FlySanDiego: a Web-aware 3D Interactive Regional Information System

Steve Lukas[1], Mike Bailey[2]
Sony Online; San Diego Supercomputer Center

ABSTRACT

Communication of regional geographic information to the population as a whole should be a municipal priority, but sadly it is not. From traffic patterns to weather information to emergency information to proposed highways, a city or county has, in electronic form, all of this useful information and more. With the ubiquity of web browsers and the arrival of online 3D graphics technologies such as VRML and Java 3D, this information could and should be made available. By using Java and Java3D, the rendering power of an OpenGL-type application can be combined with multithreading, allowing a program to invisibly access data sets from Internet sites with dedicated threads while processing user interaction with another. Any type of relevant data can be transformed into a three-dimensional interpretation and mapped over the terrain that the user is analyzing. This prototype is designed to be extremely extensible and expandable in order to accommodate future revisions and/or portability. This paper discusses the issues surrounding the creation of such a model, along with challenges, problems, and solutions.

Keywords: visualization, GIS

1. INTRODUCTION

Display of geographical information is a common computer graphics application. Many commercial GIS systems exist, and it is an active research area. It has recently been used for battlefield war game simulation ([DURBIN98]), environmental display ([CLEMATIS98] and [RHYNE93]), and deforestation data ([CLARK98]). But, most GIS efforts are aimed at playing back the results of a simulation or displaying data gathered from sensors. They assume that the data has already been taken and is available in a local disk file. This project wanted to see what could be done with data that was not already local, data that is being created on the fly somewhere else on the Internet.

Communication of local geographic information should be a municipal priority. Using San Diego County as an example, our lives are filled with brush fires, earthquake concerns, traffic jams, proposed new construction, sewage spills, and water main breaks. In another part of the country, we could also be concerned with floods and snow-blocked roads. The point is that geographical information should not be solely the domain of municipal planners. It has a purpose, and a right, to be available to the population as a whole.

Much of this project was inspired by a bad brush fire in San Diego two years ago. The rainless summers in southern California create canyons and hillsides that are tinderboxes just waiting for a spark. And, once it occurs, the ocean winds churn and spread the fire like a fever. This particular fire was deep in a large canyon. As it climbed the canyon walls, it threatened major roads and housing developments. Fire department spotters in helicopters were

---

[1] Sony Online, slukas@station.sony.com
[2] San Diego Supercomputer Center, University of California San Diego, 9500 Gilman Drive, La Jolla, CA 92093-0505, 858-534-5142, mjb@sdsc.edu

tracking the fire and directing firefighting teams. All local TV news crews were on the scene, trying to describe the fire and the subsequent road closings and evacuation areas.

The problem is that news crews are not particularly adept at describing geographic information. In fact, one crew was confused about which directions were north, south, east, and west. So, the emergency information they were giving was wrong, and in fact could have sent evacuees into the fire rather than away from it. It was then that we began to wonder how this could be done better.

Access to the World Wide Web is becoming a standard feature in most homes. A nice feature of the WWW is that information can be put online once and people can come get it at their convenience. It occurred to us that the fire spotters could radio the information to a central unit that could place the fire information on the web. It then followed that, in fact, such a system could be made so general that a multitude of information could be communicated.

2. PROJECT GOALS

The objective of this project was to create a visual Regional Information System available to the general public. Using visualization techniques, we started with weather data and expanded our feature set from there. Our design allowed for measuring and interpolating dynamic temperature and wind data in real-time, along with displaying static features such as height and landmarks for a sense of location. Our intent was to make this the backbone of an emergency information system signaling the location and intensities of fires, earthquakes, floods and other disasters on our model set to mirror to-the-minute events.

3. DEVELOPMENT

We started with a polygonal terrain map of a section of the land, in this case San Diego County. The polygonal model was obtained from USA DEM data and was decimated from about 120,000 polygons to roughly 12,000 to accommodate faster rendering on smaller-scale computers. The height data was exaggerated to make areas such as mountain ranges more prevalent and recognizable.

After taking the polygonal structure, we applied two textures to the terrain to give a more accurate reading of the terrain area. The first and default texture was a satellite image of San Diego, which gave a general idea of the appearance of the different masses on the terrain.

