Raun M Krisch

Grape Cluster
Fall 2004 - UI Team Report

http://www.cs.rit.edu/~rmk8024/grape

Team Members:
Raun M Krisch
Andy Rader
Chris Stelma
Michael Godleewski
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>INITIAL STATUS OF THE UI</td>
<td>3</td>
</tr>
<tr>
<td>GOALS FOR THE QUARTER</td>
<td>3</td>
</tr>
<tr>
<td>DOCKABLE UI</td>
<td>3</td>
</tr>
<tr>
<td>SIMILAR INTERFACES</td>
<td>4</td>
</tr>
<tr>
<td>HOW DOCKING INTEGRATES INTO THE EXISTING INTERFACE</td>
<td>4</td>
</tr>
<tr>
<td>DEPTH QUEUING WITH FOG</td>
<td>4</td>
</tr>
<tr>
<td>ADDING FOG TO THE GRAPE CLUSTER</td>
<td>5</td>
</tr>
<tr>
<td>PLUGGABLE UI COMPONENTS</td>
<td>6</td>
</tr>
<tr>
<td>INITIAL IDEAS</td>
<td>6</td>
</tr>
<tr>
<td>THE CURRENT PLUG-IN SYSTEM</td>
<td>6</td>
</tr>
<tr>
<td>TEAM DYNAMICS</td>
<td>7</td>
</tr>
<tr>
<td>RESULTS</td>
<td>8</td>
</tr>
<tr>
<td>FUTURE WORK</td>
<td>8</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>9</td>
</tr>
</tbody>
</table>
**Initial status of the UI**

The initial implementation of the interface for the Grape Cluster project was quite simple, where each user interface component directly corresponds to a feature of the underlying system. Whenever access to some part of the system was required, a component was added to the interface to trigger the specific action. This was done in order to quickly get a working prototype running as time was the most significant initial constraint. While providing all the necessary functionality, the initial user interface was difficult to maintain and difficult to extend.

**Goals for the quarter**

The high level goals for this quarter was to revamp the user interface to make it more user friendly, more visually appealing, and easier to maintain. This work would involve implementing a new interface using Swing, which has, at a minimum, the same functionality of the existing user interface. Additionally, the visualization part of the system, while functional, has some issues with realism, and depth perception that should be addressed if time allows.

After the first team meeting we came up with several areas which we thought should be researched or addressed through the quarter:

- Rendering Java3D to a Swing component
- Dockable interface
- A plug-in system where the UI is created dynamically at run time
- Switching the rendering backend to JOGL – a native interface for Java to use OpenGL
- Split the logical pieces of the interface into separate classes

For the first week, 3 team members began looking into the research areas above, while the remaining team member began to clean-up the existing user interface code such that we could reuse as much of the exist code as possible later on in the quarter.

**Dockable UI**

The Dockable interface was originally researched by Andy Rader and is a very impressive piece of work. The entire docking mechanism exists in a single class called
the JDockingPanel, and allows any existing Swing interface to have the docking ability with very few changes to the existing code.

The JDockingPanel class exists as a useful way for users of an application to move different logical “panels” of the interface to where they ultimately need it on the screen. This involves dragging or detaching a panel from the main interface window, and allowing it to turn into a free floating window capable of operating on its own. For example, the most important piece of an application can be detached from the main window, and maximized on a secondary monitor, while the less important pieces can be moved out of the way or even minimized from view. This system applies to the Grape Cluster project perfectly since the main goal of the project is to produce visualizations. Once the visualization is running, the majority of the interface is no longer required, and only the movie creating portion of the interface may be needed on the screen. The dockable system will allow the movie interface to be detached from the main window, which will remove unneeded controls from the view of the user.

**Similar interfaces**

The idea of such a utility came from applications like The GIMP, The GNU Image Manipulation Program, and Eclipse, a free Java IDE. The GIMP is set up so each part of the GUI exists as its own window, without there being any main window that contains the other components. This allows the user to position tools and windows exactly where they can be used most efficiently. Eclipse takes it a step further, and exists in a single main window initially, but allows the user to drag components out of the window, thus turning them into independent windows of their own. This behavior is also present in Microsoft’s Visual Studio suite since version 6.0 with the console, and debugging windows.

**How docking integrates into the existing interface**

The JDockingPanel can work completely transparent to any existing interface. The paradigm for the Grape Cluster’s interface before the docking ability was for a plug-in to return a JPanel object, which is then added to the interface. Now, a plug-in still, typically, returns a JPanel object, but instead of adding the JPanel directly to the interface, it is first added to a JDockingPanel, and the JDockingPanel is added to the interface. This very minor change to how the interface is constructed automatically provides the docking ability.

