Toolkit Design for Interactive Structured Graphics

Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer

Abstract—In this paper, we analyze toolkit designs for building graphical applications with rich user interfaces, comparing polylithic and monolithic toolkit-based solutions. Polylithic toolkits encourage extension by composition and follow a design philosophy similar to 3D scene graphs supported by toolkits including Java3D and OpenInventor. Monolithic toolkits, on the other hand, encourage extension by inheritance, and are more akin to 2D Graphical User Interface toolkits such as Swing or MFC. We describe Jazz (a polylithic toolkit) and Piccolo (a monolithic toolkit), each of which we built to support interactive 2D structured graphics applications in general, and Zoomable User Interface applications in particular. We examine the trade offs of each approach in terms of performance, memory requirements, and programmability. We conclude that a polylithic approach is most suitable for toolkit builders, visual design software where code is automatically generated, and application builders where there is much customization of the toolkit. Correspondingly, we find that monolithic approaches appear to be best for application builders where there is not much customization of the toolkit.

Index Terms—Monolithic toolkits, polylithic toolkits, object-oriented design, composition, inheritance, Zoomable User Interfaces (ZUIs), animation, structured graphics, Graphical User Interfaces (GUIs), Pad++, Jazz, Piccolo.

1 Introduction

Application developers rely on User Interface (UI) toolkits such as Microsoft's MFC and .NET Windows Forms, and Sun's Swing and AWT to create visual user interfaces. However, while these toolkits are effective for traditional widget-based applications, they fall short when the developer needs to build a new kind of user interface component-one that is not bundled with the toolkit. These components might be simple widgets, such as a range slider or more complex objects, including interactive graphs and charts, sophisticated data displays, timeline editors, zoomable user interfaces, or fisheye visualizations.

Developing application-specific components usually requires significant quantities of custom code to manage a range of features, many of which are similar from one component to the next. These include managing which areas of the window need repainting (called *region management*), repainting those regions efficiently, sending events to the internal object that is under the mouse pointer, managing multiple views, and integrating with the underlying windowing system.

Writing this code is cumbersome, yet most standard 2D UI toolkits provide only rudimentary support for creating custom components—typically, just a set of methods for drawing 2D shapes and methods for listening to low-level events.

Some toolkits such as Tcl/Tk [19] include a "structured canvas" component, which supports basic structured graphics. These canvases typically contain a collection of graphical 2D objects, including shapes, text, and images. These components could in principal be used to create application-specific components. However, structured canvases are designed primarily to display graphical data, not to support new kinds of interaction components. Thus, for example, they usually do not allow the application to extend the set of objects that can be placed within the canvas. We have found that many developers bypass these structured canvas components and follow a "roll-your-own" design philosophy, rewriting large quantities of code and increasing engineering overhead, particularly in terms of reliability and programmability. There are also commercial toolkits available such as Flash [6] and Adobe SVG Viewer [2]. But, these approaches are often difficult to extend and integrate into an application.

We believe future user interface toolkits must address these problems by providing higher-level libraries for supporting custom interface components. However, there is still an open question regarding which design philosophy to adopt for these higher-level toolkits. The core issue we address here is whether toolkits should be designed so that the inevitable complexity and extension of the components are supported primarily through composition (which we call polylithic) or inheritance (which we call monolithic).

In this paper, we consider these two design approaches for interactive structured graphics toolkits through two toolkits we built: Jazz,[1] a polylithic toolkit; and Piccolo,[2] a

• *The authors are with the Human-Computer Interaction Laboratory, Institute for Advanced Computer Studies, Computer Science Department, University of Maryland, College Park, MD 20742.*
E-mail: {bederson, jesse, meyer}@cs.umd.edu.

Manuscript received 16 Sept. 2003; accepted 16 Mar. 2004.
Recommended for acceptance by D. Weiss.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0145-0903.

1. The name Jazz is not an acronym, but rather is motivated by the 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 Mozilla Public License, and is available at: http://www.cs.umd.edu/hcil/jazz.

2. The name Piccolo is motivated by the music connection of Jazz and Swing, and because it is so small (approximately one tenth the size of Jazz). Piccolo is open source software according to the Mozilla Public License, and is available at: http://www.cs.umd.edu/hcil/piccolo.

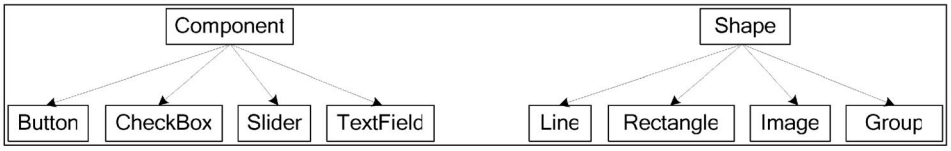


Fig. 1. Class hierarchy of a GUI toolkit (left) and a structured-graphics toolkit (right).

monolithic toolkit [14]. We provide a qualitative and quantitative analysis to compare code written using these two toolkits, looking at application speed and size, memory usage, and programmability. We are concerned primarily with issues related to data presentation, painting, event management, layout, and animation. We do not address many issues that modern UIs often include such as accessibility, localization, keyboard navigation, etc.

2 Related Work

There are a number of research [18], [22] and commercial [4], [19] structured canvas toolkits available. However, most structured canvas components provide a fixed vocabulary of the kinds of shapes they support within the canvas. It can be difficult to create new classes of objects to place on the canvas.

The InterViews framework [21], for example, supports structured graphics and user interface components. Fresco [29] 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 provide a hierarchy of graphical objects. However, these systems handle large numbers of visual objects poorly and do not support multiple views onto a single scene graph, or dynamic scene graphs. They also do not support advanced visualization techniques such as fisheye views and context sensitive objects.

A number of 2D GUI toolkits provide higher-level support for creating custom application widgets, or provide support for structured graphics. Amulet [22] is a toolkit that supports widgets and custom graphics, but it has no support for arbitrary transformations (such as scaling), and multiple views.

The GUI toolkit that perhaps comes closest to meeting the needs for custom widgets is SubArctic [18]. It is typical of other GUI toolkits in that it is oriented toward more traditional graphical user interfaces. While SubArctic is innovative in its use of constraints for widget layout and rich input model, it does not support multiple cameras or arbitrary 2D transformations (including scale) on objects and views.

Morphic [27] is another interesting toolkit that supports many of our listed requirements. Morphic's greatest strength is in the toolkits uniform and concrete implementation of structured graphics, making it both flexible and easy to learn. But, Morphic's support for arbitrary node transforms and full screen zooming and panning is weak. It also provides no support for multiple cameras.

There were several prior implementations of Zoomable User Interfaces toolkits as well. These include the original Pad system [23] and, more recently, Pad++ [11], [12], [13], [15], as well as other systems [16], [24], [25], and a few commercial ZUIs that are not widely accessible [26, Chapter 6]. All of these previous ZUI systems are implemented in terms of a hierarchy of objects. However, like GUI toolkits, they use a monolithic class structure that places a large amount of functionality in a single top-level class.

