The Asteria 3D Game Engine

Architecture Document

Samuel C. Payson Akanksha Vyas

December 17, 2011

Contents

Part I
An Introduction To Asteria

1 Introduction

1.1 What is Asteria?

Asteria is a 3D game engine being developed by members of the Clarkson Open Source Institute at Clarkson University in Potsdam, NY. The project aims to be a high-performance, open-source game engine with a clean and comprehensible design.

1.2 About This Document

Asteria seeks to remedy a common flaw in many open-source projects: a lack of good high-level documentation. This document opens with a description of the project's design at the level of subsystem interactions. It then presents a series of sections which provide enough information about the implementation of each subsystem for a new developer to get their barings in the code.

Every non-obvious abstraction is described in detail, and justifications for each design decision are made along the way. The goal is to provide sufficient information about the design of the project to allow a developer who has never seen the code to make significant contributions *without having to reverse engineer the entire system.*

Having the source be open is important, but it becomes much more useful if you have an understanding of the architect's approach to constructing it.

1.2.1 Document Maturity

It is worth mentioning that this document is just beginning its life as a technical document, and it may change drastically before Asteria is complete. This is our first attempt at creating a comprehensible guide to the code for contributors, and it may still have a long way to go.

1.3 Licencing Information

All code for Asteria is released under the GPLv3.

Part II
High-Level Design

2 Software Design

2.1 Modularizing the Design

Asteria logically partitions the code in several ways, the practice ordinarily referred to as *modularization*.

2.1.1 Levels of Modularity

Modularity is maintained at three distinct levels of granularity

1. Classes
2. Subsystems
3. Modules

Classes The code is naturally divided into classes in the typical object-oriented fashion. This is the only level of granularity that is supported at the language level, all other levels are conceptually maintained by the developers.

Subsystems Classes are partitioned logically into subsystems. Examples of these subsystems are the GL abstraction (code designed to make interaction with OpenGL more concise), and the Animation subsystem (code that knows how to render 3D models with animation). In essence, each subsystem contains code for solving problems from a different domain.

Modules Modules contain broad categories of related code. For example, there is a *Graphics/Renderer* module that encompasses all functionality related to OpenGL, 3D models and/or Scene Graph concerns. Often times the distinction is that subsystems within the same module need to know about eachother's implementation at some level to be implemented efficiently. Subsystems in seperate modules should be able to treat each other as black boxes.

2.1.2 A Look at The Modules

Figure 1 shows all of the modules that exist (or will exist) in Asteria. Modules colored blue (■) are under way, nodes colored purple (■) are external libraries (not developed by the Asteria team), and modules colored gray with dotted edges (▫) are not yet under active development.

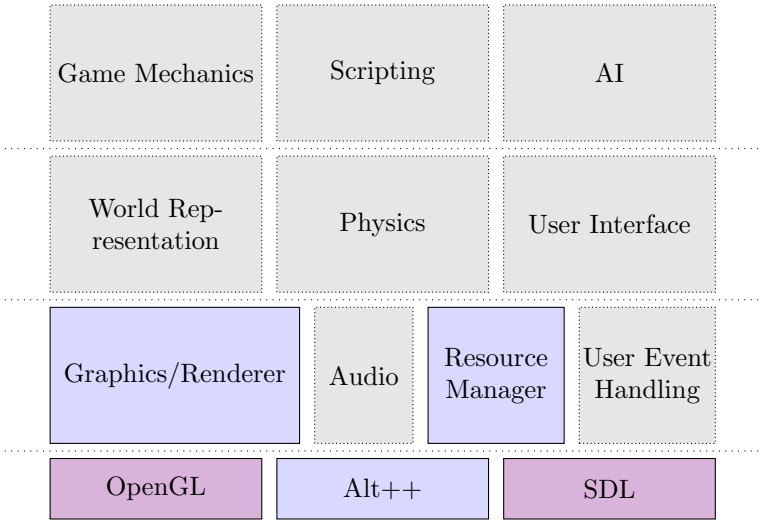

Figure 1: The Asteria module stack.

The reader will observe that the figure neatly divides the set of modules into layers. This is no coincidence, the modules listed are ordered from most general (bottom) to most specific (top).

For example, OpenGL, SDL and Alt++ could be used to implement almost any kind of game. Once we get around to implementing Game Mechanics, Scripting and AI, however, they will have to be taylored to a very specific game model in order to be of any use.

Asteria will attempt to provide solid implementations of the bottom two layers (excluding the external libraries, of course). The third layer will be implemented with as much generality, configurability, and extensibility as possible. The fourth layer, however, will only be provided in the form of an example implementation. The hope is that as time goes on more and more games based on Asteria will provide more diverse implementations of this specialized code.

2.1.3 Module Interactions

Every module is designed so that it only has knowledge of modules at lower layers. Any interaction between two modules on the same layer must be instrumented by modules at a higher layer.