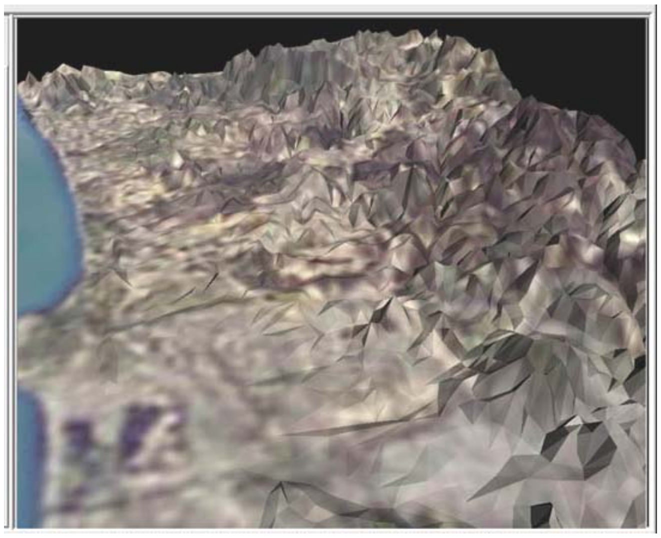

Figure 1: Satellite Image Texture

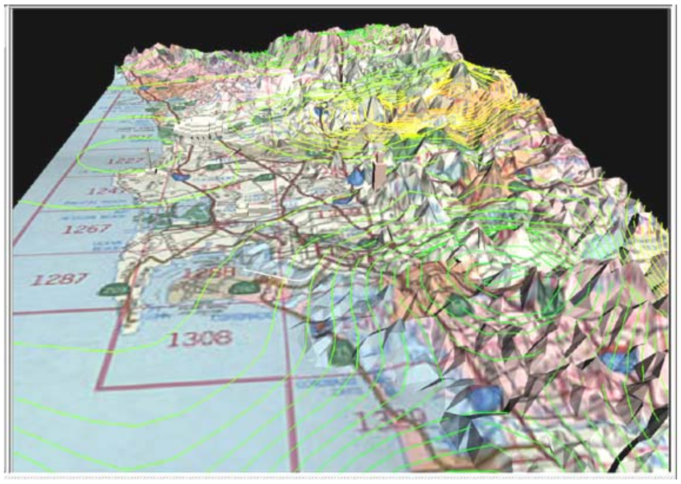
Figure 2: Scanned Map Texture

For the second texture, we used a scan of a map of the same area in order to show boundary and highway locations. The user is given an option to toggle between the three types of texturing: none, satellite, or map.

In order to enhance the sense of location, we added some constructed three-dimensional models of certain landmarks over the terrain to be instantly recognizable, such as major university landmarks and San Diego's Mt. Soledad cross. These can be turned on and off at will. The main benefit to these is that even if the satellite image is texturing the terrain, the landmarks will help identify the mapping even without highway assistance.

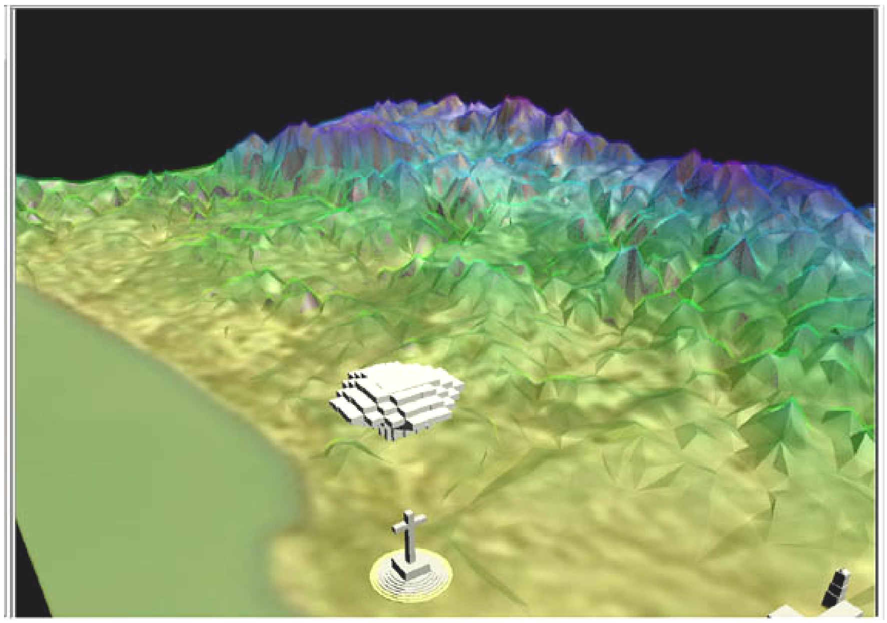
Figure 3: Landmarks

The scope of our simulation was to provide an interface for pushing information to the user. One of these informational sets is the elevation data. Elevation data is built-in to the model representation, and is represented with two options. Elevation can be seen as a transparent overlay with differential colors to signify height, and it can also be seen with contour lines that outline various height levels.