3 Polylithic versus Monolithic Designs

Object-oriented software engineers advocate the use of "concrete" class hierarchies in which there is a strong mapping between software objects and real-world things. These hierarchies tend to be easier for people to learn [20]. Modern GUI toolkits typify this design, using classes that strongly mirror real-world objects such as buttons, sliders, and containers. Similarly, toolkits for two-dimensional structured graphics usually adopt a class hierarchy whose root class is a visual object, with subclasses for the various shapes, lines, labels and images (Fig. 1).

In these toolkits, runtime parent/child relationships are used to define a *visual tree*, where each object in the tree is mapped to a portion of the display and has a visual representation. Many of the complex mechanisms necessary for modern graphical interfaces (navigation, rendering, event propagation) are contained within the class structure.

Three-dimensional graphics toolkits provide an important counterexample. Toolkits such as Java3D [5] and OpenInventor [7] use a more abstract model. Here, distinct classes are used to represent materials, lighting, camera views, layout, behavior, and visual geometry. Instances of these classes are organized at runtime in a *semantic graph* (usually a DAG) called a scene graph. Some nodes in the scene graph correspond to visual objects on the screen, but many of the nodes in the scene graph represent nonvisual data such as behaviors, coordinate transforms, cameras, or lights (Fig. 2). This design provides opportunities for introducing abstractions and promoting code reuse, though the downside is that it tends to yield a greater number of overall classes. While scene graphs are very common in 3D graphics, they are rarely used with 2D graphics.

The most important distinction between these two approaches is the style that is used to combine and add new features. The 3D scene graphs focus on the use of

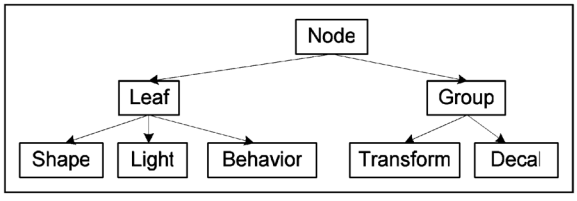


Fig. 2. Class hierarchy of a typical 3D graphics toolkit.

runtime composition to organize the many small classes in support of the desired solution. Each class represents an isolated bit of functionality, and they are designed independent of each other. In this sense, polylithic designs are more similar to Prototype-based programming systems such as Self [30] or ECMAScript [3], which use runtime instancing to create derived types.

> We define **polylithic** toolkits as being those that primarily use run-time *composition* to extend functionality.

The focus on design through composition offers the potential of reusability and customizability. By having features instantiated in small independent classes, it is likely that these classes will be useful in other contexts and may be combined in unforeseen ways. Furthermore, the ability to compose a scene graph at runtime makes such systems a good match for authoring programs which can support nonprogrammers to construct new models. This is evidenced by a broad set of 3D modeling tools such as SoftImage [8] and 3D Studio Max [1]. We call the 3D toolkit design approach *polylithic* because it consists of many small classes combined together with composition, each representing an isolated bit of functionality where several are often linked together to represent one semantic unit.

An alternative design approach is to use compile-time inheritance to define new widgets. This approach, as is commonly used in 2D GUI toolkits, defines a single large base class with support for all the functionality commonly used in the toolkit's widget set. New widgets, even ones with only modest new behavior are created by defining a new class that inherits from an existing one.

> We define **monolithic** toolkits as being those that primarily use compile-time *inheritance* to extend functionality.

We call this concrete design approach *monolithic* because these toolkits have a few large classes containing all the core functionality likely to be used by applications. These toolkits tend to be complex and have large numbers of methods. The functionality provided by each class is hard to reuse in new widgets. To support code reuse, toolkit designers often place large amounts of generally useful code in the top-level class that is inherited by all of the widgets in the toolkit. This decision leads to a complex hard-to-learn top-level class. In addition, application developers are forced to accept the functionality provided by the toolkit's top-level class—they often cannot add their own reusable mechanisms to enhance the toolkit.

3.1 Composing Functionality

A design goal of polylithic systems is to compose functionality by using a runtime graph of nodes. Each node in the runtime graph contributes a specific piece of functionality according to its type. Polylithic systems thus shift complexity from the static class hierarchy into the runtime data structure. This contrasts strongly with monolithic systems, which rely heavily on the static class inheritance hierarchy to compose functionality.

Consider the following example to point out the distinctions between the two approaches. Let us imagine a simple application that renders 50 rectangles to the screen at random positions. Then, we extend that application so the transparency of each rectangle is dependent on its position. The point here is to understand what is involved in modifying the application. The example here is trivial, but the same ideas apply when, say, extending a button widget to include a checkbox.

In our example, we will start with a base `Node` class and a `Rectangle` class which extends `Node` that renders a rectangle. The traditional monolithic approach extends `Rectangle` to create a `FadeRectangle` with the desired functionality. The new application then creates 50 `FadeRectangles`. The polylithic approach, on the other hand, creates a simpler `Fade` object by extending `Node` and then creates a scene graph where each new rectangle consists of a `Fade` object with a child `Rectangle` object (Fig. 3 shows the class structure and runtime scene graph of this example, and the complete code is included in the digital library as supplementary material).

In this example, each approach yields identical output. But, by using composition rather than inheritance to add transparency, the same `Fade` class could be reused for many different applications, not just for rectangles. And, of course, similar functionality can be achieved in general using polylithic toolkits resulting in greater reusability in general.

However, this example also immediately demonstrates the main drawback of polylithic systems: The application code is longer than that for the monolithic system because the application programmer has to create, understand, and manage more objects. Monolithic systems also tend to be more familiar to programmers used to languages such as Java or C#. On the other hand, because polylithic systems explicitly separate node types based on their functionality, they potentially encourage designers to think of useful abstractions from the outset, yielding more flexible class hierarchies.

The flexibility of a polylithic approach is likely to be especially useful when applications and objects are built dynamically at runtime. This frequently happens in prototyping systems and within design tools. In these contexts, it could be quite powerful to dynamically load a new object (potentially downloaded from the Web) and insert it into an existing scene graph-changing the behavior or look of an object at runtime in ways not imagined by the original author. Thus, there is a trade off between application code complexity and flexibility.