**Depth queuing with fog**

Depth queuing is a special application of fog used in 3D graphics systems, which is used to create the illusion of depth by making distance objects appear dimmer. Depth queuing is a per fragment operation and is accomplished by blending in a distant-dependant color
as each fragment is processed. Depth queuing works best when the distance dependant color is the same as the background color. The calculations for this technique are rather simple, and are describe below:

Let
- $C_f$ be the fog color
- $f$ be a ‘fog factor’
- $z$ be the distance from the fragment to the viewer

The rendered color can be calculated by:

$$C_s = f C_s + (1 - f) C_f$$

The for factor changes with the type of fog used, which, for most rendering systems, is typically a linear fog or an exponential for. Many rendering systems also support an exponential-squared fog type, such as OpenGL. Fog caused by clouds is usually best represented with linear fog, while other weather induced fogs, such as haze, is best modeled using a type of exponential fog.

**Linear Fog**
- $f = \frac{(\text{back} - z)}{(\text{end} - \text{start})}$

**Exponential Fog**
- $f = e^{(\text{density} \times z)}$
- $f = e^{(\text{density} \times z)^2}$

When using fog to perform depth queuing there are primary steps 3 steps to properly setup the effect. First, the fog color should be set to the color of the background. Second, the front, or start position of the fog should be set to the center of the 3D scene. And third, the back, or end of the fog should be set to the farthest point in your scene where objects are visible. The second and third steps ensure that the fog calculations only apply to what is in your scene. For example, if the scene starts at $z = 1$ and ends at $z = 10$, setting the front of the fog to 5 and the back of the fog to $z = 100$ will have little affect on the scene as domain of the fog is much larger then that of the scene. The fog equations above verify this assertion.

**Adding fog to the Grape Cluster**

All forms of fog in Java3D extend from the `java.media.j3d.Fog` class and are just another node that can be applied to the scene graph. I choose to use exponential fog, with a fog density of 0.8 the for the Grape Cluster visualization. Remember that exponential fog causes a haze to be applied over the scene, which is exactly what I wanted. The fog node is placed high in the scene graph so it is correctly applied to the scene regardless of the camera’s position, or how the scene is transformed.
Pluggable UI components

The pluggable interface was research by Chris Stelma and the resulting system works better than any of us through it would initially, but the initial idea for creating the interface at runtime was quite different from the final implementation. Chris should be highly commended for this work this quarter.

Initial ideas

Since all internal communication within the system uses an intermediate language, it was thought that interface components could be described by specifying the type of component, and what the component controls by specifying a command string. The interface would then parse the component description at runtime to generate the interface. Once running, a triggered interface component would send the specified command string to the language parser to induce the desired action.

This idea sounded good at first, but the more we discussed implementing this idea, the more the issues began to arise. We all agreed that action invoked by the interface can be describe as a string in the control language; we also agreed that any user interface component could be described by a language string and a component type, but all concerns after this brought up issues. Using a system such as this we would have to limit the types of controls the interface would have; also, this design would probably eliminate the ability to use custom user interface controls. The question of layout was what really caused us the abandon this idea. The amount of extra data required to properly lay out the interface the way the interface designer would like to interface to look would be almost impossible with this design.

The current plug-in system

We finally turned to a binary plug-in system. This idea involved loading compiled class files at runtime to generate the interface. If a class file implements a specific interface, and lives within a certain java package, the system will find the plug-in and load it into the interface at runtime, only needed a small 3 field XML data file to specify extra information about the plug-in. This method of generating the interface allows a programmer to use any component, and induce any action inside the interface. Interface components are no longer limited to sending commands through the language parser, but would be able to perform actions such as modifying the interface itself. This interface system also removed the problem of laying out the interface components as this is now the responsibility of the designer.

To implement a plug-in, 2 methods must be implemented from the plug-in interface. The first method is the build method. Build returns a Swing Component object that can be added to the user interface. This can be anything from a single button to a complex JPanel containing many components. The second method the plug-in must implement is
Send. Send takes in a single object and returns void. This method is designed to be a
generic communication point between UI components. For example, the History window
takes a string, which is then displayed on the interface. This string is suppose to be the
most recent language command sent to the parser, so that a user of the Grape Cluster
system can easily see what commands are being sent from the interface to the backend.
These strings are sent to the History window through the Send method. It is the job of
the history window to ensure that the argument sent to the Send method is, in fact, a
string. Even though many interface plug-ins will not need a communication end-point,
one is provided.

