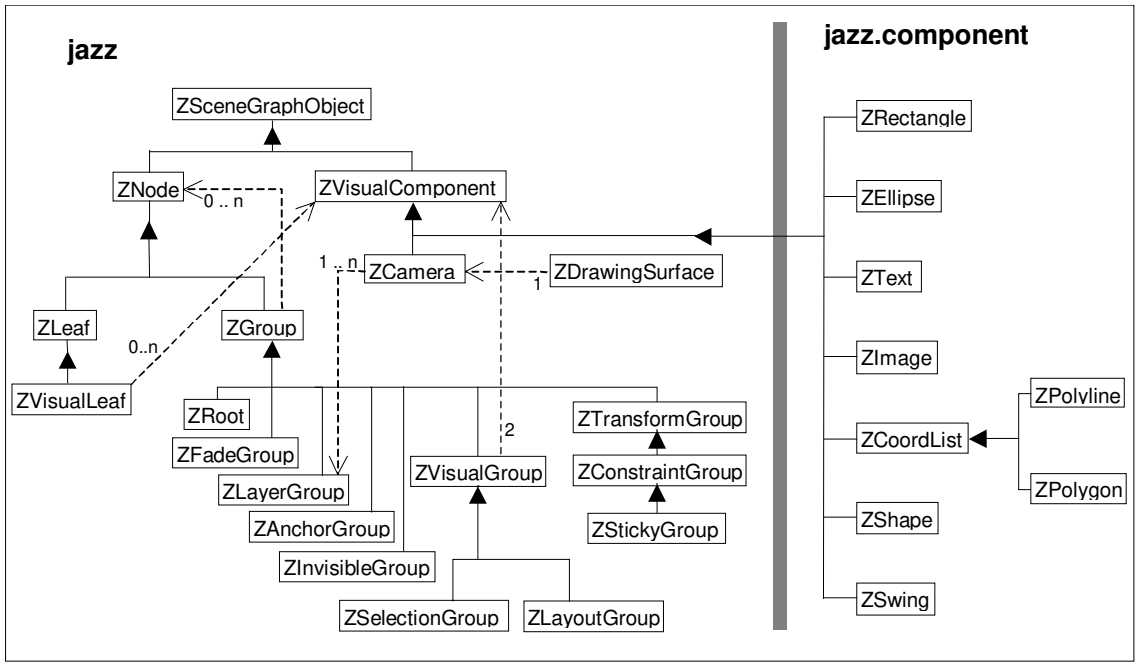


Figure 3: The Object hierarchy of Jazz.

The Jazz design follows standard 3D scene graph practices, segregating functionality into separate, non-visual grouping nodes. This approach leads to a modular scene graph design. Jazz has an extensible visual and interaction policy. It comes with a small set of visual objects and a well-defined mechanism for applications to define their own. Similarly, Jazz supports default selection, navigation, and other interaction mechanisms, but they are also designed to be modifiable by applications.

Why a 2D Scene graph?

Most application-specific widgets are built using custom data structures to support that particular application, rather than using a generic toolkit. While this approach works, it involves re-implementing many common operations from application to application. A scene graph architecture, on the other hand, provides a general-purpose reusable solution for many common operations. However this solution has costs as well. Let us look at some of the tradeoffs that come with the use of a scene graph in comparison to a custom application.

Advantages of a scene graph:

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

Disadvantages:

