CSM #4 Geophysics Interactive Graphical Display
Summer Field Session 2006

Christine Brady
Adam McCormick
Zachary Pember
Danielle Schulte

Client: Dave Hale
Advisor: Cyndi Rader

Executive Summary:

The primary goal of this project was to expand upon the existing Mines Java Toolkit (JTK) created within the Geophysics department at the Colorado School of Mines This toolkit is used to program visuals for scientific applications. Current applications do not provide good interactive models, and, for this reason, Mines is developing tools that will assist in development of better applications. Our task was to design and implement a package of classes as an extension to the Mines JTK that would create and display triangulated surfaces in 3D, referred to as the JTS (Java Triangulated Surface).

Requirements of the project included restrictions on the license under which the code must be written and the language of the code. The use of the included OpenGL wrappers was also required. To complete the project we needed to research the most effective means of implementing a bounding sphere hierarchy, creating a framework for interacting with 3D surfaces using a 2D mouse, and developing algorithms to produce relevant data such as triangle normal vectors, spatial medians, and average vertex normal vectors. This research emphasized accuracy and efficiency.

The final design included code written from scratch as well as code copied from within the Mines JTK, in order to ensure license compliance. The code reliably generates surfaces from triangle soup, surface functions, and standard triangulated surfaces such as the Stanford bunny. Test classes included with the code demonstrate this functionality using various setting to determine color and material states, as well as a logical increase in complexity in the use of the JTS code. Aspects of the code such as the maximum number of triangles were analyzed for efficiency and adjusted according to our results. All problems with rendering were solved as we developed increasingly sophisticated test code.

The requirements of the project as we understood them have been fulfilled. We have provided code that extends the Mines JTK into the rendering and manipulation of triangulated surfaces. Future extensions to this project lie in three main areas: interface extension, extended format support, and data manipulation. This package should add all required functionality to the Mines JTK and set a robust foundation for increasing utility in the future.

Abstract:

The Geophysics department at the Colorado School of Mines is developing a Java toolkit that can be used in programming scientific visual applications. Specifically, it provides graphics tools that allow display and interactive manipulation of 3D surfaces. This is important to scientific applications, particularly in the field of geophysics, because the ability to interact with surfaces and manipulate joints and faults can greatly increase understanding of a material, such as the Earth's subsurface. Current applications do not provide good interactive models, and, for this reason, Mines is developing tools that will assist in development of better applications.

Our task was to design and implement classes and methods for creating and displaying 3D triangulated surfaces. We used graphics tools from the OpenGL library that have been wrapped in Java, along with advanced data structures such as scene graphs. The problems we have addressed include: determining the most effective means of implementing a bounding sphere hierarchy, implementing a means of interacting with 3D surfaces using a 2D mouse, creating of data types to facilitate rendering, and designing algorithms to produce relevant data such as triangle normal vectors, spatial medians, and average vertex normal vectors.

Project Introduction:

The goal of this project was the creation of an architecture for storing, organizing, and rendering triangles within the Mines Java Toolkit (JTK). This architecture implements the OpenGL wrappers written in Java by Dave Hale and works with the Mines JTK scene graph language (SGL). While this architecture was not specific to rendering surfaces, such is the main reason behind this project. Its main use will be the display of data collected from subsurface analysis. However, our client tasked us w with making this implementation as general as possible. To this end, we added much extra functionality above and beyond our general mandate.

The first element of this project was to learn the Mines JTK and how to integrate with it. The Mines JTK is a package containing Java classes designed to assist in the development of scientific visual applications. Specifically, it provides graphics tools that allow the display and interactive manipulation of 3D surfaces and objects. This is important to scientific applications, particularly in the field of geophysics. The ability to interact with surfaces and manipulate joints and faults can greatly increase understanding of the data involved, such as that collected about the Earth's subsurface.

The goal of our project was to add triangle rendering functionality to the Mines JTK. This was accomplished by rendering what was termed “triangle soup” by our client. More specifically, our application takes vertices stored in an array of float values and renders the triangles using the OpenGL wrapper in the Mines JTK. An array of integers could also be specified to indicate which of the vertices would be used in each triangle, thus making triangulated surfaces easier to pass in.

