Development of a 3D Graphics Engine
Computer Systems Lab, 2006-2007

Kevin Kassing

May 30, 2007

Abstract

Computer users are already familiar with the concept of a two-dimensional graphical user interface for manipulating and visualizing data, in the form of text, spreadsheets, and similar documents. Recently, graphics hardware capable of rendering 3D user interfaces in real time has permeated the consumer desktop market. Numerous toolkits such as QT, GTK, and Swing exist for the purpose of simplifying the development of widget-based 2D graphics applications, but programming in three dimensions poses a bigger problem to developers, as the tools and concepts have been in development for a shorter period of time. Standards essential to 2D application development, such as image formats, have no acceptable 3D parallel (geometry is stored in arbitrary and often poorly documented formats). Tasks that are simple in 2D, such as detecting whether a mouse click causes a button to be pushed down become more complex in 3D. The goal of this project is to simplify the development of three dimensional interface programming, by creating a less complex way to manage the OpenGL render state and by making a unified mesh rendering interface.

Keywords: quaternion, shader, particle engine, mesh structure, material definition, triangle strip

1 Introduction

There are a number of applications which benefit from the added interactivity of 3D graphics, even despite the lack of a widespread 3D input device. The most obvious software genre to benefit from 3D graphics is computer games, but data visualization and simulations for various scientific fields are greatly aided by the addition of a dimension. The purpose of my project is to create an engine architecture which will aid the developer in using advanced graphical techniques and perform basic optimizations without requiring the developer to know virtually anything about OpenGL. Though I am developing some features that are primarily of interest to game developers, the goal is to make the engine useful for other purposes. The API should be systematic, using descriptive identifiers, and core functions should be optionally overrideable to allow the developer to optimize and customize as much as possible.

2 Background

3D graphics engines have been in existence since before dedicated graphics hardware was available. As realtime 3D rendering became possible on a workstation level over 10 years ago, developing applications which take advantage of the added third dimension has become an increasingly popular field. Modern hardware standardization allows 3D applications to be developed for a wide user base. The availability or the DirectX and/or OpenGL API on most desktop platforms simplifies the task of 3D development, but as the power of modern graphics hardware has increased, so has the effort necessary for coding. Today's hardware is sufficiently fast that programs need not have graphics code tightly embedded for optimization purposes. Well-developed 3D engines allow the developer to shift their focus from the implementation of 3D graphics and instead focus on interactivity. 3D graphics engines are available in most programming languages, and may have varying internal architecture.

While some applications merely take advantage of the added dimension for the purpose of conveying more information, some strive for realism. Computer games have long benefitted from the ubiquity of computer graphics processing capability, and use lighting, textures, materials, and recently, shaders (which can be used to modify the fixed graphics pipeline for more rendering capability and flexibility) to achieve suspension of disbelief. Additionally, water, fog, sky, and precipitation are becoming commonplace in applications that demand immersive environments. Implementing these effects takes time, much of which could be saved by using a open source or commercially licensed graphics engine.

There are a few main tasks that are often the responsibility of a graphics engine. Most 3D applications will require loading model data from a file, which is often done by the engine. Animating and rendering these models is usually performed by the engine, but are sometimes done by the application itself for optimization purposes. Managing textures and material definitions, which provide information as the the appearance of polygons, are usually managed by the engine. Collision detection and, in some cases, kinematics are accomplished by the engine API. Of course, there should be the option to bypass the use of these features so that optimizations on the part of the developer are still possible.