Monolithic and Polylithic designs as we have described them are ideals, but, in practice, real toolkits are usually a combination of each. Even in the example above, the polylithic design has a base class with some functionality from which the other nodes inherit. Similarly, even in a monolithic toolkit, there is nothing stopping a developer from creating, say, a transparency node and inserting it into the scene graph using composition. Nevertheless, the differences in design are real and these monolithic and polylithic patterns can be found as we describe in the above-referenced literature. Furthermore, the design of a toolkit influences its users to build on it following the suggested patterns. So, this paper does not attempt to partition toolkits exclusively into one camp or the other. Rather, we hope that, by identifying and studying these

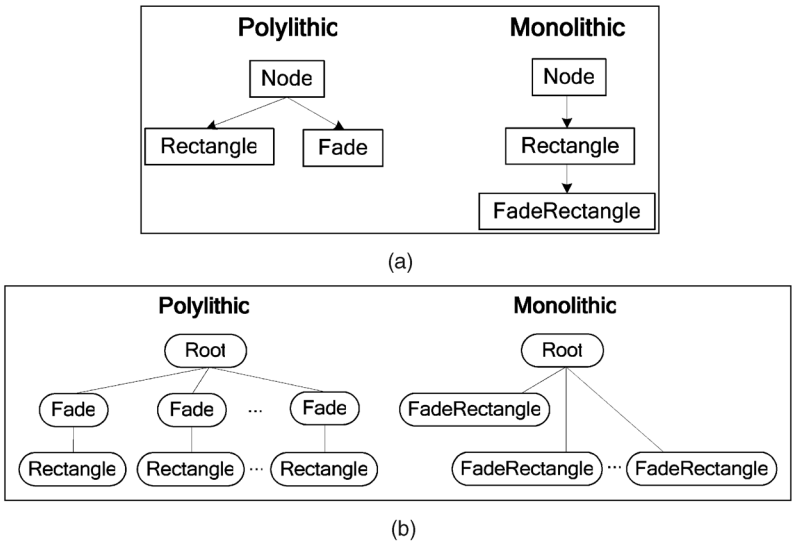


Fig. 3. (a) Polylithic and monolithic class hierarchy for rectangle example. (b) Polylithic and monolithic runtime scene graph for rectangle example.

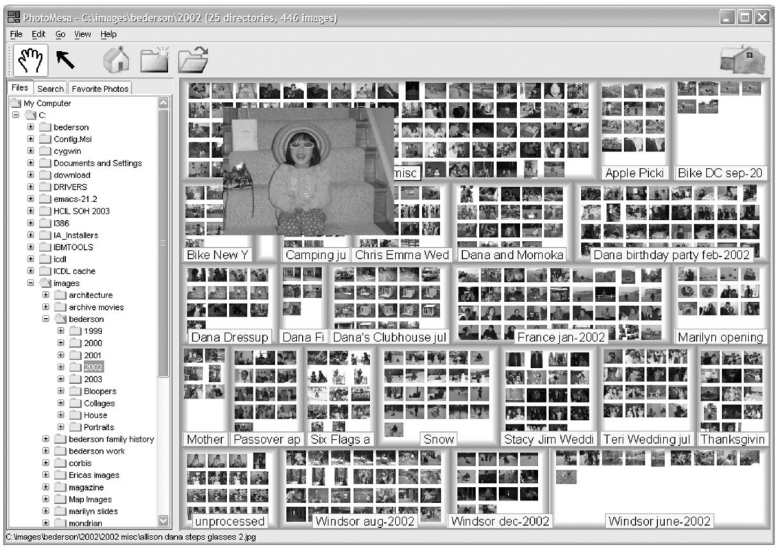


Fig. 4. Screen snapshot of PhotoMesa, written using Jazz. It uses a Zoomable User Interface to give users the ability to see many images at once, grouped by directory. PhotoMesa is available at http://www.cs.umd.edu/hcil/photomesa.

patterns, future developers will be able to better understand the trade offs of each approach.

4 The Jazz Polylithic Toolkit

We built two toolkits that explore the design space of polylithic and monolithic systems. We describe them each here to serve as a basis for understanding the trade offs in design approaches.

Jazz is a general-purpose polylithic toolkit for creating structured graphics with explicit support for Zoomable User Interface (ZUI) applications. Jazz is built entirely in Java and uses the Java2D renderer. Fig. 4 shows a screen snapshot of PhotoMesa [9], a zoomable photo browser application we built using Jazz.

Jazz follows a polylithic design, offering functionality by composing a number of simple objects within a "scene graph" hierarchy. These objects are frequently nonvisual (e.g., layout nodes), or serve to "decorate" nodes beneath them in the hierarchy 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.

Fig. 5 shows a complete standalone Jazz program that displays "Hello World!". 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. Jazz automatically updates the portion of the screen that has been changed, so no manual repaint calls are needed.

The polylithic design of Jazz leads to decoupled features that do not depend on each other; so, applications only pay for features when they use them. For instance, since not all nodes will be repositioned or resized, the base node type does not contain a transform. Instead, a transform node is created when needed and inserted above any node that should be transformed. Jazz includes similar compositional nodes to support layers, selection, transparency, hyperlinks, fading, spatial indexing, layout, and constraints.

```
import edu.umd.cs.jazz.*;
import edu.umd.cs.jazz.util.*;
import edu.umd.cs.jazz.component.*;

public class ZHelloWorld extends ZFrame {
    public void initialize() {
        ZText text = new ZText("Hello World!");
        ZVisualLeaf leaf = new ZVisualLeaf(text);
        getCanvas().getLayer().addChild(leaf);
    }

    public static void main(String args[]) {
        new ZHelloWorld();
    }
}
```

Fig. 5. Complete Jazz "Hello World!" program that supports panning and zooming. Alternatively, one can create a "`ZCanvas`" and place that anywhere a Swing `JComponent` can go.

4.1 The Jazz Architecture

A Jazz scene graph contains three basic kinds of objects: nodes, visual components, and cameras. Fig. 6 shows the object hierarchy of Jazz's core objects. Fig. 7 shows the runtime object structure of a typical application with several objects and a camera.

4.1.1 Nodes and Visual Components

The Jazz scene graph consists of a hierarchy of *nodes* that represent relationships between objects. Hierarchies of nodes are used to implement "groups" and "layers" that are found in most drawing programs and to facilitate moving a collection of objects together. A Jazz node has 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, can 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 characteristics that modify all of that node's descendants. For example, a transform node's affine transforms modifies the transform used for all child nodes. Similarly, a transparency node defines the transparency for groups of child objects.

Visual components are purely visual. They do not have a hierarchical structure and do not 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

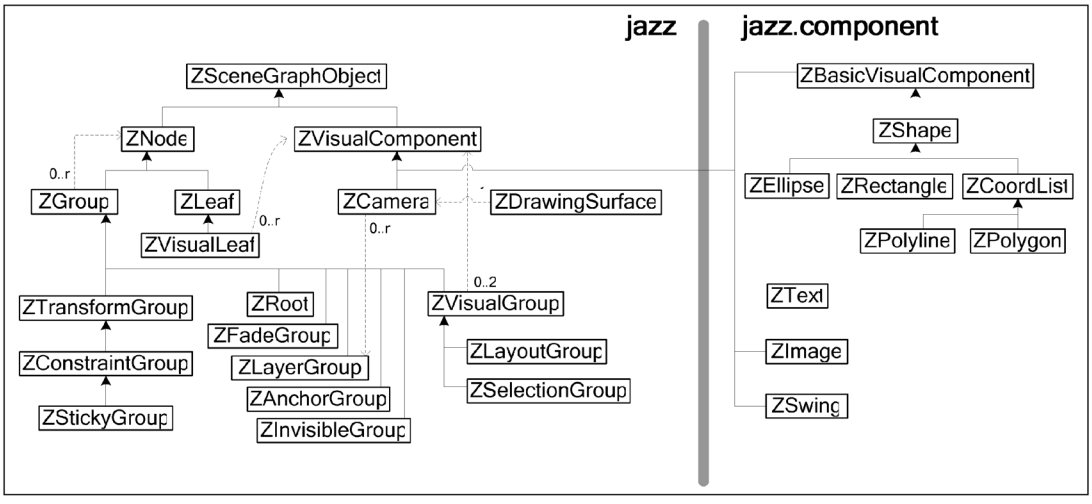


Fig. 6. Partial object hierarchy of Jazz shows the core objects used to construct visual scene graphs.

