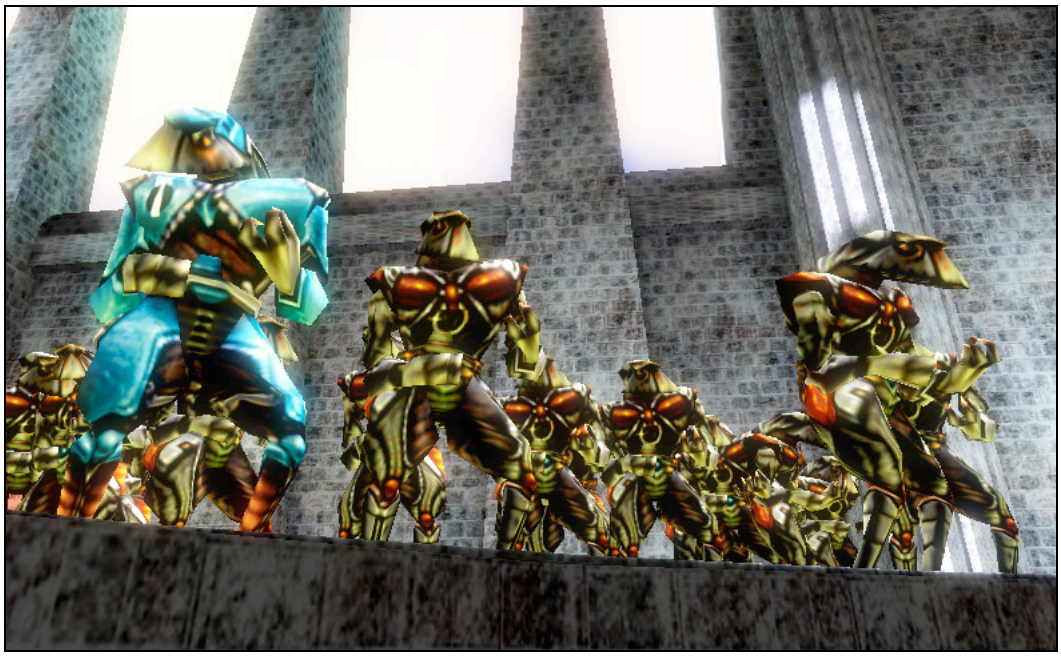
CS371 Spring 2007
Prof. Morgan McGuire

Assigned: Wed. 4/04/07
Due: Wed. 4/11/07 11:59 pm

Project 0x06: Real Time



In this assignment you will implement the simulation, pose, transformation, projection, and shading porting of a programmable hardware rasterization pipeline. Most of the implementation code will be in OpenGL 2.1 GLSL pixel shaders that execute on the GPU.

Last week, half of the class worked on the NPR assignment and half worked on the FX2 assigment. The NPR students participated in a code review. Those students will be graded this week solely on the project specification.

If you weren't in a code review last week, then for this assignment you will be graded on both the specification and our ability to present a small piece of your code at a brief one-on-one **code review** during the Thursday lab period. You will not know which part of your code you will be presenting ahead of time—I will randomly select one method and ask you to explain and defend the implementation. Your partner will not be available during the code review and you are expected to be prepared to present any piece of code, regardless of who wrote it.

This is the last regular project of the semester. From here on you will work exclusively on a final project of your own design.

Specification

FOR THIS ASSIGNMENT, YOU MAY NOT CALL THE RENDER METHOD OF ANY G3D CLASS EXCEPT G3D::Sky and G3D::BSPMap.

Part A. Create a row of models shaded in the following local illumination models:

1. Lambertian BSDF:

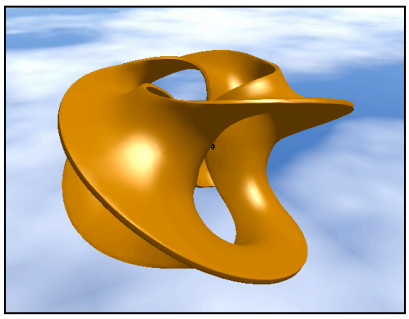
$$L(x, \omega_o) = L(x, \omega_i) * \rho_{diffuse} * \max(0, N \cdot \omega_i)$$

Where, for our purposes, $L(x, \omega_i)$ is the "light color", N is the surface normal, and ω_i is the direction to the light.