Mesh data is distributed in a wide variety of formats, but most all of them can be classified into one of two types: vertex and skeletal. Vertex formats are characterized by the storage of each individual vertex as a point relative to the origin of the model. For animated models, the vertices are stored for each frame. Because these vertices are precalculated, they are easy to render and present greater opportunities for optimization. As a byproduct of storing all the vertices for every frame, vertex animated models generally require a lot of memory for highly detailed models. Skeletal models define a hierarchy of bones and a flexible skin that is calculated from the position of the bones. The only thing that needs to be stored for animated skeletal models is the position and rotation of each bone. Each vertex derives its position from a number of weights, which are simply bone-space (relative to the origin of the parent bone) offsets. A vertex might derive from weights from multiple bones. Take for example, the skin on the inside of your elbow. It can stretch when your arm is straightened out. Bones' positions are also given in bone-space, except for one bone which is world-space. Skeletal models require less storage of data and are more dynamic, because the skeleton can be modified within the code. However, for each frame, each bone must have a transformation matrix calculated for it, which must be multiplied by the transformation matrix of its parent. Recent advances in graphics hardware have made rendering skeletal models in real-time feasible, but they are still much slower to render than vertex models.

3 Development

3.1 Requirements and Development Plan

Different 3D engines have different feature sets, but some capabilities are common to nearly all. I hope to implement this basic feature set and, time allowing, make extensions to it.

- Load model data from file in commonly used formats
- Load and manage image data and GPU shaders
- Manage lights and material definitions
- Manage the texturing of polygons, including lightmaps and multitexturing
- Control the view frustum through a user accessible camera object

The engine is written in C, for purposes of speed and compatibility with C++. In order to maximize the usefulness of the code that I have written, I decided to modularize the main features of the engine to allow for greater flexibility in the future. My base-level library is a highly optimized math library, the next level is a material definition library, and the top level is the mesh library. The core engine is separate from all these, and all the parts are designed for library and platform independence. All parts maintain their own distinct namespaces except for the math library because of its omnipresence.

4 Engine Architecture

4.1 Engine Core

The very core of the engine consists of a mutable main engine loop. There are default functions for the different parts of the standard main application loop which the developer may override, leave in place, or add to by way of callback functions. The steps of the main loop are:

1. Parse window events (resizing, exiting)
2. Parse mouse and keyboard input
3. Pre-render (clear screen, reset transform matrices)
4. Render

Adding extra engine functionality would likely cause more steps to be added to the main loop. Physics should be done before the render loop, because the render loop is usually where collision detection and response are done. A post render method might also be useful for special effects which require multiple passes or rendering to the backbuffer, such as reflections.

4.2 Math Routines

3D graphics have a specialized set of mathematics API requirements that are not handled by C standard libraries. Because of this, I decided to write an optimized math library that handles the structures commonly used in 3D graphics applications. Vectors in 2, 3, and 4 dimensions, quaternions, planes, and 3x3 and 4x4 square matrices are all included. Only the most common algorithms used in 3D graphics are included in this library, for example a method to build a transformation matrix from a rotation and displacement. Some common optimizations are included in the math library, such as a faster inverse square root and floating point absolute value, and all functions are inline-enabled, allowing the code from the functions to be substituted in place of the function call during compilation.