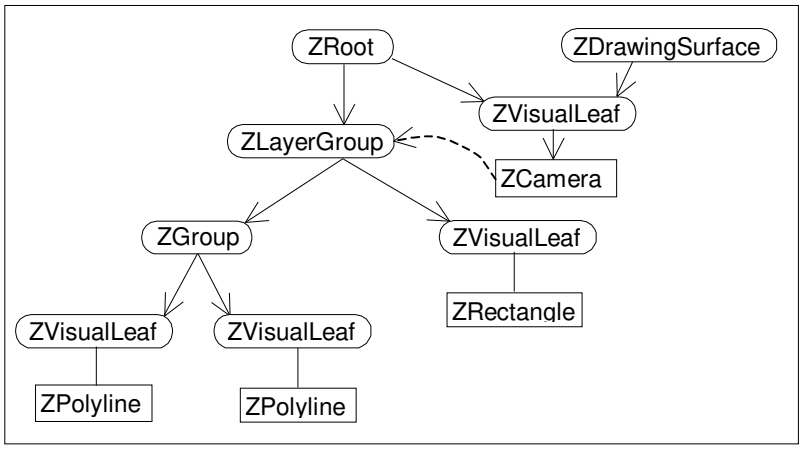
- **Footprint:** A general solution such as a scene graph will likely use more memory than a custom solution.
- **Efficiency:** It is typically more efficient to write a custom solution than to use a general-purpose scene graph. [Bradley, T., Optimizing Toolkit-Generated Graphical Interfaces, UIST 94] found that a toolkit-based solution to an Othello game ran 19 times slower than a hand-crafted solution, and consumed 18 times more memory.
- **Restrictions:** Even with the most open-ended designs, a scene graph is likely to place some restrictions on the application, which may be avoidable with a custom solution.

ARCHITECTURE

Jazz is based on three primary concepts: nodes, visual components, and cameras. Figure 3 shows the object hierarchy of Jazz's public objects that applications use. Figure 4 shows the object structure of a typical application with several objects and a camera.

Nodes and Visual Components

The Jazz scene graph consists of a hierarchy of *nodes* that represent relationships between objects. The base node type (`ZNode`) is very simple. There are more complex node types, whose features are only paid for when used. Hierarchies of nodes can be used to implement "groups"



and "layers" that are found in most drawing programs, and to facilitate moving a collection of objects together.

Scene graph nodes have no visual appearance on the screen. Rather, there are special objects, called *visual components*, which are attached to certain nodes in a scene graph (specifically to *visual leaf nodes* and *visual group nodes*), and which define geometry and color attributes.

In other words, nodes establish *where* something is in the scene graph hierarchy, whereas visual components specify *what* something looks like. All nodes have a single parent, and follow a strict tree hierarchy. Visual components can be reused – the same visual component can appear in multiple places in the scene graph, and thus have multiple parents.

There is a clear separation between what is implemented in a node and what is handled by a visual component. Nodes contain all object characteristics that are passed on to child nodes. For example, nodes are used to provide affine transforms (for translating, rotating, scaling, and shearing child nodes), culling sub-trees according to magnification, and defining transparency for groups of objects. Each of these characteristics modify all of that node's descendants.

Visual components are purely visual. They do not have a hierarchical structure (they do not even specify a transformation). Each visual component simply specifies how to render itself, what its bounds are, and how to pick it (i.e. how to detect if the mouse is over the component).

This split between nodes and visual components clearly separates code that is aware of the scene graph hierarchy from code that operates independently of any hierarchy. It enables hierarchical structuring of scene graph nodes, and also reuse of visual components. It also 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 scene graph without affecting the grouping or position of objects.

Cameras

A camera is a visual component that displays a view of a Jazz scene graph. It specifies which portion of the scene graph is visible using an affine transform. Multiple cameras can be setup looking at a single scene graph, each defining its own view of the scene graph.

Drawing Surfaces vs. Internal Cameras

Cameras are usually mapped to a *drawing surface*. This encapsulates a Java `Graphics2D` class, which supports 2D rendering. The drawing surface is usually associated with a Jazz Canvas, so that the user can see the surface on their display. The Jazz Canvas is implemented as a Java Swing component, so ZUI interfaces can be embedded in any Swing application, wherever a Swing JComponent widget is expected. The `Graphics2D` of a drawing surface can also output to an off-screen buffer, or a printer. With this mechanism, a Jazz surface can be used to display, print, or to render into a buffer so an application can grab the pixels that were rendered.

In addition to being mapped to drawing surfaces, cameras can also be treated just like any other visual component – they can be embedded in a Jazz scene graph, so that nested views of a zoomable surface can be embedded recursively in a scene. Cameras used in this way are called *internal cameras*, and act like nested windows within the world that themselves look onto the world, or onto a different world (in previous ZUI implementations, we called these "portals" [25].)

