Toolkit Design for Interactive Structured Graphics

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

ABSTRACT

In this paper, we analyze three approaches to building graphical applications with rich user interfaces. We compare hand-crafted custom code to polylithic and monolithic toolkit-based solutions. Polylithic toolkits follow a design philosophy similar to 3D scene graphs supported by toolkits including Java3D and OpenInventor. Monolithic toolkits 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, for most applications, a monolithic-based toolkit is more effective than either a hand-crafted or a polylithic solution for building interactive structured graphics, but that each has advantages in certain situations.

Keywords

Monolithic toolkits, Polylithic toolkits, Zoomable User Interfaces (ZUIs), Animation, Structured Graphics, Graphical User Interfaces (GUIs), Pad++, Jazz, Piccolo.

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 forms-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 large amounts 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 [18] 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 [7] and Adobe SVG Viewer [3]. 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.

In this paper, we consider two distinct design philosophies for toolkits to support creation of custom graphical components: monolithic and polylithic. We describe the key properties of monolithic and polylithic designs, and examine two toolkits that we built, Jazz[1], a polylithic toolkit, and Piccolo[2], a monolithic toolkit. Finally, we provide a qualitative and quantitative analysis to compare hand-crafted code with code written using these two toolkits, looking at application speed and size, memory usage, and programmability.

In this paper, 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

---

[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 Mozzilla 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 Mozzilla Public License, and is available at: http://www.cs.umd.edu/hcil/piccolo.

include such as accessibility, localization, keyboard navigation, etc. In addition, our analysis is about our two specific toolkits. While our experimental results are clearly tied to these specific toolkits, we believe that the main lessons we learned are generalizable to other monolithic and polylithic toolkits.

REQUIREMENTS FOR NEW UI COMPONENTS

When creating a new kind of UI component, the choice between using a toolkit or writing hand-crafted code must be made based upon the requirements of the particular component being built. A very simple new component, such as a range slider (where users can control two parameters instead of just one), may not warrant a toolkit-based solution. On the other hand, a more complex component such as an interactive graph probably does.

Let us start by defining our requirements for such a toolkit. In our research, we are particularly interested in new visualization techniques, such as Zoomable User Interfaces (ZUIs) [10, 11, 12, 13] and fisheye visualizations [9, 16]. We are also interested in animation and in dynamic data displays. For components that support our needs, a range of toolkit requirements arise:

1) The toolkit must be small and easy to learn and use with an existing GUI framework.

2) The toolkit must manage painting, picking and event dispatch, with enough flexibility for applications to customize these features.

3) It must be possible to write interaction handlers that provide for user manipulation of individual elements, and groups of objects.

4) The toolkit must provide support for graphics that are non-rectangular or transparent, scaled, translated and rotated.

5) Large numbers of objects must be supported so that rendering and interaction performance is maintained with complex scenes.

6) View navigations (pans and zooms) should be available, and should be animated.

7) Multiple views onto the surface should be supported, both via multiple windows, and via camera objects that are placed on the surface, used as "portals" or "lenses".

RELATED WORK

There are number of research [17, 21] and commercial [5, 18] 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 [20] for example, supports structured graphics and user interface components. Fresco [28] 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 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 [21] is a 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 for custom widgets is SubArctic [17]. 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 does not support multiple cameras or arbitrary 2D transformations (including scale) on objects and views.

Morphic [4, 26] 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, making it problematic for creating our zooming interfaces.

There were several prior implementations of Zoomable User Interfaces toolkits as well. These include the original Pad system [22], and more recently Pad++ [11, 12, 14], as well as other systems [15, 23, 24], and a few commercial ZUIs that are not widely accessible [1, 25; Chapter 6, 30]. 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 "Node" class. In this paper we compare and contrast these kinds of toolkits with Jazz, a new toolkit we have developed which follows a polylithic design, and with Piccolo, a lightweight monolithic toolkit.

MONOLITHIC VERSUS POLYLITHIC 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 [19]. 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 (Figure 1).

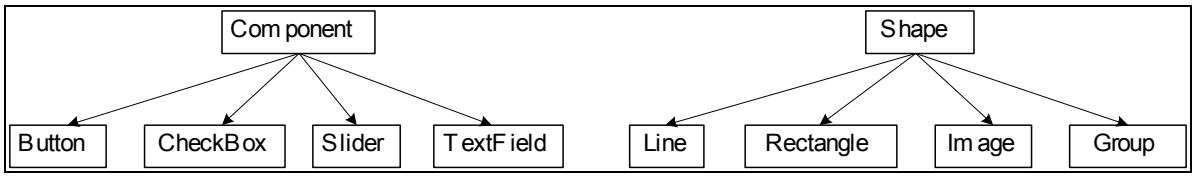

Figure 1: Class hierarchy of a GUI toolkit (left) and a structured-graphics toolkit (right).