To work with the Mines JTK SGL, each group of triangles is divided into manageably-sized subsets. This allows for easier tracing of picking operations but slows the render, so a balance had to be found. We were also tasked with producing a means of using such famous triangulated surfaces as the Stanford Bunny as test cases for our triangle groups. Our classes integrate with the Node-Group mentality and structure set forth by the SGL framework. It is designed to work transparently within this framework as if our implementation was not dividing the triangles into subgroups at all.

Requirements and Specifications - Research and Development:

For this project, we learned to use both the Mines JTK framework and Dave Hale's OpenGL wrapper software. We studied and maintained the current build configuration for the JTK so that we could build a new branch of it and produce the Java Triangulated Surface (JTS) package. The primary class within this package was to be the *TriangleGroup* class. This class was to be created with a stream (more literally an array) of float values representing Cartesian triplets and, optionally, another array of integer triplets representing the vertices used to create each triangle in the group. This project will also produce a set of utilities which will organize the *TriangleGroup* and organize it into a bounding volume hierarchy.

Our task was to determine the most advantageous implementation of this hierarchy and this class in order to optimize picking and culling of the *TriangleGroup*. Our research included triangulated surfaces in general, such as how they are stored and how they may be displayed with OpenGL. We researched how normal vectors are used in displaying and shading algorithms. In this, we determined both a means of calculating the normal vectors at each of the triangles' vertices and an architecture for storing both the calculated normal vectors and the vertex locations. We also determined how best to display the group and a means of interfacing with this display.

Requirements and Specifications - Design and Implementation:

We were tasked with writing classes to render triangles from an array of vertex locations and determine optimal partition size for dividing the triangles into bounding volumes. Optimal partitioning, in this context, was defined as bounding volume partitions that minimize the average ray tracing and rendering time. The goal of our algorithms was to accomplish this using a top down approach that begins by dividing triangles into approximately even groups, and continues by subdividing the groups into smaller groups. This continues until all bottom level groups contain an approximately equal number of triangles below a predefined maximum number. The final groups are then placed into bounding spheres to allow for quick ray tracing.

Our deliverables were to be coded completely in Java and had to be made compliant with the Common Public License (CPL). Thus, many "Free" pieces of readily available code could not be used. Most notably, no code under the GNU Public License (GPL) or any of its counterparts could be considered. This meant that the vast majority of our code and algorithms had to be written from scratch. The implementation also required us to borrow heavily from the existing JTK framework so as to minimize redundant functionality or usage conflicts.

We used the many examples set forth in the scene graph section of the Mines JTK and our knowledge of Java and graphics to design the *TriangleGroup* class, an algorithm to create a bounding volume hierarchy, an architecture for storing and retrieving vertices from the hierarchy, and utilities to aid in creating *TriangleGroup*s. The hierarchy was to be composed of nested spheres. This way the hierarchy may be traversed recursively to find any triangles along a given path, as when tracing a picking vector, or in a given volume, as when choosing what triangles are to be rendered. All algorithms were to run in linear time and to be as efficient as possible.

Design - Architecture:

This project required the design of several algorithms in order to fulfill the functional requirements. The algorithms include calculating the vertex normal vectors and controlling the division of the triangles.

The first significant algorithm is the computation of vertex normal vectors. This is an important aspect of rendering triangles as it determines how light is reflected. In order to calculate vertex normal vectors, the normal vectors of the triangles must first be calculated. The normal vectors of the triangles are calculated by forming two vectors from the three vertices of a triangle and then taking the cross- product of the two vectors. The vertex normal is an average, weighted by area, of the normal vectors of each triangle sharing the given vertex.

The division algorithm splits the group of triangles in half and is used recursively until every group has less then a finite maximum number of triangles. The purpose is to isolate small groups of triangles that can be easily rendered and searched through. In order to select specific triangles, the SGL framework must sort through every triangle within range. The algorithm provides limits so the framework only sorts through a sub group of triangles. The division is done in three dimensions according to the median of the triangle centroids. First, the extrema in each dimension are recorded. The range of each dimension is then calculated. The dimension with the greatest range is the one that is divided. This ensures that divisions occur where they will be most useful. Then, the coordinates of the centroids in the given dimension are sorted into numerical order so that the median can be found. The triangles are divided along this median.