2. Phong BSDF:

$$L(x, \omega_o) = \rho_{diffuse} * L_{ambient} + L(x, \omega_i) * \rho_{diffuse} * \max(0, N \cdot \omega_i) + L(x, \omega_i) * \rho_{specular} * \max(0, \text{reflect}(\omega_i, N) \cdot \omega_o)^s$$

Where, for our purposes, $L(x, \omega_i)$ is the "light color", ω_o is the vector from x to the eye, N is the surface normal, s is the "shininess" of the surface, and ω_i is the direction to the light.



3. Phong BSDF, dual-hemisphere ambient, and a diffuse texture map:

$$L(x, \omega_o) = \rho_{diffuse}(x) * L_{ambient}(N) + L(x, \omega_i) * \rho_{diffuse}(x) * \max(0, N \cdot \omega_i) + L(x, \omega_i) * \rho_{specular} * \max(0, \text{reflect}(\omega_i, N) \cdot \omega_o)^s$$

Where $L_{ambient}(N) = L_{bottom} + (L_{top} - L_{bottom}) * (2N_y - 1)$.

4. Phong BSDF, dual-hemisphere ambient, and environment map:

$$L(x, \omega_o) = \rho_{diffuse}(x) * L_{ambient}(N) + L(x, \omega_i) * \rho_{diffuse}(x) * \max(0, N \cdot \omega_i) + L(x, \omega_i) * \rho_{specular} * \max(0, \text{reflect}(\omega_i, N) \cdot \omega_o)^s + L(x, \text{reflect}(\omega_o, N)) * \rho_{reflect}$$

Part B. Create a real-time animated scene

Your scene can contain objects animated by changing their cframe or by changing their pose (i.e., MD2Model and some ArticulatedModels can change their shape). Although you can use the CollisionDetection routines and your knowledge of physics (e.g., $x_1 = x_0 + v_0*t + \frac{1}{2}*a*t^2$), your simulation need not be physically based. You could move objects according to a pre-scripted path or simply along a mathematical function like cos(t).

Implementation Tips

Begin by copying /usr/local/371/demos/empty/main.cpp. When you compile and run that file you will see a cube, a sphere, and a set of axes floating in a sky. Press TAB to fly the camera using first-person controls (W, A, S, D to translate, mouse to rotate). Press tilde (~) to bring up the in-game command console. The console initially has only two commands: help and exit. Press ESC once to close the console and a second time to quit the program.

Place your simulation code in the onSimulation method. The pose and cull methods should be in onGraphics. Projection and transformation go in the vertex shader. Shading goes in the pixel shader.

Vertex and pixel shaders are loaded at runtime. They are data files. So put them in the data-files directory, not the source directory.

Add your code to copy data to the GPU and render it using the default setup before you add shaders. Once you have posed a model, you can upload it using the following code:

```
// Obtain a reference to the underlying geometry
const MeshAlg::Geometry& cpuGeometry      =
      posed->objectSpaceGeometry();
const Array<Vector2>& cpuTexCoordArray    =
      posed->texCoords();

// Allocate memory on the graphics card for the geometry
// (we need some padding because VAR allocation
// rounds off to an 8-byte boundary.
VARAreaRef area =
      VARArea::create(cpuGeometry.vertexArray.size() * 2 *
            sizeof(Vector3) + cpuTexCoordArray.size() *
            sizeof(Vector2) + 24);

// Upload geometry to the graphics card
VAR gpuVertexArray(cpuGeometry.vertexArray, area);
VAR gpuNormalArray(cpuGeometry.normalArray, area);
VAR gpuTexCoordArray(cpuTexCoordArray, area);
```