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 buried within the class structure.

Three-dimensional graphics toolkits provide an important counterexample. Toolkits such as Java3D [6] and OpenInventor [2] 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 non-visual data such as behaviors, coordinate transforms, cameras, or lights (Figure 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.

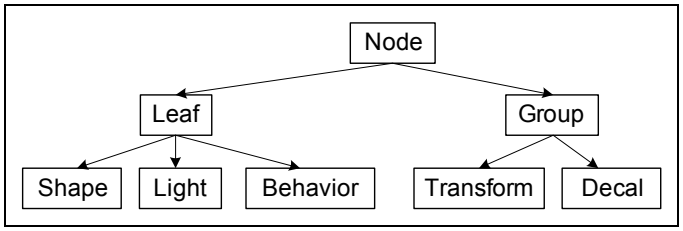

Figure 2: Class Hierarchy of a typical 3D graphics toolkit

We call the concrete design approach adopted by most 2D toolkits *monolithic*, because these toolkits have a few large classes containing all the core functionality likely to be used by applications. We call the 3D toolkit design approach *polylithic*, because it consists of many small classes, each representing an isolated bit of functionality where several are often linked together to represent one semantic unit.

Monolithic toolkits suffer from a common problem: the toolkit 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 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.

Polylithic designs on the other hand, can potentially offer both reusability and customizability, because they compose functionality through runtime instantiation rather than through sub-classing. This promise of better toolkit maintainability and extensibility led us to the polylithic design of Jazz.

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. For example, consider defining a new kind of Button object. In a monolithic GUI toolkit, you might use a class hierarchy as shown on the left in Table 1:

Table 1: Use of inheritance in monolithic and polylithic designs

// Monolithic approach	// Polylithic approach

`class Component {`
`  …`
`}`
	`class Node {`
	`  …`
	`}`

`class Transform {`
`  …`
`}`
	`class Transform extends Node {`
	`  …`
	`}`

`class Label extends Component {`
`  …`
`}`
	`class Label extends Node {`
	`  …`
	`}`

`class Button extends Label {`
`  …`
`}`
	`class ButtonBehavior extends Node {`
	`  …`
	`}`

The functionality is derived by statically extending the `Label` class and adding more methods. `Button` instances are created and added directly to the visual graph:

```
Button b = new Button();
b.setLabel("Click Me");
b.setTranslation(20, 20);
```

Now consider a polylithic design, as seen on the right in Table 1. First, `TransformNode`, `ButtonBehavior` and `LabelNode` are all defined as subclasses of `Node` – they are otherwise unrelated entities. To create a new button, the developer creates a transform, a button

behavior object and a label object independently, and adds them to the runtime scene graph explicitly to define the relationship between the button and its label, e.g.

```
ButtonBehavior button = new ButtonBehavior();
Transform   transform = new Transform();
Label           label = new Label();
transform.setTranslation(20, 20);
label.setLabel("Click Me");
button.add(transform);
transform.add(label);
```

In this example, the label is added as a child to the `ButtonBehavior` object, which is added as a child to the root transform object.

By adopting this approach to composing functionality, the same `ButtonBehavior` class could conceivably be reused for many different kinds of buttons (e.g. image-based buttons), not just for buttons with labels.

Of course, similar functionality can be achieved in both monolithic and polylithic toolkits. In polylithic toolkits, new functionality is created by composing instances, whereas in monolithic toolkits, new functionality is introduced through sub-classing. In this sense, polylithic designs are more similar to Prototype-based programming systems such as Self [29] or ECMAScript [4], which use runtime instancing to create derived types.

The example above immediately demonstrates the main drawback of polylithic systems: the application code is about twice as long as that for the monolithic system. Monolithic systems also tend to be more familiar to programmers used to languages like 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. The polylithic design approach yields more flexible class hierarchies.

This flexibility is likely to be useful when applications and objects are built dynamically at run-time. 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 in ways not imagined by the author of the original one. Thus, there is a trade-off between application code complexity and flexibility.

THE JAZZ POLYLITHIC TOOLKIT

Jazz is a general-purpose 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. Figure 3 shows a screen snapshot of PhotoMesa [8], a zoomable photo browser application we built using Jazz.