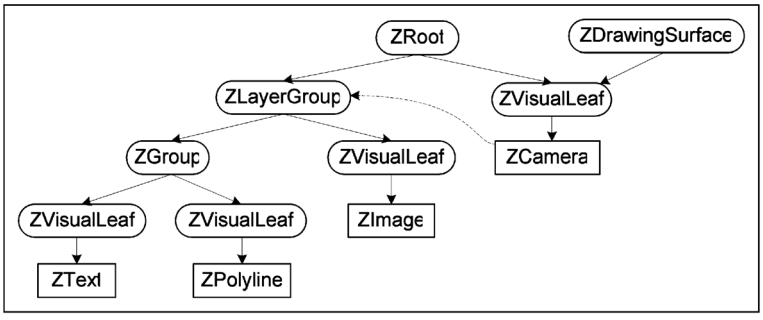


Fig. 7. Runtime object structure in a typical Jazz application. This scene contains a single camera looking onto a layer that contains an image and a group consisting of some text and a polyline. Nodes are depicted with ovals and visual components are in rectangles.

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 with squares in a subtree of the scene graph without affecting the grouping or position of objects.

4.1.2 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. Cameras can be mapped to a Swing widget so Jazz interfaces can be embedded in any Swing application, wherever a Swing `JComponent` widget is expected. In addition to being mapped to drawing surfaces, cameras can be embedded in a Jazz scene graph, so that nested views of a surface can be embedded recursively in a scene. Cameras used in this way are called *internal cameras* and act like nested windows (in Pad and Pad++, we called these "portals" [13], [28].)

4.1.3 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 is visible in each camera.

4.1.4 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.

4.1.5 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.

4.1.6 Events

Jazz supports interaction through Java's standard event listener model. There are two categories of events—input events and change events. Input events result from user interaction with a graphical object, such as a mouse press. Change events result from a modification to the scene graph, such as a transformation change or a node insertion.

4.1.7 Node Management

A drawback of the polylithic 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. 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 added 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. 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();
```

4.2 Legacy Java Code

In Jazz, visual components can be defined to wrap legacy Java code that is written without awareness of Jazz. Those components can then be panned, 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 surface.

Similarly, any lightweight Java Swing component can be embedded into a Jazz scene graph by placing it in a special Jazz visual component in the scene graph. The Swing component can then be panned and zoomed like other Jazz components, or can appear in multiple views. This means that fully functioning existing Java Swing code with complete GUIs can be embedded into a Jazz surface, and mixed and matched with custom graphics within Jazz.

5 The Piccolo Monolithic Toolkit

Piccolo is a Java toolkit that we built from scratch based on our experience with Jazz. We have also recently ported Piccolo to C#. Piccolo supports essentially the same core feature set as Jazz (except for embedded Swing widgets), but its design is monolithic rather than polylithic. This design change came from our experience building applications with Jazz. We found that the polylithic approach in Jazz met our original design goals of being easy to understand, maintain, and extend. But, managing all of the node types was too big a burden for the *application programmer*.

Piccolo gives up on the idea of separating each feature into a different class, and instead puts all the core functionality into the base object class, `PNode`. In the interest of simplicity, Piccolo also eliminates the separation between "node" and "visual component" types. Instead, every node can have a visual characteristic. In practice, this nearly halves the number of objects since most nodes ended up having a visual representation in Jazz, requiring two objects.

The Piccolo `PNode` class is thus bigger than Jazz's `ZNode` class, having 140 public methods compared with Jazz's 64 public methods. Piccolo also supports hierarchies, transforms, layers, zooming, internal cameras, etc., as does Jazz. The "Hello World" program in Piccolo (Fig. 8) looks very similar to the Jazz version.

```
import edu.umd.cs.piccolo.nodes.*;
import edu.umd.cs.piccolox.*;

public class PHelloWorld extends PFrame {
    public void initialize() {
        PText text = new PText("Hello World!");
        getCanvas().getLayer().addChild(text);
    }