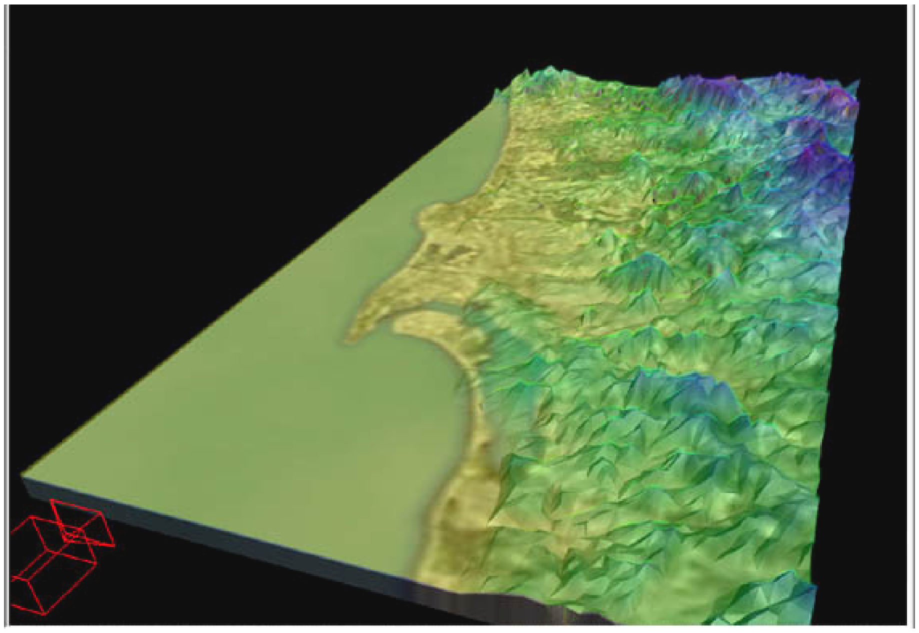
Figure 4: Transparent Elevation Overlay

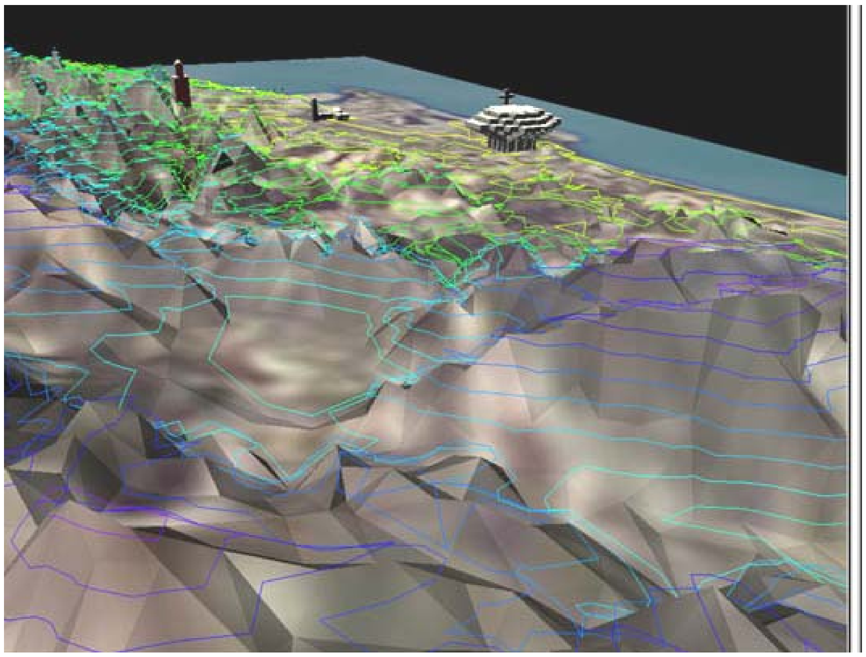
Figure 5: Elevation Contours

Although we have a default lighting model, the program also incorporates a thread that continuously computes the position of the sun and uses this as a parallel light source. The sun position is dependent on the time of day, the day of the year, and the latitude and longitude of where we are. The output of the calculation is the sun's location in relation with the terrain, and our primary light source is placed there. The user can also view a specific time and day other than the current one set in the application. When the application is launched, the current time is fetched and displayed. The user has the option of four lighting models: daytime, evening, the current sunlighting, or a custom lighting based on a specified time and day. If current sunlighting is selected, the program automatically recalculates the time and position of the sun throughout the day.