You can render using the following code (put it after the lighting setup)

```
rd->setObjectToWorldMatrix(posed->coordinateFrame());
rd->beginIndexedPrimitives();
      rd->setVertexArray(gpuVertexArray);
      rd->setNormalArray(gpuNormalArray);
      rd->setTexCoordArray(0, gpuTexCoordArray);

      rd->sendIndices(RenderDevice::TRIANGLES,
                        posed->triangleIndices());
rd->endIndexedPrimitives();
```

Vertex and pixel shaders are written in GLSL. Vertex shaders are in ".vrt" files and pixel shaders in ".pix" files. You load these from disk at runtime with the command Shader::fromFiles(vertexFilename, pixelFilename). Shader compilation errors are generated when the shader is loaded at runtime. They are often extremely hard to interpret, so I recommend making small changes, breaking complex expressions into multiple lines, and testing very frequently.

The specification for the version of GLSL implemented on our NVIDIA 7600 graphics cards is online at:

http://oss.sgi.com/projects/ogl-sample/registry/ARB/GLSLangSpec.Full.1.10.59.pdf

In the shaders, the following builtin variables are available to you:

- mat4 g3d_ObjectToWorldMatrix
- mat4 g3d_WorldToObjectMatrix
- mat4 g3d_CameraToWorldMatrix
- mat4 g3d_WorldToCameraMatrix
- mat4 gl_ModelViewMatrix = g3d_WorldToCamera * g3d_ObjectToWorld
- mat4 gl_ProjectionMatrix
- mat4 gl_ModelViewProjectionMatrix = gl_ProjectionMatrix * gl_ModelViewMatrix
- mat4 gl_NormalMatrix: inverse transpose of the upper 3x3 submatrix of the gl_ModelViewMatrix
- vec3 gl_Normal: the object-space normal (in the vertex shader only; you must create a varying variable to pass this down to the pixel shader)
- vec4 gl_Position: the transformed and projected vertex; you must compute this in the vertex shader from gl_Vertex
- vec4 gl_FragColor: the color of a pixel; you must compute this in the pixel shader

- vec4 gl_TexCoord[0]: the texture coordinate, interpolated across the face of the triangle

Variables are passed between programs using global variables. These variables can be one of three types:

1. uniform – Written by C++, read by a shader

2. varying – Written by the vertex shader, read by the pixel shader

3. builtin (do not declare these explicitly) – Read or written by the shader and the GPU as specified in the documentation.

To pass values from C++ to a uniform variable use the Shader::ArgList::set method. For example, the following are some of the variables that you will need to pass to your shaders:

```
Vector3 wsEye =
      defaultCamera.getCoordinateFrame().translation;

shader->args.set("wsLight",
       lighting->lightArray[0].position);

shader->args.set("lightColor",
       lighting->lightArray[0].color);

shader->args.set("ambientColor",
       lighting>ambientAverage());

shader->args.set("wsEye", wsEye);
```

Once you get your Lambertian implementation working, I recommend refactoring your code. The model loading, shader loading, and rendering for a single object should be in a separate class. E.g., "LambertianEntity." Since you're going to have a lot of different Entity classes, a good base class would be:

```
typedef ReferenceCountedPointer<class Entity> EntityRef;
class Entity : public ReferenceCountedObject {
protected:
    Array<PosedModelRef> posed;

    static void renderOne(RenderDevice* rd,
          PosedModelRef& onePosed);
public:

    CoordinateFrame     cframe;
    ShaderRef           shader;

    /** Updates the internal posed model and shader
     arguments from the current position and lighting. */
    virtual void pose(LightingRef& r,
           const Vector3& eyePoint) = 0;
```

```
    /** Sends the geometry from the last posed position
     (not the current object) to the graphics card with
     the right shader.  Must pose first. */
    void render(RenderDevice* rd);
};
```

Note that render is the same for all Entitys but pose is specific to the subclass. You might add a virtual simulate method if you do extra credit work with simulation (e.g., move the cframe, rotate it, or animate an MD2 or ArticulatedModel). Tip: you can't subclass those non-virtual methods.