Layers

Each camera contains a list of *layer nodes* specifying which layers in the scene graph it can see. A camera renders itself by first rendering its background, and then rendering all the layers in its layer list. This approach lets an application build a single very large scene graph and control which portion of the scene graph are visible in each camera.

Layers can be made temporarily invisible within a specific camera by removing it from the camera's layer list. Alternatively, a special node type called an "invisible group" node can be inserted into the scene graph to make all the children of a layer invisible. Changing the order of the layer nodes within a camera's layer list changes the drawing order of entire layers.

Rendering

Nodes are rendered in a top-to-bottom, left-to-right depth first fashion. Consequently, visual components are rendered in the order that their associated nodes appear in the scene graph. Changing the order of a node within a parent node will change the rendering order of the associated visual component.

Culling

All scene graph objects include a method to compute their bounding rectangle. Jazz uses this to decide which objects are visible, and thus avoid rendering or picking objects that are not visible in a given view. Bounds are cached at each node in the current relative coordinate system. Objects that regularly change their dimensions can specify that their bounds are *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.

Events

Jazz supports interaction through event listeners. These follow the standard Java event model and may be attached to any node in the Jazz scene graph. There are two categories of events – input events and object events. Input events result from user interaction with a graphical object, such as a mouse press. Object events result from a modification to the scene graph, such as a transformation change, or a node insertion. All events can be handled by attaching listeners to scene graph nodes. There can be multiple listeners per node. Unlike the standard Swing and AWT listener model, in Jazz by default each input event is passed up the tree to the listeners on ancestor nodes. However, if a listener consumes the event, the event is not passed on any further. With this mechanism, custom event listeners may be written for specific nodes that correspond to graphical items – or a listener may be attached higher in the scene graph tree, which then provides interaction support for the entire subtree below the listener. Event listeners can be written in either a specific or very general manner depending on the application's needs.

Jazz dispatches all mouse events to the node (and potentially its ancestors) returned by a *pick* operation at the location of the original mouse event on the Jazz drawing surface. Before dispatching the event, Jazz modifies the event records to reflect the local coordinates of the picked component. Visual component event handlers can therefore work in their local coordinate system.

Jazz comes with event handlers for several basic tasks, such as navigation, selection, and hyperlinks. Applications are free to use these or define their own.

COMPOSING FUNCTIONALITY USING NODE TYPES

A basic design goal of Jazz is to maintain a decoupled design so that different features do not depend on each other and so that applications only pay for features when they use them. This led us to keep the core ZNode very simple, and to add extra features by introducing new node types which are inserted into the scene graph as needed.

For instance, since not all nodes will be transformed, the core ZNode type does not contain a transform. Instead, a transform node is created when needed and inserted above any node that should be transformed.

Developers are encouraged to achieve complex functionality by composing simple node types in a scene graph rather than by using subclassing and inheritance.

To validate the practicality of this idea for ZUIs, we have developed a number of Jazz node types, each implementing a specific functionality suitable for ZUI applications, and each remaining small and manageable. In this section we discuss some of the node types we have created.

Jazz includes nodes to support layers, selections, transparency, hyperlinks, fading, spatial indexing, layout, and constraints. In this section we discuss some of these node types.

Selection and Hyperlink Nodes

Some node types associate extra characteristics with a portion of the scene graph. These extra nodes act as "decorators" following a standard object oriented design pattern [15]. They wrap the core functions of the nodes below them, adding extra functionality. For example, we have written a Jazz selection decorator node that draws its children, and then draws a selection box.

Similarly, Jazz defines a link node, which is used to create spatial hyperlinks. The link node associates the destination of a spatial hyperlink with a node, but does so without modifying the node and without the node's knowledge. When the user moves the mouse over a link node, it presents an arrow visual component to show what the link refers to. Clicking on the link navigates the camera to the linked object.