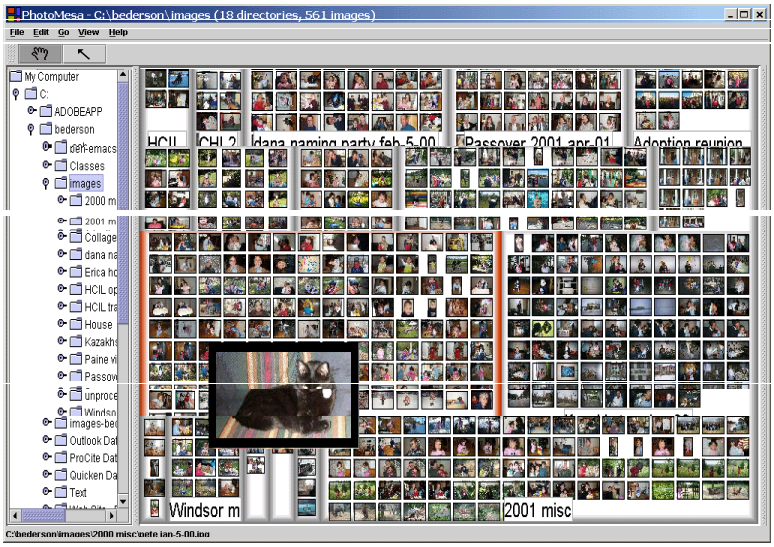

Figure 3: Screen snapshot of PhotoMesa, written using Jazz. It uses a Zoomable User Interface to give users the ability to see many images at once, grouped by directory. PhotoMesa is available at http://www.cs.umd.edu/hcil/photomesa.

Jazz is a polylithic toolkit, offering functionality 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 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.

Figure 4 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.

```
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();
    }
}
```

Figure 4: 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.

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 transformed, the core 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.

The Jazz Architecture

A Jazz scene graph contains three basic kinds of objects: nodes, visual components, and cameras. Figure 5 shows the object hierarchy of Jazz's core objects. Figure 6 shows the run-time object structure of a typical application with several objects and a camera.

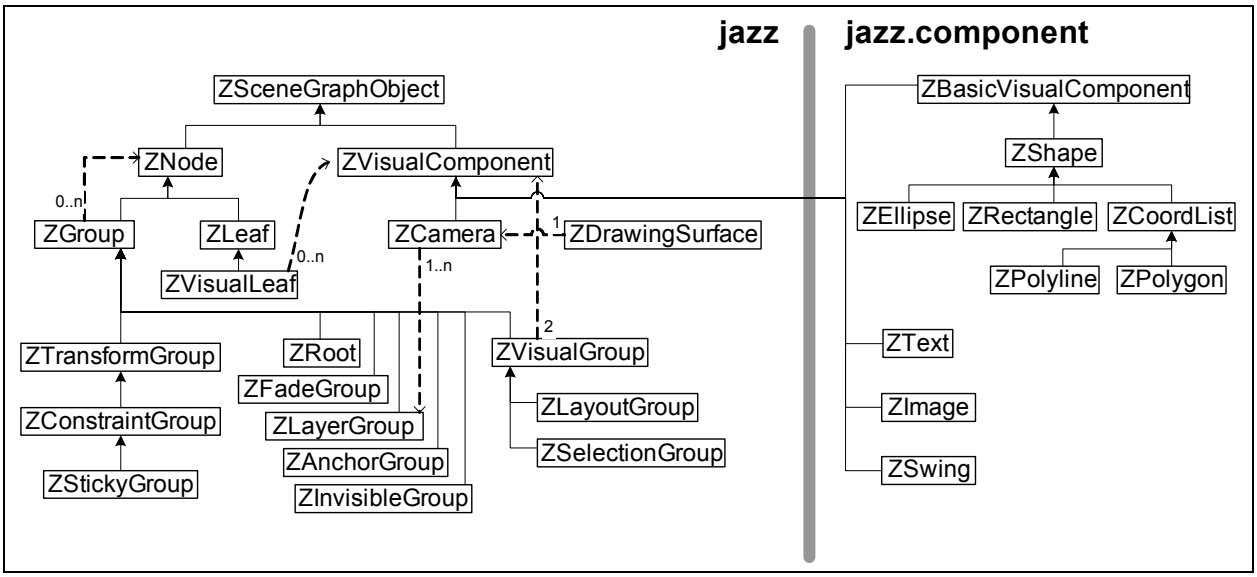

Figure 5: Partial object hierarchy of Jazz shows the core objects used to construct visual scene graphs.