    public static void main(String args[]) {
        new PHelloWorld();
    }
}
```

Fig. 8. Piccolo "Hello World!" program that supports panning and zooming. Or, one can create a "`PCanvas`" and place that anywhere a Swing `JComponent` can go.

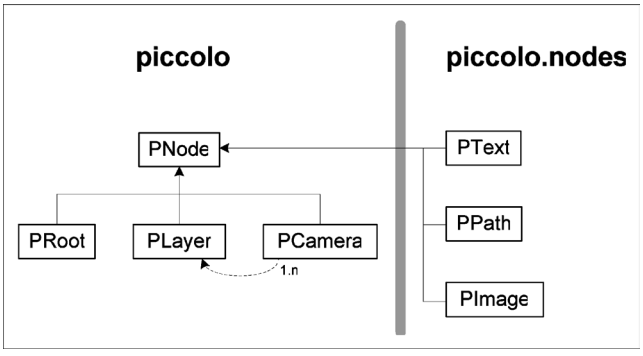


Fig. 9. Partial Piccolo object hierarchy showing the core classes needed to create a visual scene graph.

The Piccolo object hierarchy (Fig. 9) is also similar to Jazz, but again, is greatly simplified since many node types are merged into the core class. There are also fewer visual node types because they have been generalized. Fig. 10 shows a runtime scene graph using Piccolo.

As with Jazz, Piccolo caches bounds of objects and their children, and has a very careful implementation of the core scene graph traversal and modification mechanisms. It also supports region management which automatically redraws the portion of the screen that corresponds to objects that have changed.

6 Case Studies

We now try to develop a better understanding of polylithic and monolithic designs through a series of approaches. We first look at some sample applications implemented with the Jazz and Piccolo toolkits. We then analyze the performance of each toolkit. Finally, we end with a discussion of our practical experience using each toolkit and try to draw some conclusions from all of this.

It must be noted that there are many differences between Jazz and Piccolo independent of the polylithic and monolithic designs (i.e., the combination of multiple shape node types into a single "path" node). Furthermore, as mentioned at the outset of this paper, neither toolkit represents a pure design of its type. So, we present the case studies briefly and with caution in the hope that this will aid our overall understanding, but without making the claim that they in themselves prove any point.

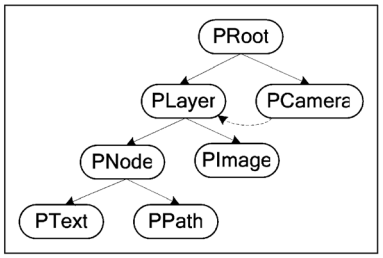


Fig. 10. Runtime object structure in a typical Piccolo application. This is the same scene that is represented by the Jazz scene graph of Fig. 7.

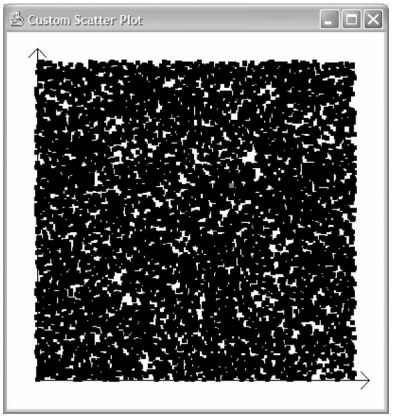


Fig. 11. Screen shot of scatter plot example.

6.1 Scatter Plot

The first case is a scatter plot that displays two-dimensional numerical data along with axes and labels (Fig. 11). The scatter plot also shows a tool tip with more detailed information about the point that the mouse pointer is over.

6.1.1 Piccolo

As is often the case with applications with structured graphics, there are two basic designs based on the granularity of the objects we create.

The simpler way to implement the scatter plot with Piccolo is to use very fine granularity and create one scene graph node for each point in the data set. This approach takes full advantage of the toolkit's capabilities. For example, the event handler is trivial since Piccolo finds the item under the mouse pointer and directs the event to the point the user is interacting with. Piccolo also performs region management so that only the portion of the screen that needs to be repainted actually does get repainted. The downside here is the overhead of the scene graph, in both memory and speed. The actual performance measurements are summarized at the end of this section.

A second implementation with coarse granularity uses a custom Piccolo node that represents all the points of the scatter plot. This node renders and picks each point in the scatter plot and additional nodes are used for axes and labels. This approach loses Piccolo's support for picking and region management per dot.

6.1.2 Jazz

Jazz could be used with either a fine or coarse granularity design, analogously to Piccolo. But, since the issues for these trade offs are similar to Piccolo, we chose just the fine granularity design.

TABLE 1
Performance Results of the
Four Implementations of the Scatter Plot

	Fine Granularity Piccolo	Coarse Granularity Piccolo	Fine Granularity Jazz
Scene render time	134.5 msec	132.8 msec	133.0 msec
Interaction time	14 msec	313 msec	14 msec
Lines of Code	139 lines	180 lines	158 lines
Class file size	8.0 kbytes	9.4 kbytes	9.2 kbytes
Memory usage our code (full application)	3,170k (3,606k)	1,685k (2,120k)	3,406k (3,842k)

6.1.3 Analysis

For each case, we measured the time it took to render the whole scene as well as the time it took to mouse over and select one dot (Table 1). We also measured code length by counting the number of lines of code as well as the size of the compiled class files. For lines of code, we excluded blank lines and comments, and included everything else. While the number of lines of code is not always a strong indicator of code complexity, we believe it is useful here since we wrote all examples using consistent formatting and style. In addition, we included the compiled class size which may be a clearer metric.

In this and all memory reports, we rely on the Java memory API. While this API is not promised to be completely accurate, we did several things to minimize problems. Before calling any memory APIs, we garbage collected, ran all finalization methods, and went to sleep for 100 milliseconds, and did that five times sequentially. In addition, before running any of our tests, we performed garbage collection and went to sleep for 100 milliseconds to minimize the chance that garbage collection would be performed during our tests. Finally, we ran all tests three times in succession, and only recorded the results of the third trial. This is intended to give the VM time to perform optimizations, just-in-time compilation, etc. We performed all tests using the standard 64 MB Java memory allocation.

Finally, we looked at the amount of memory used to run the application.[3] We tested the scatter plot code with 10,000 dots. This and all measurements reported in this paper was run on a 2.4 GHz Pentium 4 computer running Windows 2000 and Java 1.4.1 with a NVIDIA GeForce4 Ti 4600 graphics card.

These results show that there is not a major difference between Jazz and Piccolo (they are within 15 percent of each other in every metric). Rather, the major difference can be found in fine versus coarse granularity. Using a toolkit node per dot results in a performance improvement of nearly 20 times for interaction speed (because the toolkit's region management code means that only the dots that changed were redrawn). The speedup comes at the cost of memory—approximately 1.5 megabytes which works out to ~150 bytes per node, which is in fact about the size of a Piccolo dot node.

Since the application is so simple, there is not a major conceptual difference in the polylithic and monolithic approaches; although, the polylithic implementation takes a little longer to manage the extra nodes. In both cases (using fine granularity), there is not much more to the scene graph than a list of dot nodes. However, this example does show Jazz and Piccolo have similar performance and, thus, gives future comparisons more meaning.

3. This and all examples are included as supplementary material in the digital library.

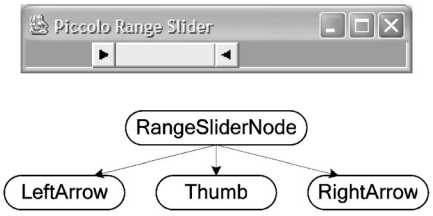


Fig. 12. Screen shot and scene graph for Piccolo implementation of the range slider.

6.2 Range Slider

The second case is a simple range slider widget. A range slider is similar to a traditional slider, but instead of using it to specify one value, users control two values. Thus, the model contains four values: minimum, maximum, lowValue, and highValue. Users control the two values (lowValue and highValue) by moving special areas on the left and right sides of the "thumb." Only simple features of the widget are implemented (i.e., no keyboard control, focus management, etc.) However, functionality is identical in all three examples.