Position and Layout Nodes

The position and scale of objects is specified in Jazz by inserting transformation nodes into the hierarchy. Active layout managers can also be utilized by inserting a layout node into the hierarchy. We have developed a ZLayoutGroup that uses a layout manager analogous to the Java AWT and Swing layout managers. Layout managers can be inserted at different levels of the scene graph, yielding hierarchical layouts. Applications may define new layout managers, or use one of the built-in layouts. Currently, Jazz has two layout managers: a hierarchical tree layout manager, which will layout any subtree of the scene graph in a standard tree structure, and a path layout manager, which will position a set of nodes along any path. The hierarchical tree layout manager is interesting in that it

shows lines indicating the linkages between nodes in the tree using a special visual component.

Constraint Nodes

We have developed nodes that use dynamic constraints to position their children. Currently we use these constraint nodes to implement "Sticky objects" - portions of the scene graph that are associated with a particular camera and that do not move when the camera viewpoint is changed. Sticky nodes subclass a constraint node contains that a transform. They modify the transform by setting it to the inverse of a specified camera's view transform whenever the camera's view changes. The subtree rooted at the sticky constraint node then does not move as the viewpoint changes. It is as if they are stuck to the camera's lens.

Culling Nodes

A basic characteristic of zoomable applications is that there can be a large number of objects in a given scene graph, many of which are not visible in a given view of the graph. For example, in a zoomed-in view, only very small objects are visible, whereas in a zoomed-out view, only large objects need be shown. Thus, it is important to efficiently traverse the scene graph, culling invisible objects. Sometimes simple bounds-based culling is not sufficient. We have developed two additional mechanisms to support culling. First, "fade" groups can be inserted in the scene graph to cull a subtree when it appears larger or smaller than a specified magnification in a view (fade groups use alpha blending to smooth this transition - hence the name). Second, a "spatial index" node can be inserted in the scene graph to provide fast access to the visible children of that node. The Spatial Index node implements an RTree index [24] which is effective when there are many nodes, but only a small percentage of them are visible at a given time. This is quite common in ZUIs since this typically occurs whenever the view is zoomed in.

CUSTOM VISUAL COMPONENTS

To define new visual components, applications extend the `ZVisualComponent` class and define two functions. The new object defines how to paint itself and how big it is. In addition, visual components may define picking methods if the object is not rectangular, so Jazz knows when the pointer is over the object.

Legacy Java Code

One of our motivations for splitting components and scene graph nodes in two was to make it easy to import non-zooming components and legacy applications into a zooming context. In Jazz, visual components can be easily defined to wrap legacy Java code that is written without awareness of Jazz. Those components can then be zoomed and interacted with by placing them in a scene graph. For example, it is possible to take some pre-existing code that draws a scatter plot and make it available as a Jazz visual component on a zooming surface.

This technique has been used to wrap existing code in two large systems. The first is a graphical simulation system from a research group at Los Alamos National Labs. We defined a new visual component that wrapped their core visual component, and were able to place their entire visualization inside of Jazz, complete with zooming and multiple views and interaction in about half of a day. The second was the LEIF system developed by DTAI [2]. This is a large information framework system with a major visualization component. With a similar technique, they were able to wrap their core object type with a Jazz visual component, and get their entire application to appear inside of Jazz.

Swing Visual Components

Any lightweight Java Swing component can be embedded into a Jazz scene graph by placing it in a Jazz ZSwing visual component in the scene graph. The Swing component can then be panned and zoomed like other Jazz components. This means that fully functioning existing Java Swing code with complete GUIs can be embedded into a zooming surface, and mixed and matched with custom graphics within Jazz. For example, a Swing interface with a table and buttons could be placed on a zooming surface and overlaid with an application-specific visualization. The Swing components can be manipulated in exactly the same way as other Jazz components, including applying rotation, scale, transparency, and multiple views. The embedded Swing integration occurs transparently to the Swing widget and to other nodes in the scene graph. An applet demonstrating Swing widgets embedded in Jazz is available at the Jazz website.

To implement embedded Swing widgets inside of Jazz, the widgets' input and output had to be remapped to accommodate their transformed rendering. Mouse input in Swing normally takes the pointer's screen location directly to the Swing component's local coordinate system. This mapping is not as straightforward since embedded Swing widgets may be arbitrarily transformed. So the ZSwing visual component registers listeners for mouse events, and forwards any events it receives to the underlying Swing component in its coordinate system.