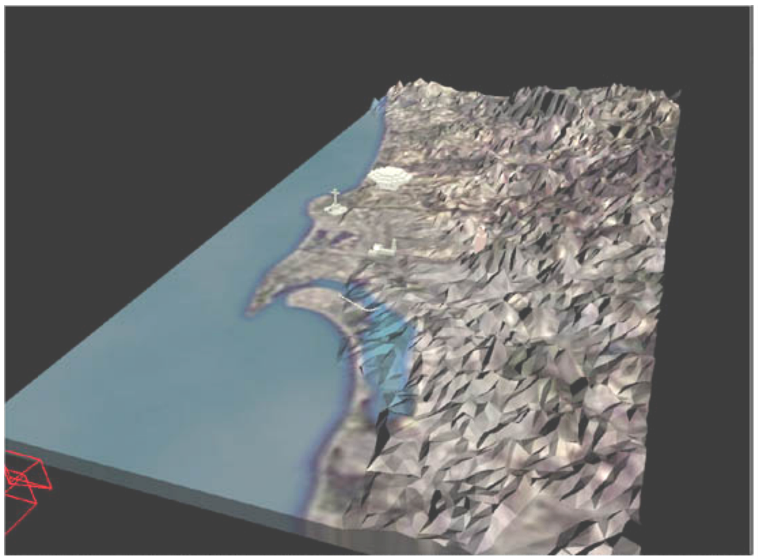

Figure 6: Time-of-Day Lighting

One important aspect of this application is that it relies on external sources for its dynamic data. Several Internet sites provide real-time data, updated several times during each hour, detailing weather information around the area. *FlySD* reaches out to these sites and parses their information to acquire the numeric data.

The sampled data is taken from as many points of measure as we can find. For temperature, datapoints are assigned colors based on their value. The color is then interpolated at the areas between datapoints. For the temperature map, these colors are incorporated into a copy of the surface of the terrain map to provide an overlay that can be placed over the terrain model. Contour lines are then drawn using the grid as a guide to represent the temperature values across the map. As with the elevation data, the temperature data can be viewed as contours or a full map. Additionally, temperature contours can be viewed with a slider to view one temperature measurement across the map.