For example, there are natural interactions that would occur between the game world and the physics engine, but these interactions should be brought about by the Game Mechanics or Scripting modules.

2.1.4 Selective Use of Asteria

The layers are designed as they are so that a developer hoping to use asteria can choose which portions of the code they want to build on. One developer might want to only utilize the first two layers, in order to reduce the amount of boiler-plate code they have to write. Another developer may only want to code the specifics of their own game, not worrying about the implementation of a physics engine or UI.

This flexibility is brought about by ensuring that a module only depends on other modules that are in layers strictly below it. In this way we can ensure that it is safe to remove any one module without effecting anything else in its layer or below.

Part III
The Subsystems

3 The Animation Subsystem

The animation subsystem parses a text-based description of a animated 3D model and draws it to the screen. It is currently implemented for the md5 file format popularized by Doom 3 (completely unrelated to the md5 cryptographic hash). Consistent with the md5 file format, it performs skeletal animation using vertex skinning. The animation subsystem has three components: the parser, the draw function and the shaders.

The parser is responsible for parsing a md5 file and storing its description. The draw function is responsible for calculating the exact coordinates of each joint in the frame. The shaders use vertex skinning to actually draw the object.

3.1 md5 File Format

Animation of objects in md5 files is represented by the position and orientation of every joint of the object at regular intervals. Different frame characterrize thses different intervals. The file contains a base frame, describing one such the orientation and position for each joint. All successive frames represent the joints as offsets from the base frame. The file also contains a hirarchy of dependencies for each joint, the frame rate of the animation, and the total number of frames, joints and animated components. More information on the file format can be found at *http://tfc.duke.free.fr/coding/md5-specs-en.html*.

For the current implementation, we are assuming that all objects are created in Blender. Blender is a free and open source software suit which is particularly useful for creating 3D models. It is a common substitute for industry standerd Maya. More information on Blender can be found at *http://www.blender.org/*.

3.2 The Parser

As the name suggests, this component parses the animation file. It uses Flex and Bison, tools for generating scanners and parsers efficiently. More information on these can be found at *http://dinosaur.compilertools.net/*. From the offsets and the base frame values, the parser calculates the position and orientation for each joint in every frame. From the hierarchy description, it calculates the parent of each joint.

md5 files use vectors in $\mathbb{R}^3$ to represent position and quaternions to represent orientation. Quaternions are an algebraic structure that are found to be very nifty for computer graphics. A description of the structure can be found in the appendix, but for simplicity we can just look at them as vectors in $\mathbb{R}^4$.

The animation subsystem populates the `md5AnimData` struct that can be found in */include/md5Structures.h*. Each joint has a position vector, an orientation quaternion and a parent. If it is a root joint the parent is set to -1.

3.3 The Draw Function

The draw function has two parts. The first part, given a time uses the framerate to calculate which two frames the animation is currently between, and how far between them it is. With this information it interpolates between the two frames to calculate the position and orientation of each joint at that instant. A more detailed description of interpolation techniques will follow.

Every joint will be affected by any change in the orientation of its parent. The draw function need to change the position and orientation of a joint with respect to the orientation of its parent. The second part takes care of this. Assuming a topologically sorted list of joints (as Blender guarantees), it starts from the top and rotates the position and orientation of every joint by the orientation of its parent. Now it is ready draw these points. It sends the position of each joint to the shaders, which then take care of this.

3.3.1 Interpolation Techniques

Interpolation techniques help identify a smooth path between the two frames. Given two vectors or quaternions they attempt to find a third vector or quaternion on this path respectively. The draw function uses linear interpolation (LERP) for the position vectors and a modified version of normalized linear interpolation (NLERP) for rotation quaternions. More information on these can be found in the appendix.

3.4 Shaders

When it comes time to compute the position of each individual vertex of the mesh, we offload the task to the GPU. We tell the vertex shader how each joint effects the final position of a given vertex, and then the GPU grabs the most recent joint data and applies it's transformations to the vertex. If multiple joints affect a single vertex, they are counted according to their weights. This is how vertex skinning is achieved.

There is significant gain from computing these values on the GPU instead of the CPU. For one, the task is what is known as *embarassingly parallel*, meaning that every step of it (i.e. each vertex' final position) can be computed without regard for the other computations being performed. GPUs typically have highly parallel architectures, so we can fully exploit this opportunity for parallelization.

We also save time because we do not need to send data over the system bus repeatedly. It is sufficient to send vertex-skinning data to the GPU once. After that the only data that needs to be send to the GPU is skeleton data, ussually thousands of times smaller than the vertex-skinning data.

4 The Resource Manager

The Resource Manager provides an abstraction of the file system with several attractive advantages. The foremost among these are

- The ability to locate files without any OS overhead.
- A uniform interface for all sorts of different files (i.e. compressed, encrypted, etc...)
- A way to store metadata seemlesly alongside files.