Similarly, the ZSwing visual component must also alter repaint requests made by Swing components embedded in Jazz. These Swing repaint requests assume rendering in a traditional GUI rather than one arbitrarily transformed. The ZSwing visual component must reroute these repaints through the Jazz scene graph, including multiple views, to properly transform the Java Graphics2D object to be used by the Swing component for rendering.

CREATING APPLICATION SPECIFIC WIDGETS

To test Jazz, we developed a number of prototypes of application-specific widgets. These widgets explore various aspects of ZUIs and general graphical application design. In this section, we report briefly on these widgets.

Basic ZUI Application

To understand the requirements of ZUIs as well as the structure of Jazz, we created a simple zoomable application

in Jazz. We built a graph editor that lets users draw a graph with many nodes that are connected by links. The links follow nodes that the user moves. The user can draw very large graphs and the view may be zoomed in or out on demand. Nodes can be grouped. When zoomed out, these node groups fade out and are replaced with a group node that represents an abstraction of the elements of the group. Finally, to support the user in understanding global context as well as detail, multiple views can be brought up simultaneously and the zoomed out views will show where the zoomed in views are. The application is available as a Web applet at http://www.cs.umd.edu/hcil/jazz/play .

It is difficult to build this application using existing GUI toolkits. GUI toolkits don't directly support zooming, multiple views, and multiple levels of information, so the application would have to manage all of these details. Building this application in Jazz was straightforward.

Since zooming, multiple levels of representation, and multiple views are all directly supported by Jazz, writing the graph drawing application was just a matter of adding appropriate nodes to the scene graph for each element of application. In addition, Jazz's support for interaction through event listeners makes it easy to add nodes and edit the graph. Object change events are generated when an object moves, so an event listener was attached to the nodes in order to update the connecting arrows so they always follow the nodes. Finally, Jazz takes care of screen updates, hierarchical transforms, etc. which simplifies application programming considerably.

Fisheye Views

The scene graph model in Jazz makes it easy to support advanced graphical features. For instance, while Jazz directly supports geometric zooming of entire scenes, it is also possible to create "fisheye" visualizations where each object is scaled according to some degree of interest function [14].

We implemented a simple fisheye view in Jazz by putting a special Fisheye decorator node above every object that we wanted the fisheye effect applied to. The fisheye node is really just a special transform node that dynamically computes its scale according to a function. Admittedly, this is a simplified fisheye view as entire objects are scaled whole rather than being distorted. However, adding more functionality would be trivial. We are currently exploring other fisheye techniques in Jazz.

Context Sensitive Objects

By default, Jazz objects support a single presentation style. It is also desirable to be able to support *multiple* representations of a single data model. That is, to support different visual displays of objects in different contexts.

Applications can easily use Jazz to present different visual representations of data in different circumstances. Many different kinds of context can be used to influence how and what gets drawn, so we sometimes call this technique *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* (the size the object is rendered at) and *camera* (the camera the object is 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, 25].

Jazz's standard visual components render themselves the same way every time, except during interactions, when they may render themselves at lower resolution for efficiency. Applications can define new visual components and nodes whose paint methods are context sensitive.

Standard software engineering approaches call for decoupled representations. 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* [15] for another visual component, and can delegate between them. A more elaborate delegation scheme is described by Fox [12].

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`, and `pick` methods to call the active visual component.

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. Figure 5 shows the basic structure of the scene graph for our delegator that supports semantic zooming.

LAZY EVALUATION

A drawback of the “minilithic” approach adopted by Jazz is that it places a burden on the application programmer since they must manage a scene graph containing many nodes and node types. Adding a new element to a scene can take several steps.

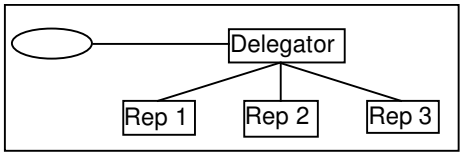


Figure 5: One way to implement semantic zooming in Jazz. The *Delegator* chooses which representation to use to paint itself depending on the current magnification.

