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

Kevin Kassing

January 24, 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. Plenty of toolkits exist for the purpose of simplifying the development of widget-based 2D graphics applications. 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.

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

0.1 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.

0.2 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.

0.3 Development

0.3.1 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
- Provide collision detection methods
- 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++. I have no requirement that all the parts of the engine be a part of a single library; one the contrary, many of the methods that could be refactored into a separate library without significantly reducing speed due to the lack of integration with the other parts of the engine are being separated into additional libraries. The mesh loading functions are part of an independent library that has its own namespace. All of the libraries are compiled into archives for static linking. The core engine is designed to be library-independent for the input and graphics APIs, and as a byproduct, platform-independent as well.

0.3.2 Engine Architecture

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.

Texture Storage

Textures, which are used in nearly all 3D applications, are stored by the engine. Images are first loaded to an image structure, which is passed to a method which converts that image into some type of texture, most commonly a simple 2D texture. The texture structures are stored in a linked list for automated cleanup, and may be retrieved by name, although they are commonly stored in the application by the pointer that is returned from the engine. To load a texture in a common file format, the only requirement is to know the file name. Binding a texture, which instructs OpenGL to use that

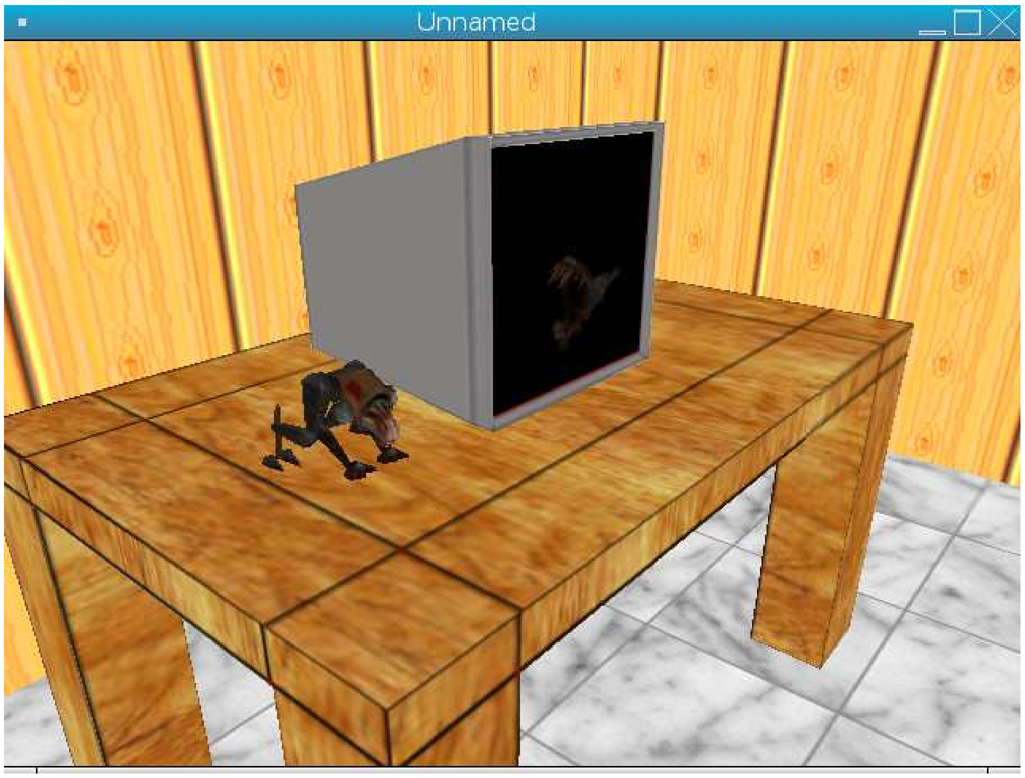

Figure 1: Rendering a model on a screen using post-display back buffer rendering

texture when shading polygons, is also done automatically through a single function for all types of textures. GPU shaders, which are not textures but are often used in a similar fashion, are also stored in texture objects.

Math Routines

3D graphics have a specialized set of mathematics API requirements that are not handled by standard libraries. My engine calls another library, also written by me, for all 3D math needs. Vectors in 2, 3, and 4 dimensions, quaternions, planes, and 3x3 and 4x4 square matrices are all included. The algorithms used in 3D graphics are not included in this library; there is no method there to build a transformation matrix from a rotation, scale, and displacement. All of those algorithms are handled in the core engine. Some common optimizations are included in the math library, such as a faster inverse square root and floating point absolute value.