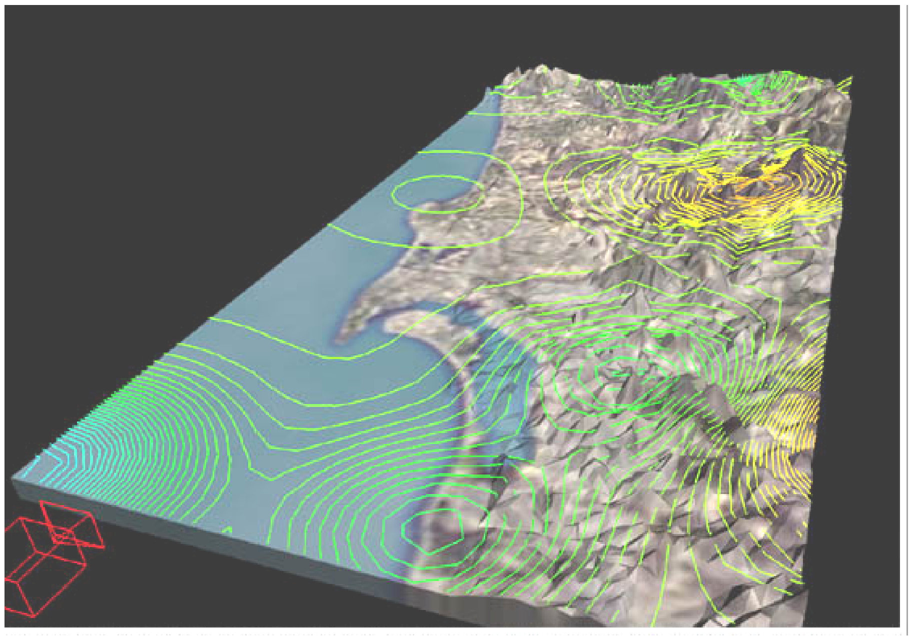

Figure 7: Temperature Contours

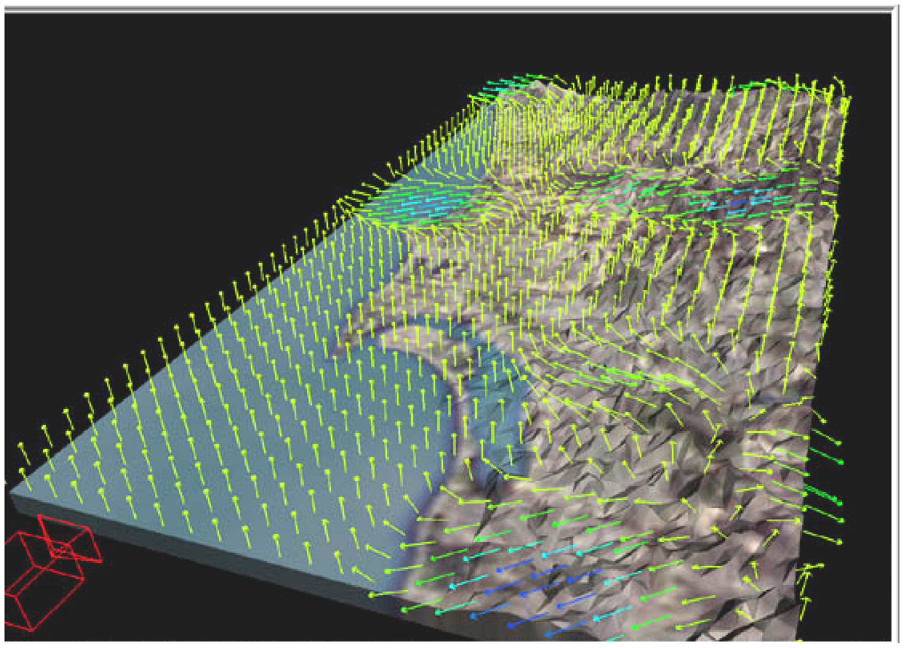

Figure 8: Wind Vectors

Wind is interpreted in a similar fashion, although instead of one numeric temperature value, there is a vector. These are interpolated along the same gridpoints as the temperature contours. However, here color represents wind strength instead of temperature value. Vector arrows are also drawn to signify the direction of the wind, and still wind is represented as vector arrows pointing straight up.

We refresh the time data, temperature data and wind data every few minutes while the user is still interacting with the scene. With the large masses of data to interpret, the initial loading of the scene is rather lengthy. To ease the frustration with this waiting period, the user is given a set of data to work with initially, with a portion of the controls unavailable until the part of the scene they are controlling is complete.

4. ARCHITECTURE

The entire application is written in Java and Java3D. When the application launches, a set of *PropertyResourceBundles* are read that determine what 3D files to use and the location of Internet data. A *webParser* parses each page and writes all of the results to a file that will be read by the main program. Since new Internet data is being read while old data is still being viewed, the new Internet data is not retrieved until the local file signals that it is complete. At that point, the program refreshes the data.

The heart of the program is in a Java package labeled as *FlyUSA*. *FlyUSA* houses all of the components and interface required to run the application, though it must be instanced by the *FlySD* package to provide *FlyUSA* with all the necessary parameters to run the program. The *FlySD* package contains all of the 3D model files along with information about where the datapoint hotspots are. Also included in the *FlySD* package are a *webParser* class which is customized to parse the specified page, and *PropertyResourceBundles* which specify application appearance, such as which user-interface controls to include and where they go.

The separation of *FlyUSA* and *FlySD* allows future developers to create a package for other cities, regions, or even states and beyond, provided the hardware can handle the large polygon count. What would be required of the developer would be 3D models to represent the terrain and landmarks, a master URL and parameters to fetch Internet websites, and a parser class to interpret the data. As long as the correct format is followed, *FlyUSA* would need no modification in order to run the new application.

Additionally, *FlyUSA* itself is designed so that it can be extended with new features. Each feature currently in *FlyUSA* has a description of the 3D data and a method that describes the user-interface widgets and adds them to a

Swing panel. In the Master3D class in the *FlyUSA* package, a list of the 3D data generators is listed in the order that they will be added to the scene, along with a description of a command that associates a value in the *PropertyResourceBundle* set with the method for adding the UI component.