Implementation - Architecture:

The algorithms described involved implementing a series of methods. The calculation of normal vectors is implemented at the top level and called as part of the *TriangleGroup* constructor before the vertex values are passed into the *TGNodeGroup* to be divided. Division of the triangles takes place in the *divide* method and uses several methods including *findExtrema*, *quickPartialSort*, *divide*, and *getCentroids*. The *divide* method is called as part of the *TGNodeGroup* Constructor.

The calculation of the normal vectors takes place in the *calculateNormals* method. First, two vectors are formed by subtracting the values of two vertices from a shared third vertex. The cross-product function is copied from the *Vector3* class within the Mines JTK and used to mathematically find the triangle normal vector. It was a non-functional requirement that we copy the methods we needed from *Vector3* instead of simply calling the function. For each vertex, the method compares that vertex to the vertices of all the triangles and counts how many triangles share that vertex. Keeping a count of how many normal values are in the array allows a cumulative average to be calculated. As new triangles that share a given vertex are found, the existing normal at that vertex is multiplied by the current count, then the normal of the new triangle is added to form the sum of all such triangles, and then that sum is divided by the newly incremented count, creating a running average. This average is already weighted by area because the normal vectors have not been normalized and are proportional to the area of the triangles.

Once the normal vectors are calculated, the position coordinates and the normal vectors are passed into the *TGNodeGroup* constructor where the division of triangles into subgroups occurs. The constructor first calls *divide* which then calls *quickPartialSort* to find the median. The *quickPartialSort* method then calls several other methods. The first method, *findExtrema*, locates and records the minimum and maximum values of the x, y, and z coordinates of every vertex. This method is called within *quickPartialSort* in order to calculate the necessary ranges. Whichever dimension has the greatest range is the one by which the centroids are ordered numerically.

The method *getCentroids* is also called within *quickPartialSort* in order to obtain an array of values representing centroids from an existing array of vertices and an index. When the centroids are sorted, the median is pulled out. The method *divide* calls *quickPartialSort* and then divides the triangles. While each group still contains more then the maximum number of triangles, *divide* creates *TGNodeGroup*s and thus, *divide* is recursively called. When the number of triangles in the groups is below the maximum number, *divide* creates *TGNode*s, which can be interpreted by the scene graph framework and can be rendered.

Design - Visualization:

A main requirement of this project was that functionality be included for displaying triangulated surfaces. To provide this, triangle rendering methods are included. The rendering functions use the Mines JTK OpenGL wrappers so that they are both fast and consistent with other rendering functions in the Mines JTK.

The included functionality accomplishes several tasks. First, the methods render a triangulated surface from an array of coordinates. The rendering methods use the triangle coordinates and the bounding sphere computed in *TriangleGroup* to produce a display of the surface. Additionally, the rendering functions use the vertex normal vectors to enable lighting and to allow for surface coloring input. Finally, the rendering methods enable the user to zoom and select objects within the display.

To test the rendering functionality, *TGTest* classes are designed to create variously sized and shaped surfaces that utilize different aspects of the *TriangleGroup* class. The first one produces a large pattern surface that confirms rendering is functional and quick. The second class produces a surface from a Math sine function that demonstrates the lighting and coloring functionality. The rest of the *TGTest* classes create different shapes, such as a bunny and dolphins, and then try to merge them without creating a selection error.

Implementation - Visualization:

At the lowest level of our implementation, the *TGNode*, all of the functionality to render the triangles is implemented. Whenever a new *TGNode* is created, it calls a series of OpenGL functions to prepare itself for rendering; these are called using the method *setupDraw*. The method *setupDraw* sets up all the graphics information needed by the *draw* method. It calls makeArrays, creates the buffers, and sets the material (either to the default or a specific state that has been passed in). The defaultMaterial is a shiny surface that has an ambient component, a diffuse component, and a white specular component (much like a shiny plastic).

The method makeArrays creates the float arrays for the normal vectors, vertices, and colors. Since the OpenGL functions require an ordered array of vertices, this method creates an array from the given vertices and indices. It also calls the *getNormal* method to create the array of normal vectors needed to render properly. For colors, the default color is a rainbow mixture dependent on the x, y, and z coordinates but you can also pass in a *ColorFunction* to set the colors. The interface *ColorFunction* allows the user to set a method to define the color of the surface. The required method takes in an xyz-coordinate and returns a color in the RGB format.