4.1 Implementation

The resource manager is implemented as a special archived file format. It is constructed like a small file system of its own, using a scheme similar to the original Unix File System.

The archive is divided into files, the first of which contains a listing, which we refer to as the *manifest*, of all of the files contained in the archive and their offsets from the beginning of the archive.

Every file starts with a meta-block that contains some important information about the block. For example if/how it is compressed, the size of the file, or user-supplied data. After this block, each file is simply a linear sequence of blocks (from the reader's perspective) containing the original file's data.

4.1.1 The Archive on Disk

While the user sees the file as linear, it is represented on disk as an inductively defined tree structure. We define two kinds of blocks, there are *data blocks* and *interior blocks*.

Data Blocks Data blocks are just regular blocks of a fixed size. These are where actual file data is stored (whether it be the meta-block or a normal file block).

Interior Blocks Interior blocks allow us to define files of arbitrary size. An interior block simply contains a set of pointers to other blocks. We specify an attribute of blocks called *depth*, with the following constraints:

1. A block of depth 0 is a data block.
2. A block of depth > 0 is an interior block.
3. An interior block with depth n is made up of pointer to blocks of depth $n - 1$.

When storing a file with n data blocks, this definition creates for us a tree of depth $\log_k n$ where k is the number of pointers that an interior block can hold. This means that given an offset into a file we can locate its block in time $\mathcal{O}(\log_k n)$. In essence it is a binary tree who's keys are file offsets.

4.1.2 Where Do We Gain Efficiency?

When the game first loads the archive, it reads in the name of every file in the archive and stores its start-offset in a hash table. This way locating a file requires only a fast lookup with no system calls or disk i/o. Since games often open and read many thousands of small files, this is a worthwhile optimization.

Also, the scheme described above for the organization of blocks on the disk allows for more cost-effective use of space. Files can expand, be deleted, or be inserted without rebuilding the entire

archive. For a game which could have frequent updates to content (especially during development) this is desireable.

A The Quaternion Algebraic Structure

Quaternions were invented by Irish mathematician Sir William Rowan Hamilton in 1843. They have been found to be useful in many fields of mathematics and computer science, including 3D graphics. Quaternions form a number system that extends the complex numbers into four dimensions. A quaternion is of the form:

$$a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k},$$

$$\text{where } a, b, c, d \in \mathbb{R} \text{ and } \mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

$$\text{also, } \mathbf{ij} = \mathbf{k} = -\mathbf{ji}, \mathbf{jk} = \mathbf{i} = -\mathbf{kj}, \mathbf{ki} = \mathbf{j} = -\mathbf{ik}$$

They common notation for q is $[s, \mathbf{v}]$, where $s \in \mathbb{R}$, $\mathbf{v} \in \mathbb{R}^3$ and $\mathbf{i}, \mathbf{j}, \mathbf{k}$ are imaginary numbers.

B More About Interpolation

Now lets talk about interpolation techniques. Three common interpolation techniques are discussed.

1. Linear Interpolation (LERP) is the simplest interpolation technique. The idea is to draw a line between two points and find a third point at the desired distance between them. Though this is useful for interpolating between position vectors, it is graphically inaccurate for quaternions. Let v_1, v_2, t where v_1, v_2 are the vectors describing the position you want to interpolate between, and t is the distance between them. Equation 1 describes the math.

$$(1 - t)q_1 + tq_2 \tag{1}$$

2. Spherical linear interpolation (SLERP) was the next technique we looked at. It is currently the most popular interpolation technique. SLERP calculates the point on the arc of a circle at the desired distance between two quaternions. Figure 2 gives us an idea of what this looks like.

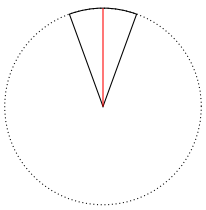

Figure 2: Quaternions rotating around a pivot

Let q_1, q_2, t, where q_1, q_2 are the quaternions you want to interpolate between and t is the distance between them. Equation 2 describes the math.

$$q_1(q_1 * q_2)^t \tag{2}$$

A more implementation-friendly version of the math can be found at *vMath.c*.

The problem with SLERP is that its computationally expensive.

3. Normalized linear interpolation (NLERP) is a fairly new interpolations technique that tries to balance the positives and negatives LERP and SLERP. The basic idea is to do the LERP operation and then normalize the result.

   Let q_1, q_2, t, where q_1, q_2 are the quaternions you want to interpolate between and t is the distance between them. Equation 3 describes the math.

$$norm((1-t)q_1 + tq_2) \tag{3}$$

   Intuitively this seems like it would be much faster and graphically correct. However, we encountered a problem with it. We realized that you need ensure that the cosine of the angles between the two quaternions is positive. On encountering a negative value, you have to negate one of the quaternions in order for it to interpolate along the correct arc. This increases that computation significantly, making it only marginally faster than SLERP, if at all. We are currently looking for a more efficient way to do this check.