Mesh Functions

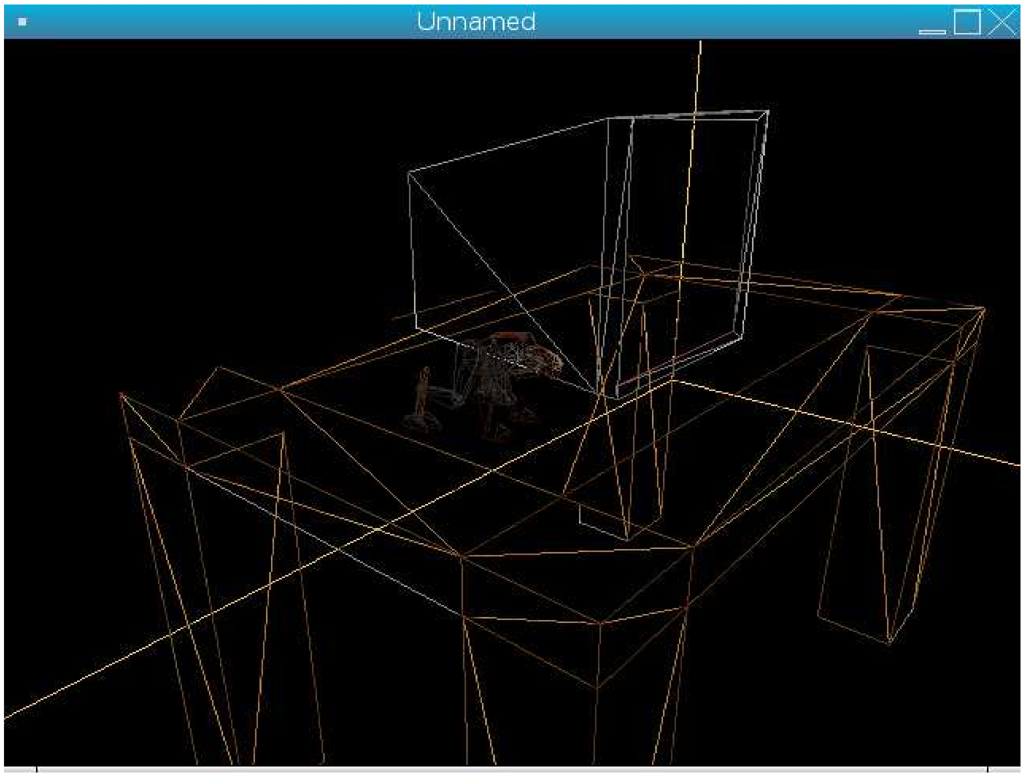

Figure 2: Wireframe display of meshes loaded from file in MD2 format

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 and ASE file formats. These meshes represent various types of meshes: including static meshes, vertex animation, and skeletal animation with skinning. This same library will also be able to convert loaded meshes to a unified mesh structure based on the half-edge mesh data structure.

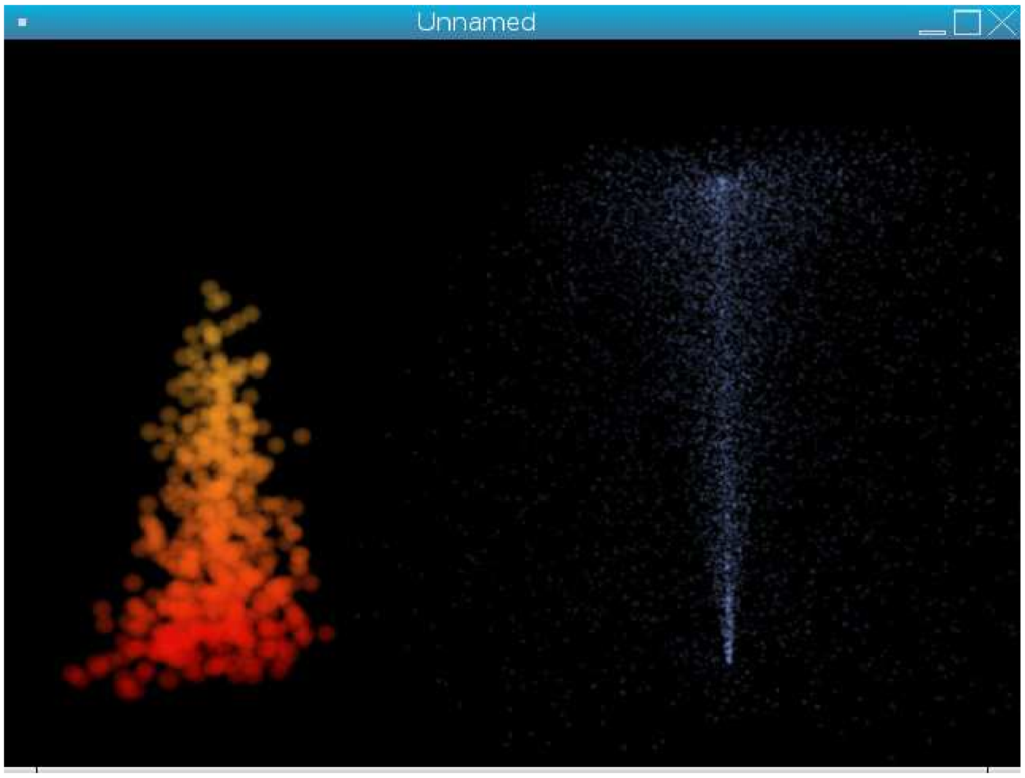

Figure 3: Simulating water and fire with particle effects

Particle Engine

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.

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.

0.4 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.

0.5 Next Steps

The next steps are clear: create a mesh format optimized for the internal design of the engine, expand the texture system, perhaps into a separate library, to manage material definitions, shaders, textures, with an interface that allows simple flat-texturing as well as multitexturing and multi-pass effects. My current solution, while speed-efficient, is not as aesthetically pleasing as I would like. Focusing on a single mesh format would also allow me to develop a mesh API without worrying about having to support data from several different model paradigms.

Bibliography

[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