Every plug-in lives within the package spiegel.viewcontrol.mapview.plugins. Every
plug-in also contains a grapecluster.xml file, which describes 4 basic pieces of
information, which will aid the plug-in loader instantiate, and then layout the plug-in. A
sample XML file would like like:

```xml
<plugin>
  <class>Raun</class>
  <position>.9</position>
  <type>control</type>
  <registeras>Raun Control</registeras>
</plugin>
```

The class tags, tells the plug-in loader the name of the class to instantiate. If there are
multiple class files within the plug-in directory, this prevents the plug-in loader from
opening each class file in the directory to discover, which class implements the plug-in
interface. The position tag is a positive floating point value used to determine layout;
lower values have higher precedence. Type can be 1 of 2 vales, either “main” or
“control”. Plug-ins of type control live in a tab, in the lower right corner of the user
interface, and the tab order is determined by the value in the position tag. “Main” plug-
ins live, on their own, in any of the other 3 corners; “main” plug-ins are loaded in a
clockwise ordering, starting from the lower left corner, the ordering is dependant on the
position tag’s value. In either case, if 2 plug-ins have the same position value, the layout
ordering is undefined. Finally, the registeras tag gives a symbolic name to the plug-in.
This name can be used by other objects to retrieve the instance of plug-in using the
getPluginCapsuleByRegistration method. This method will return an object of type
Plugin; using the Send method, as described earlier, will allow messages to be sent to the
plug-in without verifying exactly, which type of plug-in was returned.

**Team dynamics**

The team worked together quite well, especially since half the team was not working for
credit, and also, we have never met each other before. I tried to run the team with a very
liberal approach. I did not have a clear idea of what the final product of this quarter’s
work should look like, but had a few basic goals I though should be accomplished.
Therefore, everyone on the team picked an area that they were interested in researching
and was given the very high-level task of “come up with something you think would work.”

Every team member came back with a working prototype much quicker than I expected. In every case, a demo application was available in a week’s time after the research was completed. Each person developed their demo application for another week before the integration process back into the Grape Cluster code began.

After the first few weeks I stopped scheduling formal team meetings as the major problems were solved. Communication was done through email, and if needed we would find each other during the week for a quick 15 minute discussion. We also talked for some time after the weekly project meeting on Tuesday nights.

Results

The results of the project speak for them self. All parts of the new user interface that were implemented function much better than I had hoped for. I believe the dockable interface will be a great benefit to the project, especially as the project matures. As more functionality is added the project, being able to only see the interface components that are interesting in will reduce desktop clutter, and increase productivity. The plug-in system should greatly help new user interface components to be added quickly into the system. Adding user interface components without having too edit, or even view, the existing user interface code will reduce confusion, and practically eliminate the breaking of existing code when new interface components are added to the system.

Since the whole team can not have write access to the project CVS server, I should have set of a private CVS server with a branch of the Grape Cluster code that my team could have write access to. By mid-quarter I was receiving weekly code updates from 3 team members. Integrating each member’s code back into the main code branch would consume a day’s worth of time I had reserved for working on the project. Combined with reviewing demo applications, my development of new code for the project quickly slowed during the middle few weeks of the quarter.

Future work

Much of the work in the JDockingPanel required calculating offsets and window positions to properly dock and undock the windows. We have just recently discovered the SwingUtilities class, which can perform many of the calculations we performed automatically. Refitting the JDockingPanel to use this library should be a relatively trivial, and would give the code a much cleaner look.

The plug-in system could also use a bit of work. The plug-in loading code was only mildly tested with loading class files, which are not of the right format. We believe that
the loader will gracefully reject class files that do not implement the correct interface, but this was not thoroughly tested. Also, the interface plug-ins must support may need to be modified. Currently, only 1 person outside of our team has implemented a plug-in, and has not reported any problems with the how the system works. It may be possible that the plug-in interface does not expose enough of the system for some future plug-in. Refitting the interface should not cause significant issues if needed in the future. Also, the XML parsing in the plug-in system was only mildly tested as well. It may be possible for bad data to cause unexpected behavior.

**Conclusion**

I would like to commend the excellent work by my whole team, but especially Chris Stelma and Andy Rader. These 2 guys worked extremely hard and fast all quarter long, producing some very clean and easy to use code. They quickly learned how the system worked and jumped right into the code. I believe they would be a very important asset to the Grape Cluster project in the future.