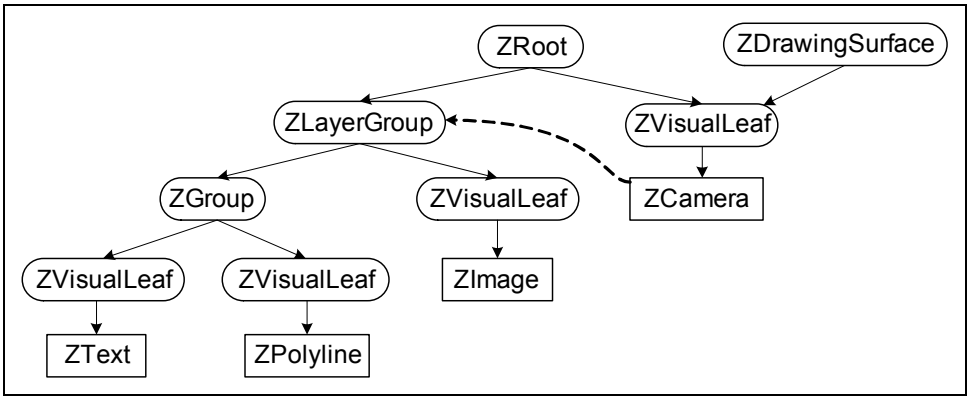

Figure 6: Run-time object structure in a typical Jazz application. This scene contains a single camera looking onto a layer that contains an image and a group consisting of some text and a polyline. Nodes are depicted with ovals and visual components are in rectangles.

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 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. 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 also be treated just like any other visual component – they 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" [27].)

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.

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 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.

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();
```

Legacy Java Code

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 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.

THE PICCOLO MONOLITHIC TOOLKIT

Piccolo is a Java toolkit based on Jazz. We are also currently porting Piccolo to C#. It supports essentially the same core feature set (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`. 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 uses a scene graph and supports hierarchies, transforms, layers, zooming, internal cameras, etc. as does Jazz. The "Hello World" program in Piccolo (Figure 7) 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();
    }
}
```

Figure 7: Piccolo "Hello World!" program that supports panning and zooming. Or, one can create a "`PCanvas`" and place that anywhere a Swing `JComponent` can go.

The Piccolo object hierarchy (Figure 8) 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. Figure 9 shows a run-time scene graph using Piccolo.

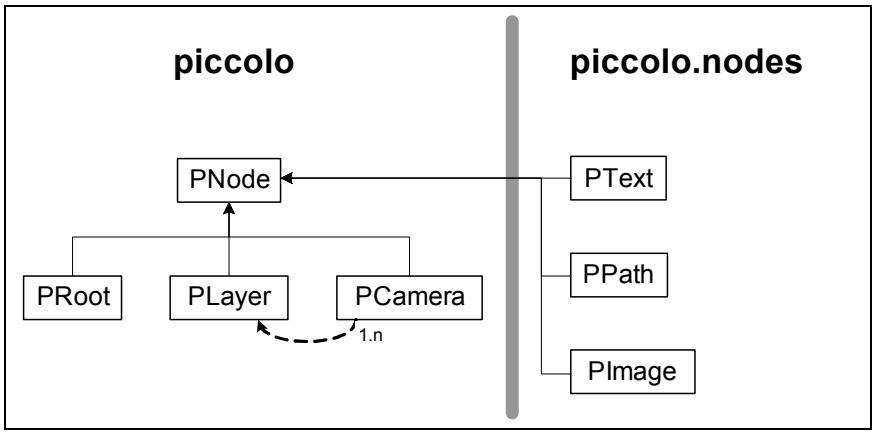

Figure 8: Partial Piccolo object hierarchy showing the core classes needed to create a visual scene graph.

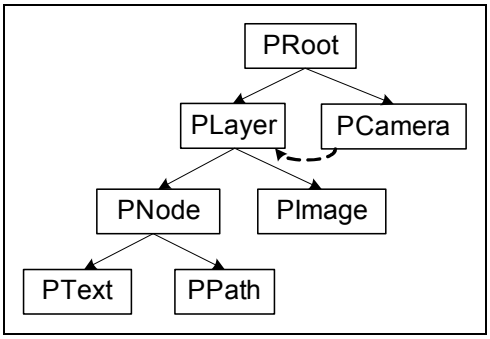

Figure 9: Run-time object structure in a typical Piccolo application. This is the same scene that is represented by the Jazz scene graph of Figure 6.

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.

CASE STUDIES

In this section, we look at a number of different problem areas that arise in typical interactive applications. For each area, we discuss how the problem is addressed in monolithic toolkits, in polylithic toolkits, and using custom code. With these comparisons, we hoped to understand the benefit as well as the cost of each toolkit approach. We tried to pick examples where the use of a

toolkit was not clearly advantageous since any application that takes advantage of a feature explicitly supported by the toolkit will clearly benefit. In the cases of Jazz and Piccolo, if an application needs zooming and multiple views, the toolkits will clearly make that easier since it is built in to the toolkit. So, rather than showing where our toolkits provide obvious benefit, we attempt to show the trade-offs of the different approaches for a broader set of realistic problems.

Scatter Plot

The first case is a scatter plot that displays two-dimensional numerical data along with axes and labels. The scatter plot also shows a tool tip with more detailed information about the point that the mouse pointer is over. We picked this example because it is typical of interactive graphics systems in that it can show a lot of data. Thus, the overhead of a toolkit seems unappealing.