4.3 Material Defintions

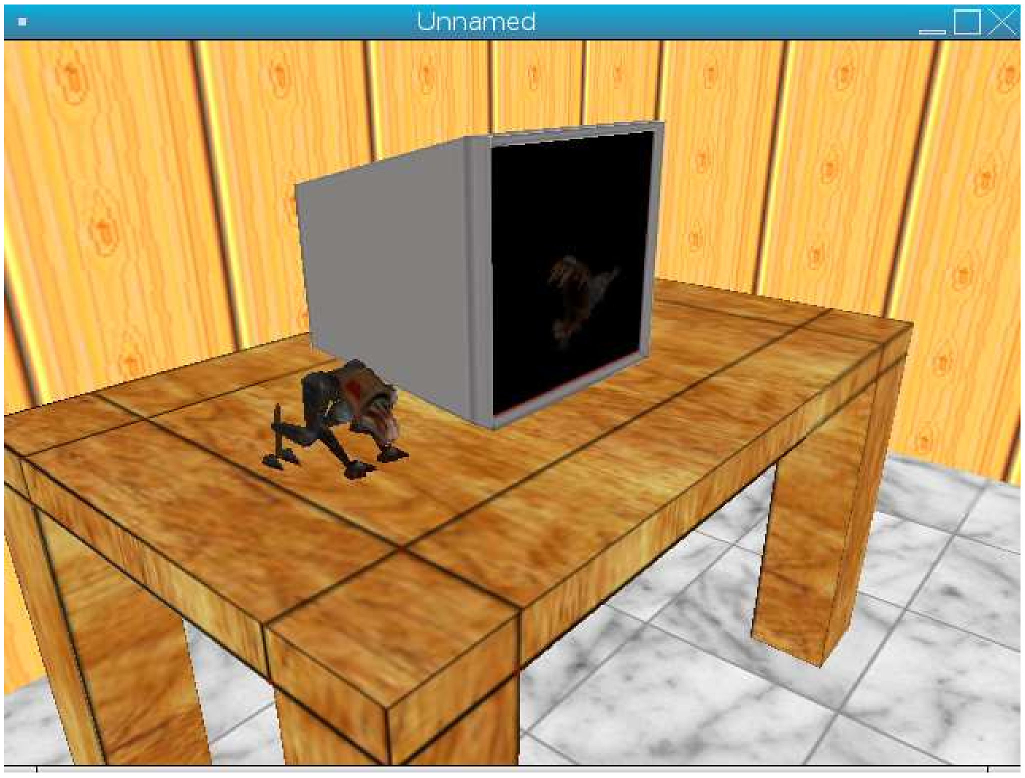

Figure 1: Rendering a model on a screen using post-display back buffer rendering

A separate library, liballoy, handles anything relating to the display of a surface. Image loading functions are included to facilitate texture generation. The engine also supports GPU shader, which are used to modify the fixed-function hardware pipeline on graphics cards for adding other effects, such as more complicated lighting. Also included is the ability to specify a pattern for blending multiple textures, know as multitexturing, and to modify the way the polygon will respond to dynamic lighting, for example, reflectivity. Materials need only be loaded once, and can be embedded in other files, such as meshes. When the bind method is called, all the properties of the material are applied and all polygons rendered afterwards will exhibit those properties until they are disabled with the unbind function. Memory allocation is handled entirely by the library for maximum efficiency.

