Randall Smith (agrippa@cs.utexas.edu) is in charge of this project.

Introduction

In this assignment, you’ll be given some metaballs code, and you’ll have to implement a bunch of rendering effects on the resulting mesh. You will implement these effects using shaders.

1 Requirements

A substantial amount of code has been set up for you already, which handles mesh and normal generation, shader loading, texture loading, particle simulation, and camera control. Sample shaders are included for diffuse lighting and planar texture mapping to get you started, and a texture library is included with all the textures you will need. Your job is to write a collection of shaders that accomplish the following tasks:

- Implement a per-pixel Blinn-Phong lighting model, including ambient, diffuse, and specular terms. You can start by copying the included diffuse shader, then modifying it to include the specular term. Refer to the OpenGL shading language specification (version 1.20) for information about the built-in uniforms that pass lighting and material information into the shader.

- Implement two types of automatic texture coordinate generation: cylindrical and spherical coordinates. In both cases, modulate the texture with simple diffuse shading. You can start by copying the included planar mapping shader, and modifying the way the vertex shader generates texcoords. When you’re done, the metaballs will appear to “swim” through the texture as they move around.

- Take your spherical texture coordinate shader and combine it with your Blinn-Phong shader, then use a specular map to modulate the specular component over the surface of the mesh. Use textures/earth.bmp as the diffuse map, and textures/earth_spec.bmp as the specular map. Your final result should show specular highlights only in the oceans.
• Implement spherical environment mapping, using textures/sphere_map.bmp as the texture. Your result should look like a very reflective glob of mercury.

• Build normal mapping on top of your spherical texture coordinate shader. Use textures/rock.bmp as the texture and textures/rock_norm.bmp as the normal map. This is a complicated shader: it will require that you create a local coordinate system for every pixel on the surface, look up the new normal in the texture map, rescale it from the range [0, 1]^3 to [−1, 1]^3, and express it in the local coordinate system. Your result should look like the simple textured version, but with much more detailed lighting.

2 Using the starter code

As mentioned before, the starter code handles a good chunk of the setup work for you. It maintains a list of compiled shaders and allows you to toggle between them at runtime.

Keyboard commands

• f: cycle through shaders
• space: pause / unpause animation

Shader and texture loading

Shaders and textures are loaded in the initGL() function in main.cpp. To register a shader, allocate a new shader object and pass the paths to your vertex and pixel shaders into the constructor. After it’s been created, you can register up to 8 textures with it using the add_texture function. Once that’s done, push it onto the list shaders. You can do all this just by copying and modifying the existing code.

When a shader object is created, it will load, compile, and link the supplied vertex and pixel shader, and emit any errors and warnings that the compiler or linker produces to the console. If your shaders contain any errors, they will show up there.

Uniforms and attributes

The code doesn’t facilitate adding your own uniforms and vertex attributes; all of the required shaders can be implemented just using the builtin ones provided by OpenGL. The exception is texture sampler uniforms, which are required to do texture lookups.

To handle this, when you add a texture to a shader object, the code will automatically register it as uniform sampler2D tex0. If you add additional textures, they will be registered as tex1, tex2, and so forth. You can use these uniforms in your fragment shader to sample the textures. Refer to the planar mapping example to see this in use.
If you would like to supply other uniforms and attributes to your shaders, you’ll have to change the code yourself.

3 Working on this assignment

In order to work on this assignment, you obviously need a computer capable of running shaders. Most of the public Linux machines in the department will do, but the best ones by far are the Intel machines in the ENS 31NR computer lab. These are brand new, top-of-the-line machines with excellent CPUs and GPUs.

Working remotely on this assignment is somewhat complicated. One cannot just open a remote X session into the Linux lab and have it work, since it will try to use whatever GPU is local on your machine instead of the one in the machine you’re connecting to. Even if your local machine is capable of running shaders, X tunneling typically restricts the available version of OpenGL to 1.4, which doesn’t support them. A better option is to just do all of your development on your local machine, then test it on the department Linux machines at the end.

We tried to make the code as portable as possible, but newer versions of OpenGL have haphazard distribution. If you encounter trouble getting the project to build on your machine, make sure to install the GLEW library and link your project against it. It goes a long way towards smoothing out incompatibilities among versions of OpenGL and giving you access to whatever your hardware can actually do.

4 Tips and tricks

Some things that may help with the project:

- In this project, mesh vertices and normals are submitted in world space, and the only transformation in the modelview matrix is the camera transform, which converts those vertices into eye space. It is very important that you keep track of which space you’re doing your computations in!

- Lighting computations are traditionally done in eye space, but that doesn’t mean it’s always the best idea. Sometimes doing lighting in world space is easier to understand.

- There is only one light source in the scene, and by default it’s on top of the viewer. This means that all specular highlights will be pointed straight at the camera. The light’s position is specified in eye coordinates, so keep that in mind if you try doing lighting in world space instead.

- The cube in which all geometry is generated extends from $< -1, -1, -1 >$ to $< 1, 1, 1 >$ in world space. This simplifies texture coordinate generation quite a bit.

- Implement as much as you can in the vertex shader. Putting anything in a pixel shader is about 100 times more expensive.
• You are encouraged to search far and wide for resources to help you understand how to write these shaders. However, don’t do yourself the disservice of copying them wholesale. You won’t learn anything, and we might catch you, which has consequences.

5 Bells & whistles

If you made it through all that, you’ve finished the project. No additional features are required, however, there are many ways to make it more awesome if you’re so inclined. If you impress the grader, it’s possible to earn a good amount of extra credit! Here are some suggestions:


• Extra particle effects: Currently, the metaballs are defined by particles in simple orbits. Replace that behavior with a more complicated particle system and see what cool effects you can come up with. Fluid simulation anyone?

• Model viewer: Dig out your old model loader code and integrate it into this project. Ever wondered what a reflective Cessna looks like?

6 Screenshot

5% of your grade in this project is a screenshot that you submit along with your code. Generate the best image you can (a custom scene file is recommended!) and we’ll put the results up in a gallery after the projects have been submitted.

Logistics

Like all projects in this class, you may work in a group of up to 2 people.

Any clarifications and revisions to the assignment will be posted on the class web page and announced in class.

In case you needed a reminder: START EARLY. You have two weeks for this project, but if you manage your time badly, you will have a serious problem. The project is due the same day as the final, so if you wait until the last minute, your life will be miserable.

Turnin and grading

You can develop your application on whatever operating system you like, but before you submit it, you must make sure that it builds and runs properly on the department Linux machines! All
grading will take place on these machines, so if your code doesn’t work on them, you’re in trouble. To submit your code, use the department’s turnin script, like so:

turnin --submit agrippa proj4 shaders/

Replace shaders/ with whatever your code directory is named. Make sure that all the necessary code is submitted, as projects that don’t build are worth nothing! On the other hand, make sure that you’re including just the files for this project, and not several hundred megs of extra junk.

Make sure you have included your (and your partner’s) name and UTCS ID in a comment at the top of each of your files. Also, include a readme explaining the usage of your program, including any menu options or keyboard commands. If you use any slip days, be sure to put that in your readme and email the TA as well.

To get a grade on this project, you’ll need to sign up for demo session with the TA after you’ve submitted your code. This is when you’ll run through all the functionality in your project, and the TA will be able to verify that everything is working as it should. This is also your chance to show off your extra features and slick interface!