Custom

The custom version of the scatter plot (Figure 10) includes a data model that has an array of items containing a point and descriptive information to be used by the tool tips. The renderer iterates through the points and paints them along with axes and labels. The event handler iterates over the points puts up a tool tip if the mouse pointer is over a point.

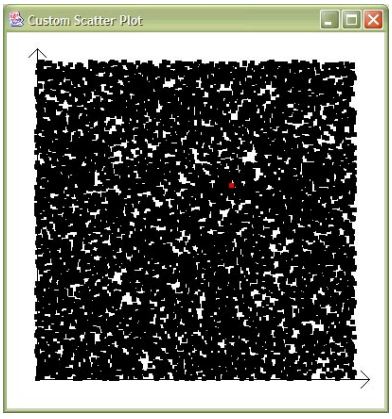

Figure 10: Screen shot of scatter plot example.

Piccolo

The simplicity of the custom scatter plot is hard to beat, but lets look at how a scatter plot would be implemented in the Piccolo monolithic toolkit. It turns out that there are two basic designs that are appropriate which offer different trade-offs.

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. Unlike the custom solution, this makes interaction with individual items quite fast. 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, comparing all the implementations.

A more conservative design with a 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 essentially maps the previous custom solution to a Piccolo node, and as such loses Piccolo's support for picking and region management. In fact, there isn't much difference between this version and the custom approach except that since the scatter plot is part of Piccolo, the user may zoom in and out, or have multiple views.

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 since that uses more memory and we are trying to understand the limits of the toolkits.

Analysis

We analyzed the four approaches to implementing the scatter plot (custom, fine granularity Piccolo, course granularity Piccolo, and fine granularity Jazz) to demonstrate the trade-offs available when creating this kind of interactive graphical application (Table 2).

For each case, we measured the time it took to render the whole scene as well as the time it took to render one dot. We also measured code length by counting the number of lines of code as well as the size of the compiled class files. 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.

Table 2: Performance results of the four implementations of the scatter plot

	Custom	**Fine Granularity Piccolo**	**Coarse Granularity Piccolo**	**Fine Granularity Jazz**
Scene render time	132.8 msec	134.5 msec	132.8 msec	133.0 msec
Dot render time	303 msec	14 msec	313 msec	14 msec
Lines of Code	242 lines	197 lines	258 lines	216 lines
Class file size	14.9 kbytes	8.0 kbytes	9.4 kbytes	9.2 kbytes
Memory usage our code (full application)	1,682k (2,232k)	3,170k (3,606k)	1,685k (2,120k)	3,406k (3,842k)

[3] This and all examples are available for download at http://www.cs.umd.edu/hcil/piccolo/paper-examples.

The results are interesting because they show that the toolkits do use extra memory as expected, but add very little degradation to the rendering time. Further, for the fine-granularity toolkit implementations, performance is dramatically improved (by more than a factor of 20). This is because the toolkit's region management meant that only the dots that changed were redrawn.

The toolkit implementations have the extra disadvantage that they require a good understanding of the toolkit itself in order to write and maintain the code. And of course, the toolkit itself takes memory. The Jazz toolkit is 460 Kbytes, and the Piccolo toolkit is 69 Kbytes (although the bigger size of Jazz is largely due to packaging choices and the size of the used classes is closer to Piccolo). So, in the final analysis, as expected, using either of these toolkits is probably overkill for a simple widget, but they do provide some real advantages. And, if the toolkit is already being used by the application for some other reasons, then the disadvantage of the size of the toolkit goes away.

Range Slider

The second case is a simple range slider widget. While our goal isn't explicitly to support custom widgets, they are representative of interactive graphics that have complex behavior. In particular, we don't address keyboard interactivity, internationalization, accessibility, data binding, themes, etc.

A range slider has several subcomponents that users can interact with. It requires layout, drawing, and movement of subcomponents, and it is relatively simple. In fact, it is so simple, the overhead of using any toolkit to build it may seem inappropriate. So let us look at the actual construction of this widget using a custom design, the monolithic Piccolo toolkit, and the

polylithic Jazz toolkit. The goal is to construct the same widget in all three cases and compare speed, size, memory usage and to examine the complexity of the designs.

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".

All versions of the range slider are implemented as Swing widgets so they can be placed in the Swing widget hierarchy wherever a `JComponent` is expected. Note that since the goal of these examples is to understand the basic design approaches, only simple features of the widget are implemented (i.e., no keyboard control, focus management, etc.) However, functionality is identical in all three examples.

Custom

We first built the range slider using completely custom code. The code follows the traditional model-view-controller approach [19], and is represented by three explicit parts of the code. The model maintains the four values of the model and a public API for getting and setting the model. It also generates a custom event whenever the model is changed, and manages a list of listeners for this event.