5. CAMERA MODEL

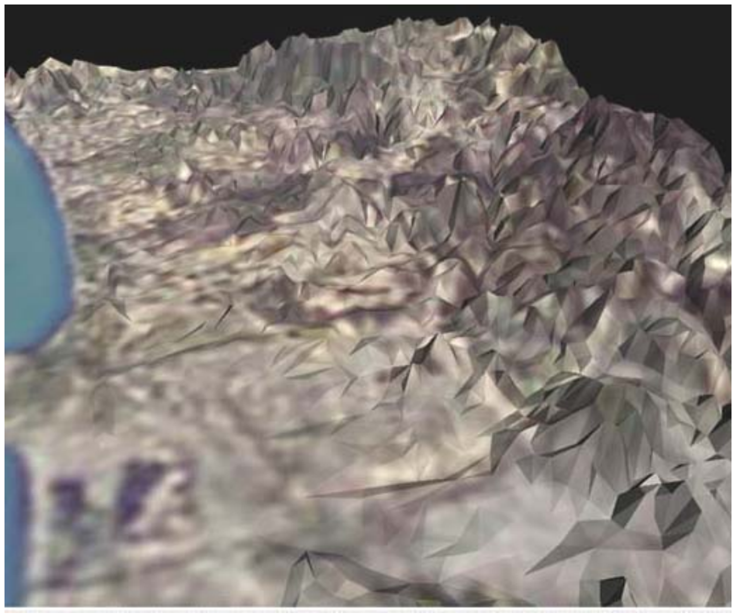

Figure 9: Camera Viewing

A very common construction for user interaction with 3D objects is akin to holding the model inside an invisible sphere. The user can rotate the model around the origin axis or increase the scale of the object to magnify it. This is known as “examine interaction”. In this construction, however, the user cannot choose any two vantage points on the model and see the relative viewpoint.

We wanted the user to be able to "flythrough" a physical terrain with true go-anywhere interaction. This required more than the examine model. We went to a model of a user-controlled camera. The left mouse button was attached to the viewpoint in a style where the user is looking through a camera, and click-dragging the left mouse button would rotate the camera's view: click-drag left, the camera rotates to the left; click-drag right, the camera rotates to the right. Click-dragging up and down rotates the camera in those directions as well, but limits were placed at the ninety-degree angle to prevent the user from cocking their head completely backward or forward.

While this allowed the user to freely look around, the user was still stationary. This control was added to the middle mouse button, and allowed the user to travel in four directions: forward, backward, left, and right. We called this the "Pan camera" because the user could look straight down over a piece of terrain and move forward to analyze the terrain as if the camera were attached to the underbelly of a helicopter.

A third and final control was added, "Camera level," to elevate and lower the camera to allow the user to fifth and sixth degrees of movement using the right mouse button.

Continuing with the camera model for interaction, we wanted the user to also be able to adjust the field-of-view angle. This is like changing camera lens’s focal length, from wide-angle to zoom. We integrated this with a slider placed in the camera control tabbed pane.

Finally, to make the camera model complete, we created a second stationary camera to act as a "God's eye view" of the scene. Viewing from this angle, any and all changes to the global scene that were made were instantly noticeable. This view created an excellent starting angle for the application, where the user can instantly grasp the terrain-at-large, and then jump into the moveable camera when desired. All of the camera controls are active regardless of which camera we are looking through, although all of the camera controls only affect the user immersion camera and only the viewing aperture angle changes both cameras. A wireframe representation of the user camera is shown on-screen in the "God's eye view", and the user can move the camera where desired before jumping into it, and can toggle back and forth.

6. JAVA3D AND ITS ROLE

Still in its relative infancy, Java3D ([SOWIZRAL98]) is a graphics API for displaying three-dimensional scenes. Java3D utilizes C calls to OpenGL beneath the abstraction layer. Although both APIs are designed for three-dimensional graphics, the two have rather differing structures to work with, each with its own advantages and disadvantages.

A key advantage of OpenGL is that it makes direct hardware calls. Since different implementations of OpenGL are used from system to system, OpenGL drivers must be developed for the specific hardware in order to use them to their fullest. Because of this, OpenGL sits closer to the hardware, and its calls are performed very efficiently. Java3D sits a step away from the hardware, but at times makes up for this with its optimized display scene graph. The biggest speed consideration seemed to not be the choice of the API, but the Java language itself. The more non-Java3D Java code that had to be executed, the slower the program got

However, Java has native support for multithreaded programming, while C++ does not. This was a crucial aspect to the project as we could acquire dynamic data from the Internet invisibly to the user who is busy interacting and observing. Once the data was fully acquired, it could be processed and updated asynchronously.

The structure of Java3D involves a series of nodes that are grouped together in a scene graph as parent and child objects down the hierarchy. Where OpenGL is designed to carry out an orderly series of "draw" routines every update to create a 3D scene, Java3D automates the process internally. From a programming standpoint, the nodal scheme allows for ease in programming of multiple camera views in a 3D scene by simply adding camera node objects to the scene graph and choosing one or more of them at any one time to be active.