4.4 Mesh Functions

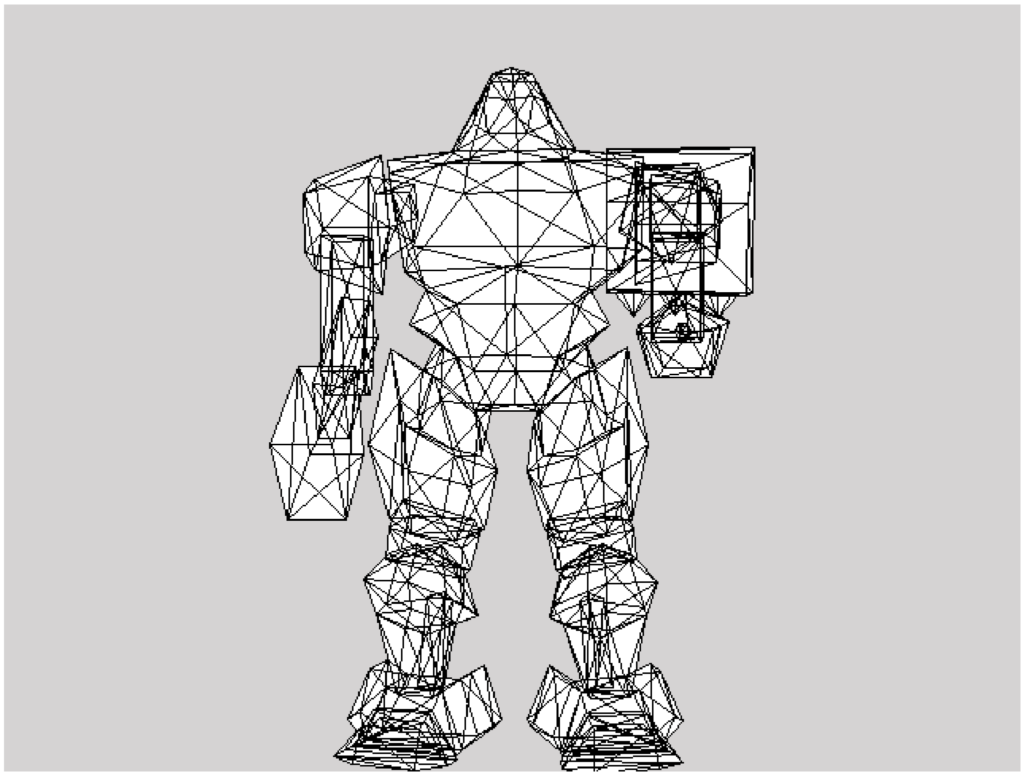

Figure 2: Wireframe display of meshes loaded from file in MD2 format

Most 3D applications need to load some kind of mesh from a file. There are several different file formats, and writing a parser can be a difficult task. For this reason, I have written a library to handle loading mesh data from a few common file formats, specifically the MD2, MD3, MD5 file formats. These meshes represent various types of meshes: including non-animated meshes, vertex models, and skeletal models with skinning. All meshes are converted to a common internal mesh structure: the half-edge structure. In this model, the mesh stores data on triangle faces, edges, and vertices. Each edge stores its starting vertex, the next edge, and a corresponding edge in the opposite direction on an adjacent triangle. Each face stores simply the first edge, and the vertices are completely separate, allowing them to be stored in high-performance graphics hardware memory in allocated regions knows as Vertex Buffer Objects. This internal representation uses the native animation type of the original mesh, so skeletal models are not converted to vertex and vice versa. The advantage of using the half-edge format is that it is easily modifiable in code, simplifying level-of-detail operations, and also has characteristics useful for collision detection. It is generally slower and less memory-efficient than other formats, but it is generally still acceptable, especially when optimized.

4.5 Animation Types

As stated earlier, the engine supports both vertex and skeletal animation. Vertex animation implies storing the position of each vertex (in world-space) for every frame of an animation. These models are inflexible in the sense that it is difficult to modify the animation at runtime. Skeletal animation takes advantage of the predictable motion of certain meshes. A virtual skeleton is created, a hierarchy of bones, and a flexible skin of vertices is created around it. The vertices deform predictably, and so the only information that needs to be stored for each frame is the position and orientation of each bone relative to its parent. The compromise is that matrix multiplications must be used to transform each vertex. In some cases, vertices are linked to multiple bones, and even more calculations must be made. Recent hardware developments have made real-time rendering of skeletal models possible. As an added bonus, programmatic control of the parameters of the animation (the skeleton's configuration) is possible, and used to create unique and smooth animation in recent games.