The view consists primarily of a paint method that renders the widget to the screen. The paint method calculates the size of the widget subcomponents (trough, thumb, and arrows) based on the model and widget dimensions, and then draws each subcomponent.

The controller consists of event handlers that respond to user interaction. The event handlers perform a "picking" operation that determines what subcomponent the mouse is over, and then performs the appropriate action.

Piccolo

The Piccolo implementation of the range slider also follows the model-view-controller architecture, but instead of implementing the view and controller pieces by calculating positions of subcomponents at render and pick time, 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.

When the range slider is first created, the scene graph of subcomponents is created (Figure 11). Each component is defined by sub-classing 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 easier. The node is then scaled to the requested size.

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

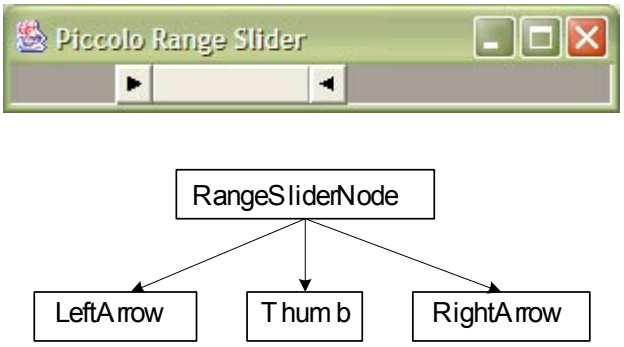

Figure 11: Screen shot and scene graph for Piccolo implementation of the range slider.

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 sub-nodes per conceptual subcomponent (Figure 12). 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.

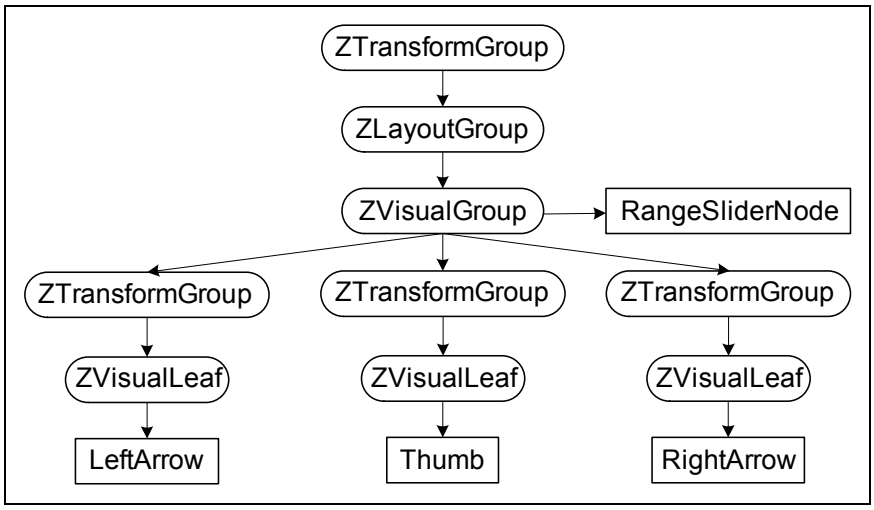

Figure 12: Scene graph for Jazz implementation of the range slider.

Analysis

Now let us compare the three implementations of the range slider in terms of speed, size, memory usage, and overall design. 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 which with our hardware was rendered by the graphics chip and thus took almost no time. Table 3 shows the speed of rendering the entire range slider widget (measured by taking the average of rendering it 1,000 times). The toolkit overhead appears because there is some time spent in setting up each render.

The code length was similar in all three cases, but was organized differently (Table 3). The custom version had more code devoted to rendering and event handling, and the toolkit versions had more code devoted to scene graph creation. The toolkits did use more memory, and this is a typical result of toolkit use. The toolkit implementations require the same amount of memory for the model, and then extra memory to represent the scene graph.

Table 3: Range slider measurements. Lines of code (LOC) for the range slider widget implementations. Total LOC includes miscellaneous code not counted in the categories.

Task	Custom	Piccolo	Jazz
Render widget	0.047 msec	0.109 msec	0.109 msec
Model LOC	99 lines	107 lines	107 lines
View LOC	50 lines	87 lines	110 lines
Controller LOC	110 lines	62 lines	72 lines
Total LOC	375 lines	382 lines	432 line
Total class file size	5.8 kbytes	11.9 kbytes	16.2 kbytes
Memory usage our code (full application)	0.04k (466k)	4.96k (475k)	5.16k (478k)

In this and all memory reports, we rely on the Java memory API which is not promised to be completely accurate. In this case, the custom code size appears to be underreported because we know that more than 40 bytes are used – but it does accurately show major trends.