In practice there is typically a primary node that the application cares about (usually the visual leaf node) and then there are several decorator nodes above it. We have implemented special support for managing these kinds of scene graph structures, using the notion of scene graph *editor* objects.

An editor instance can be created for any node on the scene graph. It has methods for obtaining parents of the node that are of a specific type. It uses lazy evaluation to create those parent nodes as they are required. Jazz maintains a special bit in each node specifying if it is created by an editor. With this structure, if an application wants to obtain a transform node for a given node in the scene graph, it can simply call:

```
node.editor().getTransformGroup().
```

If the node does not have a transform node associated with it, a new transform node is created and inserted above the node. Otherwise the existing transform node is returned. The `editor()` method actually calls a factory [10] to create the editor, so applications can define custom mechanisms for editing nodes.

CURRENT STATUS

Everything described in this paper is currently implemented in Java 2 and is publicly available as open source software at http://www.cs.umd.edu/hcil/jazz. Jazz is being actively used by a number of research groups, has been used in university courses in and outside of the University of Maryland, and is being used within two commercial products currently being developed.

The Jazz distribution comes with a sample application called HiNote 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, 11] that provides collaborative storytelling tools for children, available at http://www.cs.umd.edu/hcil/kidpad. The second is an authoring tool for creating presentations. The authoring tool builds on our experience making zoomable presentations over the past several years. The authoring tool works as a plug-in for Microsoft PowerPoint allowing any existing presentation to be brought into a zoomable space. This tool is not yet publicly available.

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 representations. It is a descendent from previous Zoomable User Interfaces that we have built in the past.

While Jazz does not introduce many substantially new individual ideas, it is novel in bringing together a variety of techniques from different domains. Jazz takes scene graphs from 3D graphics, screen and interaction techniques from 2D widgets, functionality from previous ZUI systems, and puts these elements together with clean decoupled object oriented design. Using Jazz, developers can write serious zoomable applications and advanced visualizations with a clarity and efficiency that has not been possible before.

The biggest contribution of Jazz is the creation of a graphics toolkit built using a "minilithic" design. By encouraging composition over inheritance, the Jazz feature-set is highly decoupled. This makes the code easier to maintain and extend compared with monolithic approaches. We and others have used Jazz to build a variety of applications. This proof by example demonstrates that the approach has potential. There are, however, trade-offs with any design, and the minilithic approach also has costs.

Our experience with Jazz so far shows us that the biggest concerns with the Jazz design is ease-of-use and efficiency. The downside of a minilithic approach is that the application developer must manage many more objects than with a more traditional design. While you only pay for the features you use, you need a new node instance for each feature. While we have attempted to minimize this burden through the use of "editors", the developer still has to be aware of many node types.

Another basic issue is efficiency of a scenegraph-based solution compared to an entirely custom solution. An alternative to Jazz for some visualizations would be to build a custom data structure representing the model, and then to build a render method that simply walks through the model, rendering the entire scene. This approach is simple and very efficient. We believe that for simple applications, that kind of custom implementation may be best. The advantages of a scenegraph-based approach don't appear until the application requires features such as selection, layers, fading, or spatial indexing. Once a number of these features are required, the custom approach becomes very tedious, and difficult to maintain. So, while we do not yet have any quantitative data to inform a designer, it appears that there is a threshold of complexity above which Jazz provides more benefit than cost.

While we have not yet performed a rigorous quantitative performance analysis, we and others have used Jazz for a number of applications. We have consistently found that its performance is good. The primary bottleneck appears to be rendering large numbers of objects, and not the overhead incurred by traversing and maintaining the scenegraph.

We look forward to continuing the development of Jazz, and increasing our understanding of the trade-offs of the minilithic scenegraph design that we have chosen. As Jazz is used for more projects, we will gain enough experience to carefully analyze the construction of systems using Jazz, and compare them to alternative approaches.

ACKNOWLEDGMENTS

We enjoyed our collaborations with those involved with Pad++, especially Jim Hollan, Jason Stewart, Allison Druin, Britt McAlister, George Furnas and Ken Perlin.