In the actual subclasses, I recommend using a IFS models for the Lambertian and Phong entities and either 3DS (loaded with ArticulatedModel) or MD2 (loaded with MD2Model) models for the textured Entitys, since very few of the stock IFS models have texture coordinates. You can find IFS files in /usr/local/371/data/ifs and MD2 models in /usr/local/371/quake2.

The advantage of having an Entity base class is that in your App you can maintain an array of EntityRef instead of separate variables for each object you add to the scene—this is essential if your scene will have a lot of objects or if you want a generic way of making them interact with rendering.

The textures that go with MD2 models are often too dark. You can multiply them by 2 in your pixel shader or use the Texture::PreProcess::brighten setting to adjust the intensity when loading the texture using Texture::fromFile.

When working with MD2 models, beware that the models are in two pieces. tris.md2 is the "character" and weapon.md2 is the "weapon". So your character will be unarmed unless you store *two* MD2Models.

GLSL is really picky about syntax and conventions. Here are some tips:

- You have to write the decimal point in floating point constants: 1.0 instead of 1.
- sampler2D is the data type of a 2D texture when declared as a uniform in a shader.
- In the vertex shader, gl_MultiTexCoord0 is the builtin for reading the current texture coordinate; gl_TexCoord[0] is the builtin for writing the coordinate in the vertex shader and for reading it in the pixel shader.

Extra Credit Ideas

Here are some directions that you could take your rendering or animation for extra credit.

Remember non-photorealistic rendering? Try making a cartoon-shaded object, where there are only a few tones on the surface and the outline has black lines. There are a number of methods for accomplishing this, some of which can be implemented using only a pixel shader and some which will require you to write additional C++ code.

Real-Time Rendering describes several other BSDFs that are suitable for implementation in a rasterization system, including the Torrence-Sparrow model and various anisotropic BSDFs.

Implement texture coordinate generation in the vertex shader. Some models, like sphere.ifs, have no texture coordinates. But you can generate texture coordinates for them in the vertex stage. For example, cylindrical texture mapping sets the texture y coordinate to the object-space vertex y coordinate and the texture x coordinate to the arc tangent of the object-space x and z coordinates. There are many other possible mappings including spherical and projected.

A good way to enhance your animated scene without much effort is to drop a BSPMap object into the background. Quake3 BSP maps are readily available online (there are also some in the /usr/local/371/data/quake3 directory). They

come in compressed ".pk3" files. Rename the PK3 file to .zip and you can uncompress it. Inside the zipfile you will find several subdirectories. When loading the map, the path to the directory containing "maps", "textures", etc. is the one that you pass to the BSPMap constructor. The name of the ".bsp" file in the maps subdirectory is what you provide as the filename. You can't pose a BSPMap—just invoke its render method to draw it. Make sure that you don't have a shader set on the RenderDevice at that point in onGraphics, though!

A program that has multiple copies of the same MD2Model or IFSModel loaded is wasting memory. The point of posing objects is that you can load the geometry once and put it in a different pose for each entity instance. Use a static Table<std::string, MD2ModelRef> to create a cache and prevent models from being created more than once.

You now know enough to make a basic 3D game! MD2Model::Pose has a doSimulation method that will take care of animating your characters and. You can implement the onUserInput method to detect key strokes and use them to move characters (or just write AI for the characters). The FirstPersonController class that normally drives the debugging camera can instead be used to drive a character. Firing a weapon just means creating a new Entity that has a high velocity—and CollisionDetection has the methods you need to tell when the bounding box of a projectile overlaps the bounding box of a character.

Submitting Your Solution

1. Put your name, e-mail address, and the name of the file in a doxygen comment at the top of **each file**.

2. Create a "doc-files" directory that contains a **readme.html** file with your name, e-mail address, partner's name (if you worked in a pair) and anything you'd like to point out to me when I'm reviewing your program.

3. Put screenshots of each of your BSDF objects in the doc-files directory and include them in readme.html. Also include either a sequence of at least five well-spaced frames from your animated scene, or a short movie in MP4 or MOV format.

4. Delete all generated files using "icompile --clean". Do not hand in a build this week.

5. Change to the **parent** directory of your project and run the FreeBSD command:

```
submit371 realtime
```