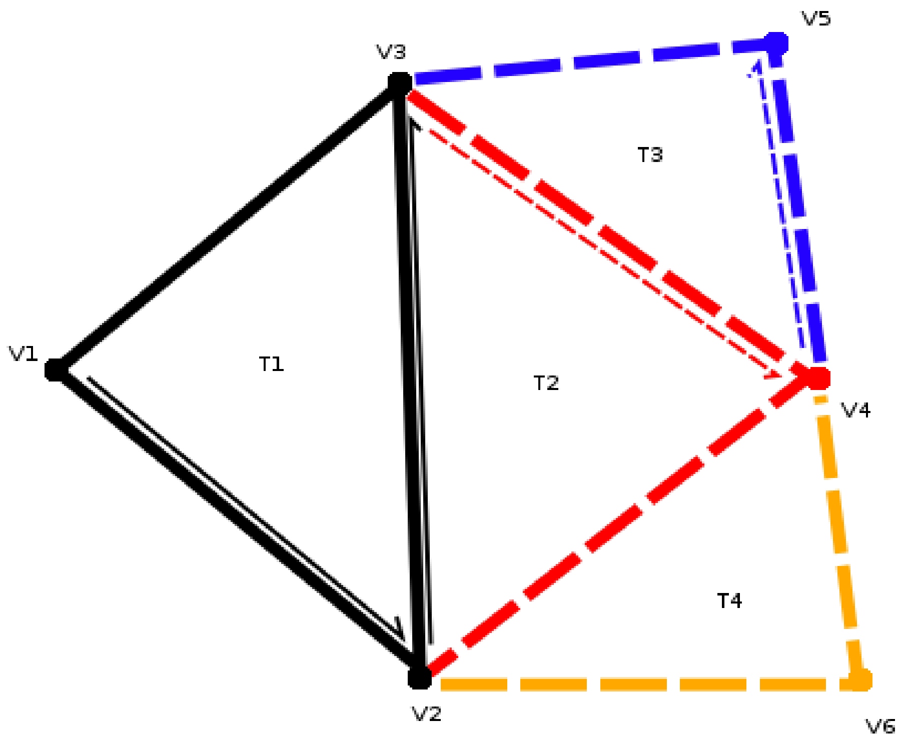

Figure 3: Exploiting triangle connectivity

4.6 Mesh Optimizations

The simplest way to render a mesh is to iterate over the constituent triangles and pass all three vertices to the graphics card for rendering. Unfortunately, the speed of transmitting the vertices from RAM to the graphics card is a limiting factor for frame rate. In order to speed up the rendering process, the connectivity of the triangles in the mesh can be exploited. After the first triangle has been defined with all three vertices, the triangle connected to the first by the edge which was defined by the last two vertices can be rendered using only one additional vertex. In the figure, T1 would be defined by V1, V2, and V3. The graphics card will remember that the last edge was defined by V2 and V3. When vertex V4 is sent to the graphics card, a triangle will be formed with the edges V2 to V3, V2 to V4, and V3 to V4 (the stored edge). Even though the edge V2 to V4 is not explicitly given, it is assumed, thereby reducing the amount of data that needs to be sent over the system bus. In order to draw T3, the next vertex would be V5. A problem arises when trying to build a strip that renders T1, T2, and T4. If the vertices of T1 are passed to the graphics card in the same order, then the only edge that can be used to render T2 will be on the wrong side of the triangle to allow for rendering T4. Instead, after submitting vertex V3, V2 is submitted again. This reverses the order of the last edge, so that T2 can be rendered by passing V4, and the stored edge will be from V2 to V4, which means T4 can be rendered by submitting V6. This operation does require an extra vertex to be sent over the bus, but this trick is quicker than beginning a new triangle strip, which would require 3 vertices to be sent. In this example, it would be trivial to detect the immediate swap and fix the order of the first vertices so that the swap would be avoided, but in longer strips, swaps are inevitable.

