Real-time 3D Graphics in Haskell

Eric Thong (ethong@cs.stanford.edu)

Background

OpenGL is a low-level API designed for drawing 2D or 3D graphics. Since OpenGL is just a drawing library, it does not include specifications for things like audio, windowing, nor a scenegraph. It is up to the client to keep track of and organize objects in the scene.

OpenGL is built around the idea of a “rendering pipeline” in which vertices stored in an array in system memory are shipped over to the GPU, transformed in the vertex shader, further transformed in the geometry shader, assembled into “primitives” like lines or triangles, rasterized into pixel sized fragments, transformed in the fragment shader, then finally output to a buffer and/or screen. OpenGL provides functions that manipulate the pipeline at each stage. For example, the client can supply OpenGL with a depth-test function so OpenGL doesn’t draw pixels that are behind other pixels.

Originally, OpenGL described a fixed-function rendering pipeline, but over time, more programmable functionality has been added. Shaders are programs that run on the GPU. OpenGL’s shading language, GLSL, is used to create vertex shaders, geometry shaders, and fragment shaders, which are programs that operate on their respective data. A fragment shader, for example, can take a fragment’s normal vector, the location of a directional light, and light the fragment appropriately. GLSL programs have C-like syntax and are sent to the graphics driver as plain text. OpenGL provides functions that command the graphics driver to compile, link, and use the shader.

Why Haskell

Thanks to Haskell’s strong type system, Haskell programs that compile, tend to “just work”. OpenGL programming in languages like C++ can be error prone because the API does not enforce type safety. For example, the in plain OpenGL, the client has to specify how a vertex shader should read the vertex buffer. If a wrong stride length is specified, this can lead to subtle memory bugs. In Haskell, it’s possible to eliminate this problem by giving the elements in the vertex buffer a type and only allowing certain operations on that type in the vertex shader.

Plain OpenGL in C++ can seem verbose and clunky. In order to do basic things, it is often necessary to allocate temporary variables and buffers. For example, to find out if a shader has compiled correctly, the client has to allocate a temporary status integer to pass into `glGetShaderiv`. If upon reading the status integer, the compilation has failed, then the client

has to allocate another temporary buffer and pass it into `glGetShaderInfoLog` to get the compilation error.

The Gpipe library, which was used in this project, implements strong type safety made possible by Haskell. Vertex buffers in the Render monad are strongly typed so that it is unnecessary to specify how to access vertex data in the Shader monad. In plain OpenGL, it's necessary to describe how the variables from the vertex shader are connected to the fragment shader. With Gpipe, that is free - the vertex shader and fragment shader live together in the shader monad and data from the vertex shader is seamlessly passed to the fragment shader.

In order to guarantee type safety in the shaders, Gpipe implements its own shader DSL. Shader programs written in the shader monad become transformed into plain text GLSL behind the scenes. It's possible to examine the GLSL by using an OpenGL debugger. Shaders written in the shader monad are type checked by GHC at compile time and are much less likely to fail when the generated GLSL is compiled again by the graphics driver.

Design and Implementation

The core of real-time 3D graphics is the render loop where each iteration produces a frame. For this project, the render loop takes various state as input (keyboard/mouse state, camera state, physics state, and shape state), runs the simulations, then outputs the various states. For a given sub-simulation, multiple input states can be used to produce multiple output states that can be fed to another sub-simulation. I'll describe the various simulations next.

The input simulation takes the previous input state, and the current input and produces a new input state.

The camera simulation takes the previous camera state, previous window state, current window state, previous input state, and current input state and produces a new camera state. The camera simulation implements an arcball camera, which rotates the scene based on user input. Arcball cameras provide an intuitive way for users to manipulate the scene. Imagine an enormous sphere that takes up the entire window. When users click and drag to rotate the scene, they are rotating that sphere. Given two points on a sphere, it's possible to calculate the two respective vectors from the origin to the sphere surface. The cross product of the two vectors yields an axis of rotation and the arccosine of the dot product yields a rotation angle. From that, it's possible to create a rotation, which can be applied to the previous rotation to create a new rotation'.

The shape simulation takes the previous shape state and outputs a new shape state. It is responsible for keeping track of the scene geometry. Currently, the shape simulation

simply generates triangles at varying position offsets and destroys the list of shapes once it gets too large.

The physics simulation takes the previous physics state and the current shape state and outputs a new physics state. Currently, the physics simulation applies a velocity to each triangle.

The current shaders are simple. The vertex shader applies the model view projection transformation matrix to each vertex and the fragment shader produces depth values that are used in the depth test.

Future Work

I had some concerns about passing so much state to each loop iteration. As more simulations and more complex simulations are added, it's also harder to connect outputs of one simulation to the inputs of another. One potential solution is functional reactive programming (FRP). In FRP, time dependant functions operate on streams of data and functional programming is used to combine, filter, and otherwise manipulate streams. In FRP, the behavior streams are described as being composed from one another. When an asynchronous events occurs, it triggers automatic updates in the relevant streams.

Conclusion

Having never done OpenGL nor Haskell programming, I've learned a lot from this project. Real-time 3D graphics is completely doable in Haskell, but it does lack library support. Most graphics libraries for Haskell have only been used for small scale projects and are missing features. For example, Gpipes lacks geometry shader support. However, this only means there is a lot of potential in this area!