The Fraction Engine

An analysis of the Fraction Game Engine

Oliver Charles (depth.of.field@gmail.com)

1 Background

1.1 About the Fraction Engine

The Fraction Engine is my own engine. I have created it from the ground up to educate myself about the techniques used with creating a game engine. The engine is still progressing; I don't plan to rewrite it for as long as possible. Many of the techniques I have learnt during the creation of this engine have proved incredibly valuable – so in the respect of my goal in the engine, I feel I have succeed already.

The engine has been designed to be "generic" – that is, usable for most different types of games. This generic concept meant that I would have to put time into different techniques, weighing-up the pros against the cons and choosing what I feel best suits the engine. I also had to learn about optimisations, things such as spatial-partitioning, render clusters and so on.

Once I feel it is ready, I plan to release it as open-source to the community, a gift to everyone for all the help I have received.

Anyway, that's generally the background of the engine.

1.2 About the author

I'm only a 15 year old with a huge passion for programming and computer technology. I have never taken or attended anything that is to do with programming generally due to the lack of options at my school.

Everything I know programming wise has been taught to me by either people over the internet, experimenting myself, or reading a book and trying it myself.

I hope to get into the games industry, so I designed this engine to help me get to grasp not only with game engine technology, but also creating a strong object-orientated framework.

2 Engine Structure

2.1 Overview

Establishing a good, strong engine was straight away top of my to-do list. I spent a long time learning about design-patterns before I began work on the engine.

I will use this section to describe some of the things I believe stand out well in the engine.

2.2 The Entity I/O System

This is something, I am quite proud of and have only recently implanted. I wanted a way for entities to interact with each other, but without this being hard coded or scripted. I took the basic idea from the Source Engine entity system. An entity can have inputs, and outputs. Outputs are 'fired' when something happens. Inputs are linked into other entity's outputs so they can work independently.

The diagram below (Figure 2.2.1) demonstrates this system, with a basic switch that opens a door.

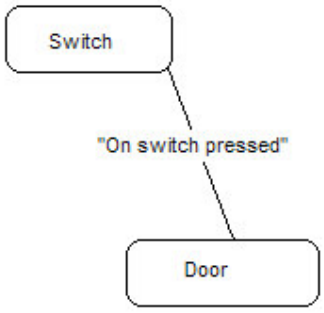


Figure 2.2.1 Entity IO System.

In this hypothetical example, the door has one of its inputs attached to the switch's "On switch pressed" output. The door will have attached a custom

method to this output, so that when it is fired, the door can open.

The code for this is really simple, which is what I think is the beauty of it. It uses Reflection, which was a completely new topic when I was overcoming the entity IO problem.

```
Switch openDoor = new Switch();

Door doorToOpen = new Door();

EstablishConnection(openDoor,
  doorToOpen, "onOpen","Open");
```

Here we see how simple it is to create the connection. "EstablishConnection" takes the parameters source (who will create the output), the destination (who will receive the output), the output name (simply defined as an event in the source) and the input name (simply defined as a method in the destination instance).

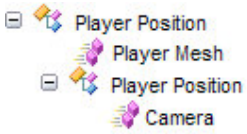
2 Rendering Features

2.1 Overview

The visual aspect of games used to, when I first began creating the engine, be the most appealing to me. So naturally, as I began the engine all sorts of ideas came into my head for 'cool' graphics features. Those that have been implanted are described in more detail below, and the rest still remain on my to-do list waiting to be added.

2.2 Powerful Scene Graph

A scene graph was something that made me "ummm…ahhh" about for quite a while. The Fraction Engine now has a stable scene graph were users are required to link meshes into the graph. This is what a basic third person camera could look like in the engine…



The scene graph nodes link well with other systems allowing for quad-trees, octrees and other spatial partitioning techniques.

It also means that any object can be assigned to a scene graph node, so they all share a similar matrix (and position in the world).

One final thing is the picture demonstrating the scene graph layout is actually taken from the engine debugger. The debugger allows the developer to manipulate the scene graph at run time.

2.3 Shaders

The Fraction Engine supports Microsoft 'Effect' files. These files are shaders written in a high-level C like language. It is good because it allows programmers to rapidly create shaders in a familiar language.

The Fraction Engine handles shaders incredibly well. All the user needs to do is load a shader, apply it to a surface's material and… that's it! The engine will automatically set variables that follow the HLSL semantics. The world, view, projection, world-view-projection-inverse-transposed matrices and more are all automatically set which was good.

The way that the engine does this is by looping through the variables in a shader, on load, and gets the handles to the variables. Then when the shader is enabled the shader will set these handles and if a variable with a certain semantic does not exist in the file, then it's not set.

The shader's will be able to be set as the render-cluster mode. This is useful for games that are highly shader based (games that replace the fixed-function pipeline with there own, maybe per pixel, shader engine – for example) making it very efficient swell.

2.4 EASY TO USE AND EXTENDABLE LIGHTING SYSTEM

The lighting system currently uses the fixed function pipeline, but this is so transparent, it could easily be transformed into shader stuff (which I plan on doing). The lights all inherit from a base light interface so that each light class only has the properties it needs.

It's only a very extendable system, like the volumetric light is really an extension of point light – but it creates a mesh as well. The volumetric mesh is just 4 quads that fade to an alpha of 0 (per vertex alpha) – so nothing fancy.