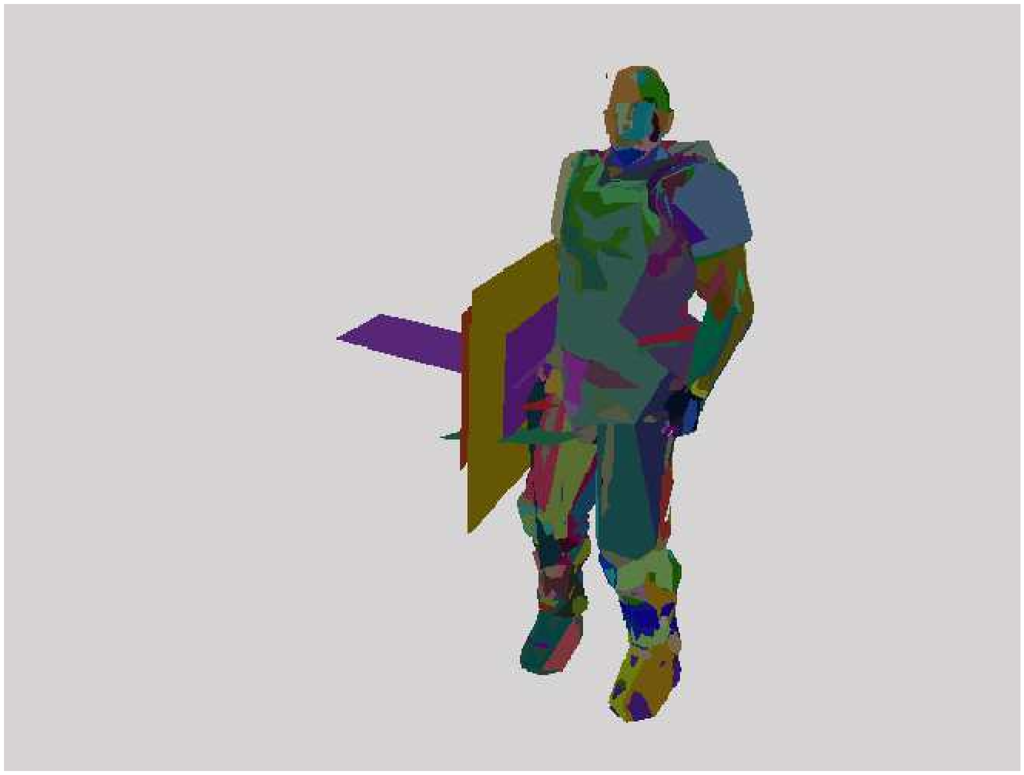

Figure 4: Rendering a skeletal mesh using triangle strips

The heuristic I used when generating triangle strips from mesh data is known as LNLS - least neighbors, least swaps. For any given triangle after the first in a strip, there are at most two choices to continue to. If there are indeed two neighboring triangles, the first tiebreaker is number of neighbors. If both neighboring triangles have the same number of neighbors, then the one which does not require a swap is chosen. The quality of the strips generated is dependent on the topology of the mesh.