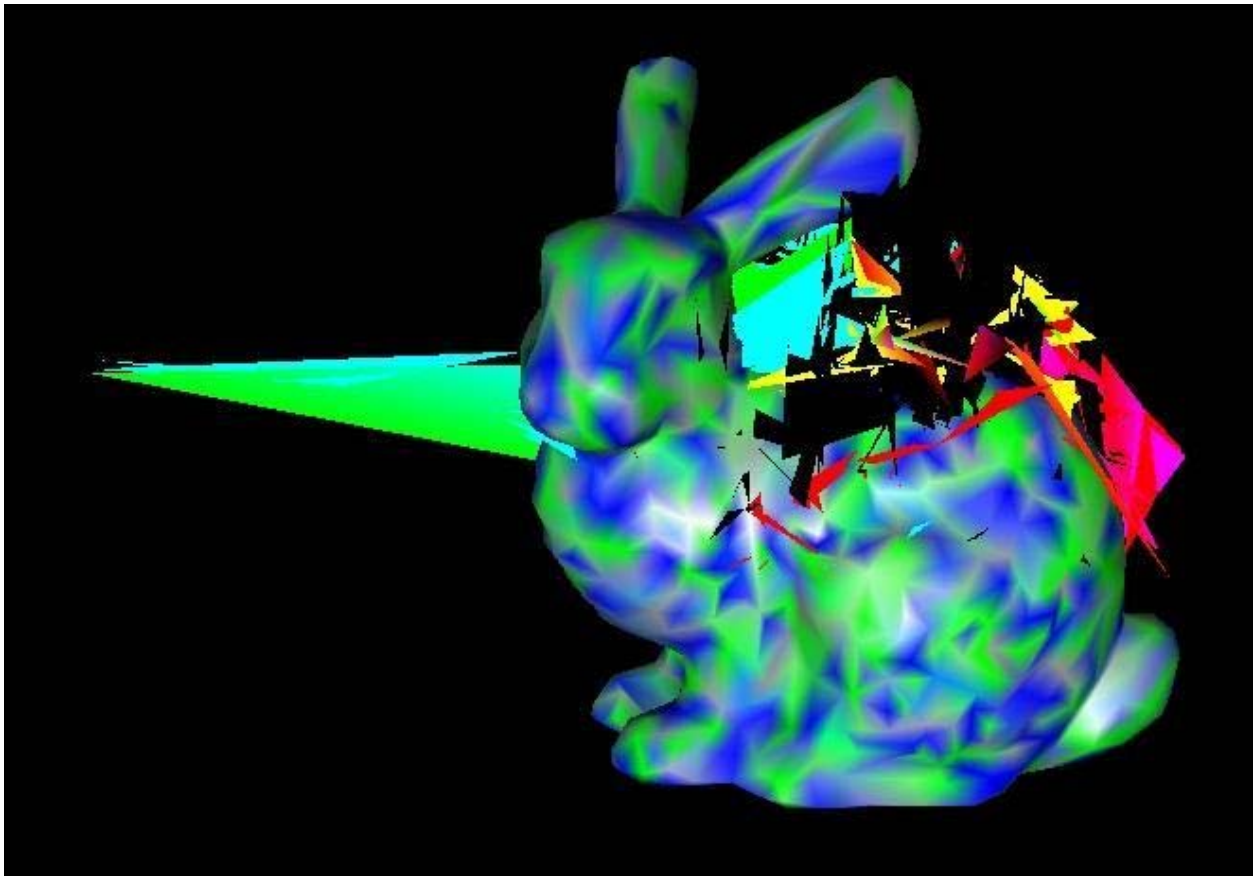
The method *draw* uses OpenGL to render the triangles in each *TGNode*. OpenGL requires the use of float buffers to render triangles, so *vertexBuffer*, *normalBuffer*, and *colorBuffer* are created in the *setupDraw* method to pass in the vertices, normal vectors, and color data. Using these buffers, *draw* then renders the array of triangles in the *TGNode*. Every time a change (rotation, translation, zoom, etc...) is made to the image or to the display screen, *draw* is called. Whenever an object is selected or unselected, the method selectedChanged marks the current image for redrawing using the dirtyDraw method. If something has been selected, *draw* then sets the width of the white highlight using glPolygonOffset and draws the outline of the selected triangles on top of the existing image. Since the rendering must be done at this level and the *TriangleGroup* must be dealt with as a whole, every *TGNode* must render as if it were selected whenever the Triangle group to which it belongs is selected.