6.2.1 Piccolo

The Piccolo implementation of the range slider follows the model-view-controller architecture [20]. A scene graph of objects is created to represent the subcomponents of the range slider, and Piccolo's layout mechanism is used to position them (Fig. 12).

Each component is defined by subclassing the core Piccolo node type and overriding its *paint* method. The base node `RangeSliderNode` also overrides `layoutChildren()` which is called whenever Piccolo determines that the layout needs to be updated, either because the widget size or model has changed. Piccolo also takes care of managing what and when to paint. The nodes are laid out in a local coordinate system that matches the model values which makes calculation straight forward. The node is then scaled to the requested size.

Finally, the controller is defined by an event handler that updates the model, but since Piccolo transforms the mouse coordinates into the node's local coordinate system (which was designed to match the model), the application does not have to convert between screen and model coordinates.

6.2.2 Jazz

The Jazz implementation is very similar to the Piccolo implementation. The main differences are in the node structure and the layout. Jazz's polylithic design requires several extra nodes (Fig. 13). The Jazz layout mechanism is part of the polylithic design, and a `ZLayoutGroup` node encapsulates the layout algorithm that is applied to the `ZVisualGroup`'s children.

6.2.3 Analysis

It turns out that while rendering is almost always the bottleneck in graphical applications, that is not true in this case. The widget is rendered with small simple rectangles

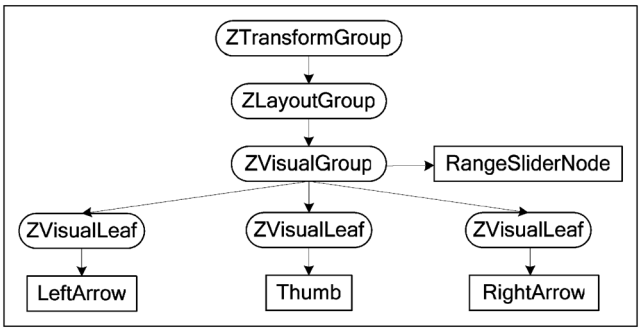


Fig. 13. Scene graph for Jazz implementation of the range slider.

TABLE 2
Range Slider Measurements

Task	Piccolo	Jazz
Render widget	0.109 msec	0.109 msec
Lines of code	254 lines	297 line
Total class file size	11.9 kbytes	16.2 kbytes
Memory usage our code (full application)	4.96k (475k)	5.16k (478k)

Lines of code for the range slider widget implementations.

which with our hardware was rendered by the graphics chip and, thus, was extremely fast. Table 2 shows the speed of rendering the entire range slider widget (measured by taking the average of rendering it 1,000 times). The code length and memory usage was similar in each case.

In terms of code design, there is a clear difference in the number of objects that have to be created and managed (Figs. 12 and 13). There is also an important difference in the structure of the code which can affect maintainability.

In the case of Piccolo changing the look of the arrow would require modifying the paint method of the arrow node, which means writing code.

In Jazz, on the other hand, the nodes that form the arrow are just a node within the scene graph. There is no LeftArrow type. Instead, there are just instances of scene graph nodes that form a tree representing the left arrow. So, to change the look, you would replace just the visual component representing the arrow, possibly via a visual authoring tool, without writing any code.

Similarly, to change the layout of the RangeSlider, in Piccolo you must modify the `layoutChildren` method, which means writing code and adding properties to the range slider. But, what happens if you then want a third layout (e.g., diagonal)? In Jazz, layout managers are objects which are disassociated from the things they lay out. You can replace the layout manager with any other substitute layout manager, for example, a vertical layout. Again, this could potentially be done with a visual authoring tool without any coding while the monolithic version requires coding.

A key distinguishing factor between monolithic and polylithic approaches is that monolithic toolkits favor coders who want to create subclasses and add methods. Polylithic toolkits favors designers who want to manipulate graphs of generic types rather than write code.

6.3 DateLens

The last example we looked at was an animated fisheye distortion calendar visualization. We picked this because it is a complex animated graphic display which represents an actual application we wrote called DateLens [10]. We felt this was a particularly challenging task for a toolkit-based solution because we wrote DateLens ourselves using a custom approach because we were fearful of the overhead that the toolkits would add.

We abstracted the core visualization and interaction component of DateLens and implemented it with polylithic and monolithic approaches. In this case, we also implemented a third "custom" version without either toolkit in the hope of developing an understanding of how these toolkits might influence code design. The result is a simple application with a grid of dates. Clicking on one date enlarges that date while shrinking the others using animation for the transition. Clicking on any other date animates a focus change to the clicked on date (Fig. 14).

As with the other two examples, the trade offs of polylithic versus monolithic approaches were similar. In addition, we saw that the toolkits added noticeable overhead in comparison to the custom approach since there are many small nodes. We analyzed the three solutions as with the first two examples, and the results are summarized in Table 3.

We did in fact observe an interesting phenomena with the custom solution. It's rendering speed was significantly faster than the toolkit solutions, but this was not because of the overhead of the scene graph traversal. Rather, it was because we used a faster rendering technique in the custom solution (drawing a single background and horizontal and vertical lines on top of it). The toolkits encouraged a rendering technique with localized rendering for each object, so we drew one rectangle per date which was slower. This points out a subtle cost of toolkits which is that their structure sometimes encourages nonoptimal designs.

7 Performance Studies

Toolkits have two major performance costs: rendering and scene graph maintenance. So, this section looks at the speed of these two tasks, comparing the Jazz and Piccolo toolkits to each other for both tasks and to custom rendering for the first task.

Since the structure of the scene graph can affect performance, we performed rendering tests with four different structures with varying breadth and depth. We performed all tests using the Java2D renderer to paint 10,000 100 x 100 pixel rectangles. Times are reported as the average over 10 measurements. The results for the tests described here are summarized in Table 4.

These results show that the toolkits incurred an average 4 percent performance penalty for scene graph traversal. Obviously, this percentage depends on the complexity of the objects being rendered. But, since many application graphics are more complicated than a rectangle, we could expect to see the relative cost of the scene graph traversal to decrease for many real applications. We also see that the penalty for traversing deeper scene graphs where many parent child traversals must be made is modest.

7.1 Scene Graph Manipulation Performance

Adding, removing, and modifying scene graph nodes can take a significant amount of time because the toolkits cache various properties such as hierarchical bounds. Jazz caches

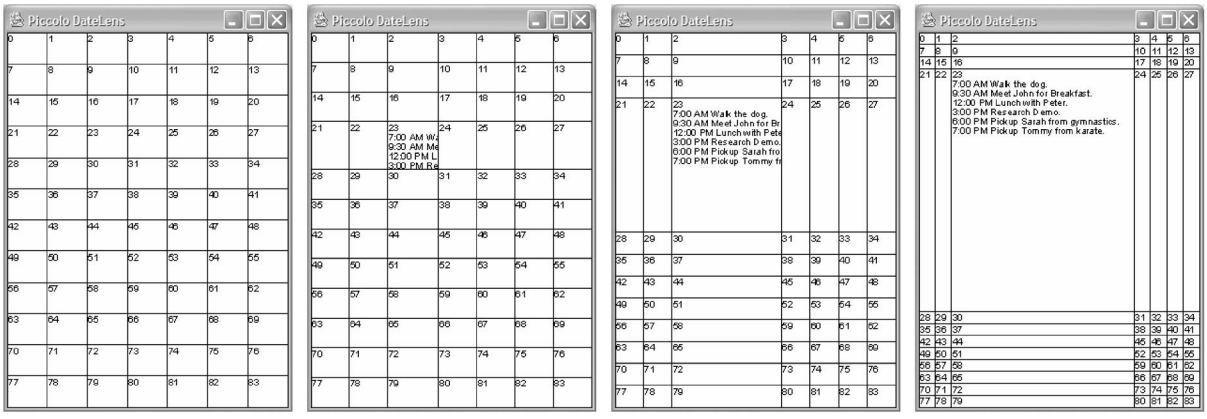


Fig. 14. Screen shots from the calendar example during an animated transition.