We would like to thank our fellow members of the HCIL, especially the students in the seminar on ZUIs that had the

patience to use early versions of Jazz and helped to identify its "features". Our thanks to Jim Mokwa and Maria Jump for their contributions to Jazz. We greatly appreciate the careful reading of earlier versions of this paper by Bay-Wei Chang, Jason Stewart, and Allison Druin.

Most importantly, the many users of Jazz have helped us design, debug, and understand the requirements of Jazz and have made Jazz much more broadly useful than would have been possible otherwise. Two important early users were David Thompson at LANL and Mike Behrens at DTAI.

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

REFERENCES

1. Java [Web Page] (2000). URL http://www.javasoft.com.
2. LEIF [Web Page] (2000). URL http://leif.dtai.com.
3. MerzCom [Web Page] (2000). URL http://www.merzcom.com/.
4. Perspecta [Web Page] (2000). URL http://www.perspecta.com/.
5. SGI OpenInventor [Web Page] (2000). URL http://www.sgi.com/Technology/Inventor/.
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*, pp. 3-31.
7. Bederson, B. B., & Meyer, J. (1998). Implementing a Zooming User Interface: Experience Building Pad++. *Software: Practice and Experience, 28*(10), pp. 1101-1135.
8. Bederson, B. B., Wallace, R. S., & Schwartz, E. L. (1993). Control & Design of the Spherical Pointing Motor. *In Proceedings of IEEE International Conference on Robotics and Automation (ICRA 93)* New York: IEEE,
9. Benford, S., Bederson, B. B., Åkesson, K.-P., Bayon, V., Druin, A., Hansson, P., Hourcade, J. P., Ingram, R., Neale, H., O'Malley, C., Simsarian, K. T., Stanton, D., Sundblad, Y., & Taxén, G. (2000). Designing Storytelling Technologies to Encourage Collaboration Between Young Children. *In Proceedings of Human Factors in Computing Systems (CHI 2000)* ACM Press, pp. 556-563.
10. Booch, G. (1994). *Object-Oriented Analysis and Design With Applications*. Addison-Wesley.
11. 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.
12. Fox, D. (1998). Composing Magic Lenses. *In Proceedings of Human Factors in Computing Systems (CHI 98)* ACM Press, pp. 519-525.
13. Fox, D. (1998). *Tabula Rasa: A Multi-scale User Interface System*. Doctoral dissertation, New York University, New York, NY.
14. Furnas, G. W. (1986). Generalized Fisheye Views. *In Proceedings of Human Factors in Computing Systems (CHI 86)* ACM Press, pp. 16-23.
15. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
16. Hudson, S. E., & Stasko, J. T. (1993). Animation Support in a User Interface Toolkit. *In Proceedings of User Interface and Software Technology (UIST 93)* ACM Press, pp. 57-67.
17. John K. Ousterhout. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.
18. Linton, M. A., Vlissides, J. M., & Calder, P. R. (1989). Composing User Interfaces With InterViews. *IEEE Software, 22*(2), pp. 8-22.
19. Myers, B. A., McDaniel, R. G., Miller, R. C., Ferrency, A. S., Faulring, A., Kyle, B. D., Mickish, A., Klimovitski, A., & Doane, P. (1997). The Amulet Environment: New Models for Effective User Interface Software Development". *IEEE Transactions on Software Engineering, 23*(6), pp. 347-365.
20. 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.
21. Perlin, K., & Meyer, J. (1999). Nested User Interface Components. *User Interface and Software Technology (UIST 99)* ACM Press, pp. 11-18.
22. Pook, S., Lecolinet, E., Vaysseix, G., & Barillot, E. (2000). Context and Interaction in Zoomable User Interfaces. *In Proceedings of Advanced Visual Interfaces (AVI 2000)* ACM Press, (in press).
23. Raskin, J. (2000). *The Humane Interface*. Reading, Massachusetts: Addison Wesley.
24. Samet, H. (1990). *The Design and Analysis of Spatial Data Structures*. Addison-Wesley.
25. 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.
26. Tang, S. H., & Linton, M. A. (1994). Blending Structured Graphics and Layout. *In Proceedings of User Interface and Software Technology (UIST 94)* ACM Press, pp. 167-174.