4.7 Particle Engine

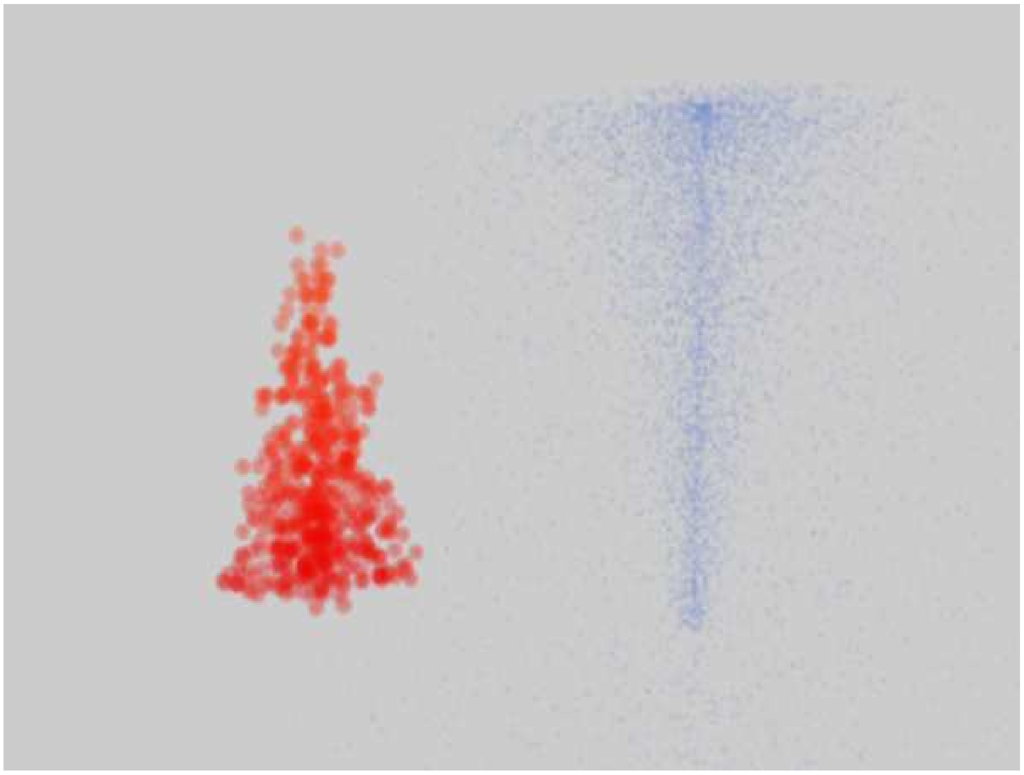

Figure 5: Simulating water and fire with particle effects

A computationally inexpensive method for special effects involving lots of objects (the water droplets in a fountain, for example) is a particle engine. My particle engine consists of two objects: the emitter and the particles themselves. The emitter is responsible for keeping track of all the particles in a doubly-linked list, performing kinematics equations on them, and rendering them. The emitter also stores the texture and size of the particles. The particles store their position, velocity, acceleration, and color. There is a method which is called every timestep which decides whether or not to spawn some number of particles (based on a spawn rate parameter), and gives them their initial position, acceleration, and velocity. The velocity is randomly rotated from the specified velocity by some angle which determines the arc. The developer may override the default spawn or kinematics functionality by way of callbacks. The particles are rendered using a billboarding technique, which ensures that they are always rotated to face the camera, which gives the illusion of volume, even though each particle consists only of a square.

4.8 Debugging Tools

I have developed a small suite of debugging tools to be used in testing my program. It includes a logging tool, which is useful for keeping track of problems that are not fatal, and do not need to be reported to the user immediately. The log file keeps track of events such as loading models, texutres, and scenes. A timestamp is prefixed to each entry, and the log can be written to by the engine and the application using it. The most useful debugging tool is a profiler, which allows the developer to time the execution speed of a block of code and view a report detailing where most time was spent. Tools such as gprof already exist for the purpose of timing the speed of functions, but my more tightly integrated profiler allows the developer to test the speed of separate blocks of code within a function. Being able to immediately see where performance bottlenecks are speeds up the vital optimization process.

5 Results and Discussion

In addition to meeting the majority of the requirements I set down for my 3D engine, I would deem my project a success based on what I've learned in terms of API design and my success in implementing them in my engine. I have created an engine which legitimately makes 3D programming simple enough for someone who does not know OpenGL, and I have learned principles of C and C++ code design and organization that I will benefit from later on. I have created libraries that are useful in their own right, without any context relating to the core engine, and I have made steps toward platform- and library-independence.

References

[1] Botsch, Mario et al., ”OpenMesh - a generic and efficient polygon mesh data structure”
http://graphics.ethz.ch/ mbotsch/publications/openmesh.pdf (January 23, 2007)

[2] Clark, James H, ”Hierarchical Geometric Models for Visible Surface Algorithms”, Association for Computing Machinery, Inc. 1976. Presented at SIGGRAPH 1976.

[3] Shoemake, Ken, ”Quaternions”, University of Pennsylvania.

[4] Schmalstieg, Dieter and Schaufler, Gernot, ”Smooth Levels of Detail”, Vienna University of Technology, Austria.

[5] Fuchs, Henry et al., ”On visible surface generation by a priori tree structures”, Association for Computing Machinery, Inc. 1980.

[6] Lin, Ming C. et al., ”Collision Detection: Algorithms and Applications”, University of North Carolina.
Http://www.cs.unc.edu/ geom/collide.html

[7] Gottschalk, Stefan, et al., ”OBBTree: A Hierarchical Structure for Rapid Interference Detection”, University of North Carolina.
http://www.cs.unc.edu/ geom/OBB/OBBT.html

[8] Vanecek, Petr. ”Comparison of Stripification Techniques”, Center of Computer Graphics and Data Visualization, Pilsen, Czech Rep.
http://www.cescg.org/CESCG-2002/PVanecek/paper.pdf

[9] Vanecek, Petr. ”Triangle Strips for Fast Rendering”, University of West Bohemia in Pilsen.
http://www.kiv.zcu.cz/publications/2004/tr-2004-05.pdf

[10] Kornmann, David. ”Fast and Simple Triangle Strip Generation”, Varian Medical Systems Finland.
http://www.dlc.fi/ dkpa/strip/newstrip.pdf

[11] Nuydens, Tom. ”Triangle strip generation”, Delphi3D.
http://www.delphi3d.net/articles/viewarticle.php?article=tristrips.htm