The nodal structure of Java3d allows for the dynamic addition or removal of sets of data via *BranchGroups*. Each data set (i.e. temperature map, temperature contours, landmarks set, wind vectors) is stored within its own *BranchGroup*, and methods are provided for attachment and detachment of the Groups in order to satisfy any number of Groups for simultaneous display. All of the options are then presented to the user, who can turn any or all of them on or off at any time to customize what is being viewed.

Additionally, Java3D allows the use of more standard file formats for use in texturing objects, unlike OpenGL. The advantage of using image filetypes such as JPEG is Java3D's ability to import compressed image file formats natively without relying on obscure translation utilities.

Java3D's power as a standalone application provides the project with access to network data, multitasking, and portability. This project allowed us to explore Java3D’s features in-depth and report on its capabilities in a real-life application environment.

7. USER INTERFACE DESIGN

In the design of our prototype, user-control was a top priority. The design of the user-interface should be intuitive, customizable, and easily operational. We felt that a project as involved as this should naturally use Java's Swing set in order to give a greater set of controls to the user than Java's AWT had.

One type of panel is called a *JSplitPane*, which allows a panel to be divided either horizontally or vertically into two panels of variable size, much like an Internet Browser's frames feature. We wanted the user to be able to be able to resize any panel of their choosing, including the Java3D window. The interface then consisted of nested *JSplitPane*s to hold the buttons and sliders, along with a toolbar menu and a status bar.

In order to incorporate Java3D into a customizable Swing panel set, it seemed a straightforward task to place a Java3D *Canvas3D* object into one of the split Swing panels. Upon doing this, resizing the Swing panel containing the Java3D *Canvas3D* object became problematic.

Swing had settings for each panel in a *JSplitPane* to have its own preferred minimum and a preferred maximum setting, which should have allowed for each window to become larger or smaller using the mouse interface. What ended up happening was that although the user could make the Java3D panel bigger, there was no going back as the panel could not be resized to be any smaller again.

Swing does have the ability to use multiple nested panels, each with its own content. This ended up being the workaround, as Swing panels can be resized only if they do not *directly* contain a *Canvas3D* object. The panel that initially contained the Java3D object would now contain a *JSplitPane*, called *JSplit1*. *JSplit1* would be divided horizontally, with content in its top panel and null content in its lower panel. If we place the *Canvas3D* object in the top panel of *JSplit1*, then although the *Canvas3D* object is not resizable within *JSplit1*, *JSplit1* itself is resizable. Since the bottom panel of *JSplit1* has null content, and since we set that null content's preferred maximum size to zero, it appears as a very thin border and does not get in the way. To the user, *JSplit1* is not a split panel, but a panel holding a *Canvas3D* object with a small border at the bottom. To the application, *JSplit1* is a master controller for the resizing of the *Canvas3D* object.

The other pitfall uncovered with Swing is that Swing is classified in the Java world as using "lightweight" components. This means that Swing is composed entirely of Java code and does not refer to native data structures or peer classes. This is contrasted with to heavyweight components, which are elements that are interfaced by Java and need to subclass from other objects. Java3D is a heavyweight component, as it accesses 3D system calls in order to provide the OpenGL rendering.

Lightweight and heavyweight components are not meant to be mixed together. Any heavyweight component will overlap any lightweight component. This was an unpleasant discovery as our design involved a set of Swing user-interface panes bordering the Java3D scene window. All of the Swing components that were to appear as the top-most objects of the user-interface, such as tooltip text and pulldown menus, were overlapped by the entire Java3D window pane when passed over, as if the entire Java3D window was on its own layer hovering above the Swing elements.

The solution to making the Swing elements topmost was to force each set of Swing components to be heavyweight. This actually degraded performance of the application. According to Sun, it isn't desirable to have too many heavyweight windows. Since we are forcing several Swing components to be heavyweight to accommodate the Java3D panel, this became a sacrifice for the sake of a better user interface.

A legend was made to explain the colors used in the model. Unlike other panels in the application which would have increased workspace size when the entire application was sized larger, the legend panel had to be stretched as it needed no extra workspace size. For this we turned to the Java AWT drawing capabilities and placed the information in another *JSplitPane* to appear above the *Canvas3D* panel when desired.

8. RESULTS

8.1 Camera Model

Of all the camera controls, "Pan camera" was the one that gave the most trouble. By default, moving forward and backward was attached as moving in the latitude-longitude plane. But then if the user was not facing straight forward, this felt awkward, so the direction the user was facing had to be incorporated to achieve the desired effect.