Selection is done by the SGL Framework using the *pick* method. *pick* uses a 2D mouse to select a 3D object. It does this by drawing a line from the near cutting plane to the far cutting plane and testing to see which of the bounding spheres it intersects. It will then test within the bounding sphere to see if the line intersects with any of the triangles. If an intersection is found, the *TriangleGroup* becomes selected and this changes the execution of the *draw* method accordingly.

Project Progression:

Initially, the project began with a few setup delays due to the way the graphics cards worked with the OpenGL and with some of the environment paths. After our workspace was prepared, we spent approximately 2 weeks developing theory and trying to familiarize ourselves with the structure of the Mines JTK. Once we began implementation, the process took a derived form of extreme programming with paired programming, frequent builds, and design refactoring whenever the code we had written either didn't work right or was too inefficient. The last week of coding primarily focused on creating test programs and debugging.

One type of problem we encountered included selection errors that either crashed the program when a pick was made, or selected only a small group of triangles rather than the entire object. Additionally, we had problems with extra random triangles being rendered, which we termed "The Black Plague," because it rendered odd black triangles that seemed to erase large parts of the surfaces, and "Bunny Cancer," because it spawned randomly colored, positioned, and oriented triangles in addition to the models (Pictured Below). Finally, with some help from our client and a bit of debugging, our project was completed.



Conclusions and Future Directions:

As the project unfolded, we soon found that much of the design had to be altered as the implementation proved many changes were advantageous and many design elements were impossible. This demonstrated the need for frequent iteration and constant integration. It also forced us to work collaboratively on many of the more difficult problems and allowed us to bond as a team. This endeavor has also shown the Mines JTK to be a robust alternative to Java 3D.

Within the scope of this project, we have produced an intuitive and flexible solution for creating and using triangulated surfaces. Our solution allows for use with unconnected triangles as well as triangulated surfaces and has functionality built in to produce such surfaces from both functions and properly formatted files. We have included as much optional functionality as our scope has allowed, even going so far as to produce interfaces and scripts for user-defined functions and the import of models from VRML. We have produced many test classes to demonstrate the functionality of our solution which should, collectively, showcase the use of our *TriangleGroup* and TriangleUtils classes, and our SurfaceFunction and *ColorFunction* interfaces. These test classes should also demonstrate a logical progression of complexity for the use of *TriangleGroup*.

Future extensions to this project lie in three main areas: interface extension, extended format support, and data manipulation. While we have produced Test classes which implement a very functional interface, *TriangleGroup* itself may only be interacted with at a superficial level; the view may be changed but not the actual model. There has been some talk between ourselves and our client about the functionality to drag and scale the model as well as alter the camera's viewpoint. Currently, only a very simplistic file format is supported and the only functionality to translate other formats is a Perl script used separately from the rest of the program. With some further work, such popular formats as PLY, in which the Stanford Bunny is distributed, could be accommodated. Functionality could also be added to manipulate data and save the manipulated data to a file. Currently, the appearance of the data (*ColorFunction*'s and transforms) may be altered easily, but the data itself is not changed so that changes are not seen on reload. Further functionality would allow data to be manipulated and transferred which would make the entire system more functional.

Reference:

[1] D. Hale, "Mines JTK: Revision 487," http://boole.mines.edu/jtk. Last accessed on June 19,2006.

[2] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL Programming Guide Fifth Edition.* Boston: Silicon Graphics, 2006.

[3] "JavaTM 2 Platform, Standard Edition 5.0 API Specification," http://java.sun.com/j2se/1.5.0/docs/api/index.html. Last accessed on June,19 2006.

[4] Stanford University Computer Graphics Laboratory, "The Stanford 3D Scanning Repository," http://graphics.stanford.edu/data/3Dscanrep. Last accessed on June 19, 2006.

Figures - Examples Taken From Test Code