somewhat more than Piccolo, including both the local and global bounds of each node. Since this can speed up interaction performance, we thought at the time that this was the right design, but it turns out that the cost of maintaining both of these caches makes significant modification of the scene graph quite expensive. Piccolo caches just the local bounds of each node (i.e., the size of the node and its children, maintained in the parent's coordinate system.) This is much less expensive to maintain and compute, while still offering performance benefits.

We ran tests to analyze how long it takes to build, translate, and add 10,000 nodes with varying hierarchical structures (not counting the time spent to instantiate the nodes) to a scene graph for both toolkits (Table 5). The "Build" times are the time to add and create the nodes.

This table shows the overhead of both toolkits compared to a custom application where there is no scene graph and, thus, no cost for modifying the visual representation of the data (since there is none). The most important result here is that the overhead for animating a significant number of objects within Piccolo is acceptable. If our performance goal is 30 frames per second (i.e., 33 msec per frame) and only 0.4 msec is spent on scene graph manipulation, then only about 1 percent of the total time per frame is spent on Piccolo scene graph manipulation.

8 Our Experience

The last way to understand the differences between polylithic and monolithic approaches is through personal experience. Our group has built a number of applications with both Jazz and Piccolo of moderate complexity (approximately 10,000 - 50,000 lines of code each). Our applications are all end-user visualization-based systems. In addition, others have written a number of applications using both toolkits (which are described on the toolkit Webpages.[4]) The more significant applications that we wrote are:

- PhotoMesa—A zoomable photo browser (Jazz and rewritten for Piccolo.NET).
- ICDL—The International Children's Digital Library (Jazz).
- KidPad—A children's storytelling program (Jazz).
- SpaceTree—A Piccolo-based generic tree browser (Piccolo).
- TaxonTree—A biological taxonomy tree browser (Piccolo).

Between teaching new users about Jazz and building these applications on top of Jazz, we found that managing the polylithic node structure was a significant programming burden. Even after we added the "editor" concept to help hide the details of the node structure, we found that the multiple nodes remained a problem. The primary reason is that programmers had to keep an understanding of the scene graph in their heads. Bugs often occurred when someone did not point to the right node. For instance, deleting a node would mean finding the top-most node associated with that node and deleting a chain of nodes down to the bottom-most node associated with that chain. If the right nodes were not deleted, hard-to-debug problems would occur, such as when an extra node was left in the scene graph.

When we switched to Piccolo, we found that not only did we stop having the type of problems associated with multiple nodes per conceptual object—but it was also much easier to learn. Even though the overall functionality is very similar to Jazz, it is conceptually much simpler—and, this is largely due to the monolithic approach.

Furthermore, our experience with performance and size of code is consistent with the case studies described previously. While the Jazz code was often harder to write and maintain, it's performance was similar to Piccolo, and the actual amount of code was not significantly different.

TABLE 3
Performance Results of the Three Implementations of the Graphic Calendar

	Custom	Piccolo	Jazz
Scene render time	1.5 msec	2.1 msec	2.2 msec
Lines of Code	272 lines	171 lines	219 lines
Class file size	10.4 kbytes	10.4 kbytes	13.2 kbytes
Memory usage our code (full application)	7.5 k (484k)	8.0 k (516k)	10.2 k (535k)

4. All referenced applications are available for download from http://www.cs.umd.edu/hcil.

TABLE 4
Rendering Speed for a Tight Custom Loop, Piccolo, and Jazz for 10,000 Rectangles with Four Different Scene Graph Structures

Task	Custom	Piccolo	Scene graph Overhead	Jazz	Scene graph Overhead
10,000 rectangles	265.0 msec	270.3 msec	2 %	282.8 msec	7 %
1,000 groups of 10 rectangles		273.4 msec	3 %	281.2 msec	6 %
100 groups of 10 groups of 10 rectangles		267.2 msec	1 %	281.3 msec	6 %
10 groups of 10 groups of 10 groups of 10 rectangles		270.4 msec	2 %	278.1 msec	5 %

9 Conclusion

We have found that composition (polylithic) and inheritance (monolithic) based approaches each have their merits. The difficulty for the toolkit designer is deciding how far down the composition path to go.

In our experience, there is little to indicate that composition yields significantly different performance characteristics than inheritance. In addition, code sizes for the two approaches are similar. Deciding between composition and inheritance is, therefore, mainly a matter of identifying who the users of the toolkit are, and what the expected lifecycle of the toolkit will be.

Toolkit designers use composition because it defers implementation decisions until runtime, using nodes in a runtime graph to represent application functionality. Doing this offers the toolkit implementers and component writers greater freedom to design, modify, extend, and maintain the toolkit, e.g., by substituting one node type for another in the runtime graph. However, the penalty is that the toolkit objects becomes more abstract, and the application writer must learn to work with a greater number of classes and with more relationships between objects. For certain classes of users, this may make the toolkit too hard to use. We have observed that significantly greater learning time is required to learn to use compositional toolkits. The resulting code quality may suffer, or users may be unable to discover how to implement a working solution at all.

As an alternative, toolkit designers may use inheritance and hardwire design decisions into the class hierarchy. This yields a toolkit which, in our experience, is far easier to learn and to program against. The resulting code is also slightly more compact and easier to read. But, the penalty is that the policies adopted by the toolkit are harder to modify or repurpose.

When the toolkit designer can anticipate the needs of the application writer or if the application writer is anticipated to be a student or a nonprofessional programmer, we believe that inheritance is a more appropriate strategy. In our case, an inheritance-style design is appropriate for Piccolo since it is designed primarily with educational and research users in mind.

Also, since it is a fourth generation implementation, we have been able to create an inheritance-based class hierarchy that we believe represents a "sweet-spot" of the functionality required for our application scenarios.

As requirements for software interfaces change and grow (e.g., with the addition of new features, such as accessibility aids or automation features), static class hierarchy decisions may prove too limiting and hard to work within. For platforms that are designed to survive extended evolution, the extra complexity added by composition is probably worthwhile. In particular, larger engineering teams may benefit from the loose coupling offered by compositional designs.

One interesting area of future research is hybrid systems which offer the characteristics of both inheritance and compositional approaches. This might be achieved presenting multiple compositional nodes as a single "super-node," for example, as we attempted to do with the Jazz "editors." Another approach is to follow the inheritance design pattern, but delegate pieces of functionality to helper objects that can be plugged in. We do not yet have clear guidelines for these hybrid approaches.