Calculating the angle of rotation of the camera viewpoint in degrees and multiplying by the direction of the pan command moves the camera in the direction relative to the user's viewpoint. For example, if the user is facing backward, (or at any angle) click-dragging up would move the camera forward in the user's direction. Again, since the user would only move in the latitude-longitude plane with this camera control, staring at the sky and moving forward would not move the user in the Y direction, but would instead act as if the user were, indeed, staring at the sky and walking down the street.

Additionally, since we cannot assume that every user will be using a three button mouse, a set of camera controls was placed in one of the tabbed menu panes. These controls allow the user to select which of the three modifiers would be attached to the left mouse button. As a style point, for any of the three primary controls selected for the left mouse button, the remaining two controls were reshuffled onto the middle and right mouse buttons to avoid two buttons having the same effect.

8.2 Speed and Detail

In order for the application to run smoothly, the polygonal detail was reduced. This is a compromise that will hopefully be less necessary over as technology advances, where higher polygon accounts can be achieved and rendered in a reasonable amount of time. More detail would allow a finer representation of the terrain and atmosphere. The addition of more landmarks and animated features would heighten the immersion of the application.

8.3 Density of Data Points

Presently, the number of data locations found is extremely sparse and not an accurate indication of the continuous weather pattern around the city. As more research goes into the program and more datapoints are found, it will be a simple matter to edit the *PropertyResourceBundles* to accommodate the new data.

8.4 Applet Options

Due to security issues with Java applets, *FlySD* is written as a Java application that cannot be instantly run as an applet. In order to convert it to an applet, we would need to modify the Internet data retrieval methods and file writing.

One way of doing this is with perl scripting. As a Java applet can only access Internet data from the server that it resides on, a perl CGI file can be written and placed on the same server. The perl script could take as input a URL parameter, open a socket connection, retrieve the information page from the site, and send it back out as a web page, bypassing the security issues. The tradeoff on this technique is the usage of an external perl program and making twice as many URL requests, one for the CGI file and one for the socket connection, for each URL to fetch.

9. CONCLUSIONS AND FUTURE WORK

This application was done as a prototype so that we could learn what it would take to really turn this into a municipal tool. From that, we have learned the following:

- This application was too slow to be as useful as we had hoped. This is most likely due to the use of Java/Java 3D/Swing instead of C++/OpenGL/GLUT. Much work is underway to speed Java applications using better JIT techniques. We hope this will be enough.

- Speed aside, Java was a great language in which to implement this application. The easy availability of threads made it straightforward to invisibly check online information and re-display it when it changed. Java’s URL classes made it straightforward to gather information from the WWW. This would have been much more difficult in C++.

- Java 3D is at its very best in navigation-type applications such as this one. Java 3D's optimized scene graph display list design anticipates threads that are asynchronously poking new information into the display. Java 3D updates the scene display when necessary and does it in a way that does not create graphics glitches from incomplete scene graph edits. This synchronization would have been much more difficult in OpenGL.

We would like to redevelop this application in the future to be the true municipal communication tool we had originally envisioned it to be. We envision that multiple municipal departments would provide information by creating files on their own web sites. This program would then access that information periodically or by user direction and be include in the display. In this way, any interested citizen could query a variety of areas of interest, from current weather or traffic conditions to general city plans to reactions to emergency situations.

ACKNOWLEDGEMENTS

This work was supported by an SDSC/NPACI summer internship.

REFERENCES

[CLARK98] Dru Clark and Rosemarie McKeon and Richard Marciano and Michael Bailey. "Rear-Projecting Virtual Data onto Physical Terrain: An Exercise in Two Senses Being Better Than One", *Proceedings of IEEE Visualization '98*, October 1998, pp. 451-454.

[CLEMATIS98] A. Clematis and M. De Martino. "Using GIS to Visualize Environmental Information: A Case Study Based on Digital Terrain Models", *Computer Graphics International 1998*, June 1998, Hanover, Germany.

[DURBIN98] Jim Durbin et al, "Battlefield Visualization on the Responsive Workbench", *Proceedings of IEEE Visualization '98,* October 1998, pp.463-466.

[RHYNE93] Theresa Rhyne and Mark Bolstad and Penny Rheingans and Lynne Petterson and Walter Shackelford. "Visualizing environmental data at the EPA", IEEE Computer Graphics & Applications, Volume 13, Number 22, March 1993, pp. 34-38.

[SOWIZRAL98] Henry Sowizral, Kevin Rushforth, and Michael Deering, *The Java 3D API Specification*, Addison-Wesley, 1998.