Finally, we examine the design of the three solutions. In all three cases, the portion of the code that represents the model is essentially the same. The differences are in the view and controller.

The custom implementation has code to convert between screen and model coordinates, and while this code uses relatively straight forward linear interpolation, it is a bit tricky and often confuses beginning programmers. The toolkit implementations avoid this by defining the widget in the same coordinates as the model, and then use the transform to scale the widget to fill the desired space. However, even this solution has some subtleties because we wanted the arrow sub-component to be a fixed width, independent of widget size – and so when the widget was resized, the nodes representing the arrows had to be resized in a reciprocal manner. And of course, the toolkit implementations could have used the same coordinate system transformation as the custom implementation. So, in this case, the potential advantage of the toolkit was burdened by the mismatch between the static transformation that was offered by the toolkit and the more dynamic layout needed by the application.

A subtle difference is the degree of encapsulation of the subcomponents within the different implementations. The custom version has code for the subcomponents all mixed together. So, to change how the arrows look, for example, requires changing the paint method for the whole widget which requires an understanding of how the entire widget is painted, and could lead to bugs or unintentional changes outside of the thumb. The toolkit versions, having an object for each subcomponent, have a clear encapsulation of each one.

In the case of Piccolo, to change the look of the arrow you would modify the paint method of the arrow node, which means writing code. The arrow node is a “captive” type, encapsulated by RangeSlider.

In a polylithic toolkit such as 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 favors 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. And custom solutions clearly favor coders.

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 are writing called DateLens (www.cs.umd.edu/hcil/datelens). 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 custom, Jazz and Piccolo approaches. 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 (Figure 13).

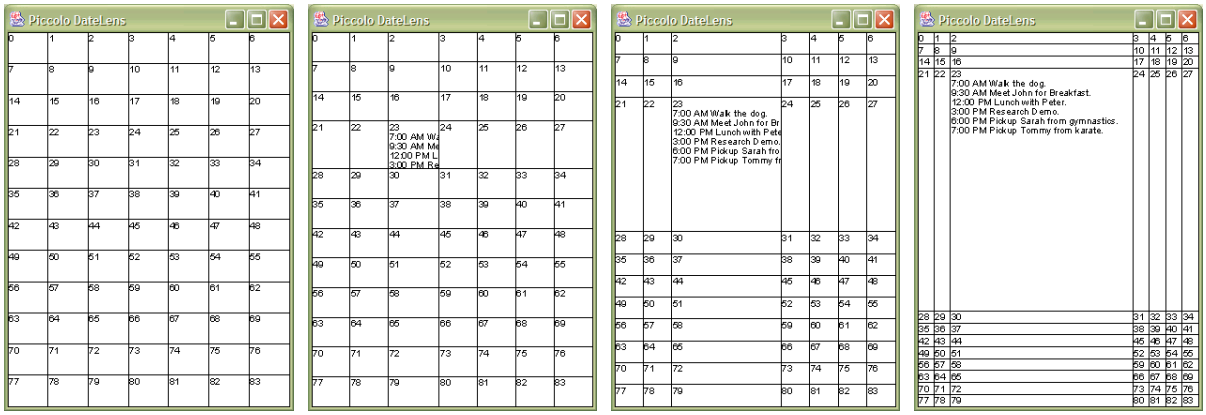

Figure 13: Screen shots from the calendar example during an animated transition.

As with the other two examples, the trade-offs in applying the three approaches to implementing the graphic calendar were similar. The toolkits made some tasks easier, such as picking and animating, but added some overhead since there are many small nodes. We analyzed the three solutions as with the first two examples, and the results are summarized in Table 4.

Table 4: Performance results of the three implementations of the graphic calendar

	Custom	Piccolo	Jazz
Scene render time	1.5 msec	2.2 msec	4.0 msec
Lines of Code	365 lines	224 lines	278 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)

In this case, the custom solution 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 non-optimal designs.

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 100x100 pixel rectangles. Times are reported as the average over 10 measurements. The results for the tests described here are summarized in Table 5.

Table 5: 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 %

These results show that the toolkits incurred an average 4% 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.

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 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.

Table 6: Scene graph manipulation times for Piccolo and Jazz. The notation "*n x m rects*" means *n* groups of *m* rectangles.

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

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 6). 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% of the total time per frame is spent on Piccolo scene graph manipulation.

CONCLUSION

This paper compares three approaches to constructing new interaction components. Jazz is a graphics toolkit built using a "polylithic" 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 polylithic approach also has costs.

Our experience with Jazz so far shows us that the biggest concerns with the Jazz design is ease of programming and efficiency. The application developer must manage many more objects than with a monolithic 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.