TABLE 5
Scene Graph Manipulation Times for Piccolo and Jazz

Task	Piccolo	Jazz
10,000 rectangles		
Build 10,000 nodes	16.0 msec	219.0 msec
Translate 10,000 nodes	0.4 msec	23.5 msec
Remove 10,000 nodes	5.3 msec	5.3 msec
1,000 x 10 rects		
Build 10,000 nodes	16.0 msec	218.0 msec
Translate 10,000 nodes	0.4 msec	50.8 msec
Remove 10,000 nodes	5.3 msec	5.3 msec
100 x 10 x 10 rects		
Build 10,000 nodes	15.0 msec	226.5 msec
Translate 10,000 nodes	0.4 msec	62.5 msec
Remove 10,000 nodes	5.0 msec	10.6 msec
10 x 10 x 10 x 10 rects		
Build 10,000 nodes	16.0 msec	226.5 msec
Translate 10,000 nodes	0.4 msec	82.3 msec
Remove 10,000 nodes	5.0 msec	10.3 msec

The notation "n x m rects" means n groups of m rectangles.

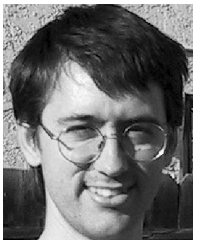
ACKNOWLEDGMENTS

The authors enjoyed their collaborations with those involved with Pad++, especially Jim Hollan, Jason Stewart, Allison Druin, Britt McAlister, George Furnas, and Ken Perlin. They would like to thank their fellow members of the HCIL, especially Jim Mokwa and Maria Jump for their early contributions to Jazz. Most importantly, the many users of Jazz and Piccolo have helped them design, debug, and understand the requirements of both toolkits, and have made them much more broadly useful than would have been possible otherwise. This work was funded by DARPA's Command Post of the Future and Semantic Web projects.

REFERENCES

[1] 3D Studio Max, http://www.3dmax.com, 2003.
[2] Adobe SVG Viewer, http://www.adobe.com/svg/, 2003.
[3] ECMA Script, http://www.ecma-international.org, 2003.
[4] ILog, www.ilog.com, 2003.
[5] Java3D, http://java.sun.com/products/java-media/3D/ 2003.
[6] Macromedia Flash, http://www.macromedia.com/software/flash/, 2003.
[7] SGI OpenInventor, http://www.sgi.com/software/inventor/, 2003.
[8] SoftImage, http://www.softimage.com, 2003.
[9] B.B. Bederson, "PhotoMesa: A Zoomable Image Browser Using Quantum Treemaps and Bubblemaps," *Proc. ACM Symp. User Interface Software and Technology, CHI Letters,* vol. 3, no. 2, pp. 71-80, 2001.
[10] B.B. Bederson, A. Clamage, M.P. Czerwinski, and G.G. Robertson, "DateLens: A Fisheye Calendar Interface for PDAs," *ACM Trans. Computer-Human Interaction,* vol. 11, no. 1, pp. 90-119, 2004.
[11] B.B. Bederson and J.D. Hollan, "Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics," *Proc. User Interface and Software Technology (UIST 94),* pp. 17-26, 1994.
[12] B.B. Bederson, J.D. Hollan, K. Perlin, J. Meyer, D. Bacon, and G.W. Furnas, "Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics," *J. Visual Languages and Computing,* vol. 7, pp. 3-31, 1996.
[13] B.B. Bederson and J. Meyer, "Implementing a Zooming User Interface: Experience Building Pad++," *Software: Practice and Experience,* vol. 28, no. 10, pp. 1101-1135, 1998.
[14] B.B. Bederson, J. Meyer, and L. Good, "Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java," *Proc. ACM Symp. User Interface Software and Technology, CHI Letters,* vol. 2, no. 2, pp. 171-180, 2000.
[15] B.B. Bederson and J.D. Hollan, "Pad++: A Zooming Graphical Interface System," *Proc. Conf. Human Factors in Computing Systems (CHI 95),* 1995.
[16] D. Fox, "Tabula Rasa: A Multi-Scale User Interface System," doctoral dissertation, New York Univ., New York 1998.
[17] G.W. Furnas, "Generalized Fisheye Views," *Proc. Conf. Human Factors in Computing Systems (CHI 86),* pp. 16-23, 1986.
[18] S.E. Hudson and J.T. Stasko, "Animation Support in a User Interface Toolkit," *Proc. Conf. User Interface and Software Technology (UIST 93),* pp. 57-67, 1993.
[19] J.K. Ousterhout, *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.
[20] B.E. Krasner and S.T. Pope, "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System," *J. Object-Oriented Programming,* vol. 1, no. 3, pp. 26-49, 1988.
[21] M.A. Linton, J.M. Vlissides, and P.R. Calder, "Composing User Interfaces With InterViews," *IEEE Software,* vol. 22, no. 2, pp. 8-22, 1989.
[22] B.A. Myers, R.G. McDaniel, R.C. Miller, A.S. Ferrency, A. Faulring, B.D. Kyle, A. Mickish, A. Klimovitski, and P. Doane, "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Trans. Software Eng.,* vol. 23, no. 6, pp. 347-365, 1997.
[23] K. Perlin and D. Fox, "Pad: An Alternative Approach to the Computer Interface," *Proc. Computer Graphics Conf. (SIGGRAPH),* pp. 57-64, 1993.
[24] K. Perlin and J. Meyer, "Nested User Interface Components," *Proc. ACM Symp. User Interface Software and Technology, CHI Letters,* vol. 1, no. 1, pp. 11-18, 1999.
[25] S. Pook, E. Lecolinet, G. Vaysseix, and E. Barillot, "Context and Interaction in Zoomable User Interfaces," *Proc. Conf. Advanced Visual Interfaces (AVI 2000),* pp. 227-231, 2000.
[26] J. Raskin, *The Humane Interface.* Addison Wesley, 2000.
[27] R.B. Smith, J. Maloney, and D. Ungar, "The Self-4. 0 User Interface: Manifesting a System-Wide Vision of Concreteness, Uniformity, and Flexibility," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 95),* pp. 47-60, 1995.
[28] M.C. Stone, K. Fishkin, and E.A. Bier, "The Movable Filter as a User Interface Tool," *Proc. Conf. Human Factors in Computing Systems (CHI 94),* pp. 306-312, 1994.
[29] S.H. Tang and M.A. Linton, "Blending Structured Graphics and Layout," *Proc. Conf. User Interface and Software Technology (UIST 94),* pp. 167-174, 1994.
[30] D. Ungar and R.B. Smith, "Self: The Power of Simplicity," *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 87),* pp. 227-241, 1987.

Benjamin B. Bederson received the BS degree from Rensselaer Polytechnic Institute in 1986, the MS degree in 1989, and the PhD degree in 1992 at New York University in the Courant Institute of Mathematical Sciences in Computer Science. He is an assistant professor of computer science and director of the Human-Computer Interaction Lab at the Institute for Advanced Computer Studies at the University of Maryland, College Park. His work is on information visualization, interaction strategies, and digital libraries. He is also the president and CEO of Windsor Interfaces which commercializes software developed at the HCIL. From 1990-1992, he was a research scientist at Vision Applications, Inc. working on miniature robotics and computer vision. He worked as a research scientist at Bellcore in the Computer Graphics and Interactive Media research group, and as a visitor at the New York University Media Research Laboratory in 1993 and 1994. From 1994-1997, he was an assistant professor of computer science at the University of New Mexico.



Jesse Grosjean writes and sells Mac OS X software on the Web at www.hogbaysoftware.com. He also provides support for the University of Maryland's Piccolo open-source project. After graduating from Macalester College in 1999, he spent a year in a basement, developing zooming interfaces at Cognective, a Cambridge, Massachusetts-based startup. From 2000-2003, he focused on implementing zooming interfaces and toolkits, as a research associate at the University of Maryland's Human-Computer Interaction Lab.

Jon Meyer received the BA degree in artificial intelligence from Sussex University and the MS degree in computer science from New York University. He is a computer scientist, teacher, and artist. He has 15 years of experience in the software industry, as a software engineer and a researcher, specializing in computer graphics, animation, and user interfaces. He has worked as a program manager at Microsoft and as a research scientist at New York University's Media Research Laboratory.