Final Project Report

Ruby's Return to Reality

by

Zev Friedman and Michael Kent

Ruby's Return to Reality is a demo screen from a children's game (age range is estimated to be between 5 and 14) in which the main character, Ruby, is magically transported from her own world into the world of a stick puppet theatre by the Sinister Sorcerer C++. While there, Ruby meets up with Prestidigitator Python the Powerful who is willing to help her if she first helps him save his world.

During the course of Ruby's adventures, she gains a number of sidekicks, including Fluffy the Cat, Fluffy the Dog, and Fluffy the Fish. She also encounters the inhabitants of the land, who send her on tasks, mostly consisting of finding objects and solving geometry-related puzzles, such as finding reasonable tessellations for various unusual shapes. These characters will have alliterative names based on programming languages; some of our favorites we have come up with so far are COBOL the Koala, Lisp the Lion, Haskell the Hawk, and finally Logo, the Scribbling Turtle.

Our demo takes place after she has successfully saved the world of puppet theatre and is now collecting the last geometric piece, a 4D hypercube (tesseract), needed for the Magic Mystery Machine, with which Python will be able to transport her back into her own reality.

Installation and Platforms

This project will compile on both Windows and Mac, however towards the end we exceeded past the point where my MacBook Air could handle it and rather than giving a sane error or rendering only some things, it would cause a segmentation failure. **Fortunately**, this runs on Zev's Windows laptop without error. So your mileage may vary, but the framework is cross platform.

Compilation

The source code is located on github at https://github.com/TTU-Graphics/FinalProject. When downloaded, it will create a folder called FinalProject.

Windows

To compile on windows, load the FInalProject solution in Visual Studio, build the project, and run it (with or without debugging).

Mac

To compile on Mac, move into the directory FinalProject/FinalProject and run the **make** command. The makefile will create an executable file called *menger* so run **./menger** to run the program.

Linux

We were not able to verify Linux compatibility, but likely only small changes to the makefile are required.

Usage

The camera can be moved around the scene by clicking and dragging. This will cause the camera to orbit at a specific distance. You can move the ruby character with the ijkl keys where **i** moves towards the back of the scene and **jkl** moves left, towards the front of the scene, and right respectively.

Implementation

Object Oriented Design

Since about half way through the semester, Mike has been working on making the OpenGL interaction framework Object Oriented. To this extent, there are several key classes:

- **AbstractModel** - simply defines some functions every model should have (this has recently been created to facilitate instancing of models).
- **Model** - The basic model. This will create the buffers and interact with the shader files handles the basic implementation of the AbstractModel.
- **ShadedModel** - extends Model to handle the shading variables.
- **Light** - holds information on lights.
- **ObjModel** - extends ShadedModel and objs Wavefront .obj files.
- **GLSLProgram** - directly interacts with the shader and provides easy access functions to what would normally be long or multiple OpenGL calls. Many functions also end in `(const char*, …)` which is meant to be used exactly like sprintf and allows for input of arrays in such manners as `("Light[%d].Position", index)`.

Model Loader

The model loader is implemented in the **ObjModel** class. It is designed to read Wavefront .obj files. For our project, we have taken some liberties and made some assumptions:

- One one model per file.
- Models are triangulated (meaning no quads).
- Models must have a texture and define the texture coordinates.
- Models have only one material.
- Each material has a texture and normal map (not simply bump, though I believe it is defined the same way).

Shaders

This project includes several classes of shaders of varying complexity. The most basic is a simple pass-through shader with vertex-based coloration and no lighting calculations. This shader applies solely to objects generated by the particle fountain. Similar to this is the Tesseract shader, which includes logic in the vertex shader which performs rotation about the six principal planes of rotation in 4d space and projection into 3d space. Next, we have a dynamically built shader which implements fragment based Blinn-Phong shading, supporting a variable number of lights. Unused in this project, we have a texture-mapping shader which simply applies a texture to an object based on texture coordinates provided. This shader does not consider lights.

Finally, the shader which is used for the majority of the project implements fragment based Blinn-Phong shading on texture-mapped objects with normal maps. Because Blender exports normal maps in object space, no mucking about in tangent space is required and normals can be read straight from the normal map file at points given by the texture coordinates. This normal mapping allows us to simulate higher polygon count objects by using more accurate normals for lighting calculations, giving the illusion of higher geometric complexity on the inside of an object (though as it does not change the actual geometry of the rendered object, silhouettes are still jagged).