Piccolo provides an alternative toolkit design. In Piccolo, every node incurs costs – you pay for features up front. Piccolo is also more restrictive, because more functionality is built into the core node type. However, Piccolo applications are much easier to write, because the programmer has to learn fewer classes, and because those classes offer richer functionality.

Because efficiency is always a concern, we compared the efficiency of toolkit-based solutions to custom solutions. There are clearly many trade-offs between the custom and toolkit-based solutions described in this paper. So it is difficult to make a clear recommendation for any one approach. However, based on the case studies and performance analysis described here, we can state the following general design guidelines:

- For ease of programming applications, we have found monolithic toolkits to be the best.
- For use within prototyping applications where designers may want to change the look and feel without coding, polylithic toolkits offer the best structure.
- For very small and simple applications, custom solutions are best. The advantages of a toolkit-based approach don't appear until the application requires features such as region management, selection, layers, or fading.

Our final analysis is that structured graphics toolkits can make writing and maintaining code easier. The speed overhead of these toolkits is minimal, but memory usage is a concern. Further, when there is a choice, monolithic toolkits are likely to be easier to use for application programmers, but polylithic toolkits offer more flexibility which is likely to be especially useful for prototyping and design environments.

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 Jim Mokwa and Maria Jump for their early contributions to Jazz. Most importantly, the many users of Jazz and Piccolo have helped us 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] Perspecta (2000). http://www.perspecta.com/.

[2] SGI OpenInventor (2000). http://www.sgi.com/Technology/Inventor/.

[3] Adobe SVG Viewer (2002). http://www.adobe.com/svg/.

[4] ECMA Script (2002). http://www.ecma.ch/.

[5] ILog (2002). www.ilog.com.

[6] Java3D (2002). http://java.sun.com/products/java-media/3D/.

[7] Macromedia Flash (2002). http://www.macromedia.com/software/flash/.

[8] Bederson, B. B. (2001). PhotoMesa: A Zoomable Image Browser Using Quantum Treemaps and Bubblemaps. *UIST 2001, ACM Symposium on User Interface Software and Technology, CHI Letters , 3*(2), pp. 71-80.

[9] Bederson, B. B., Czerwinski, M., & Robertson, G. (2002). *A Fisheye Calendar Interface for PDAs: Providing Overviews for Small Displays.* Tech Report HCIL-2002-09, CS-TR-4368, UMIACS-TR-2002-48, Computer Science Department, University of Maryland, College Park, MD.

[10] Bederson, B. B., & Hollan, J. D. (1994). Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. *In Proceedings of User Interface and Software Technology (UIST 94)* ACM Press, pp. 17-26.

[11] 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.

[12] Bederson, B. B., & Meyer, J. (1998). Implementing a Zooming User Interface: Experience Building Pad++. *Software: Practice and Experience, 28*(10), pp. 1101-1135.

[13] Bederson, B. B., Meyer, J., & Good, L. (2000). Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. *UIST 2000, ACM Symposium on User Interface Software and Technology, CHI Letters, 2*(2), pp. 171-180.

[14] 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,

[15] Fox, D. (1998). *Tabula Rasa: A Multi-scale User Interface System*. Doctoral dissertation, New York University, New York, NY.

[16] Furnas, G. W. (1986). Generalized Fisheye Views. *In Proceedings of Human Factors in Computing Systems (CHI 86)* ACM Press, pp. 16-23.

[17] 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.

[18] John K. Ousterhout. (1994). *Tcl and the Tk Toolkit*. Addison-Wesley.

[19] Krasner, B. E., & Pope, S. T. (1988). A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Journal of Object-Oriented Programming, 1*(3), pp. 26-49.

[20] Linton, M. A., Vlissides, J. M., & Calder, P. R. (1989). Composing User Interfaces With InterViews. *IEEE Software, 22*(2), pp. 8-22.

[21] 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.

[22] 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.

[23] Perlin, K., & Meyer, J. Nested User Interface Components. *UIST 99, ACM Symposium on User Interface Software and Technology, CHI Letters, 1*(1), pp. 11-18.

[24] 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, pp. 227-231.

[25] Raskin, J. (2000). *The Humane Interface*. Reading, Massachusetts: Addison Wesley.

[26] Smith, R. B., Maloney, J., & Ungar, D. (1995). The Self-4.0 User Interface: Manifesting a

System-Wide Vision of Concreteness, Uniformity, and Flexibility. *In Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 95)* ACM Press, pp. 47-60.

[27] 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.

[28] 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.

[29] Ungar, D., & Smith, R. B. (1987). Self: The Power of Simplicity. *(OOPSLA 87)* pp. 227-241.

[30] Zaphiris, P., & Mtei, L. Depth vs. Breadth in the Arrangement of Web Links (1997). http://otal.umd.edu/SHORE/bs04.