Project 0x03: Scenes & Shaders

Due Friday, Oct. 10, 2008 at 6:00 pm

1 Specification

Implement a real-time rendering system that supports the following features:

1. Direct illumination
2. Ability to fly the camera using FirstPersonManipulator (implemented by default in G3D)
3. Local point source emitters
4. Spatially-varying Lambertian BSDF
5. Spatially-varying Normalized Blinn-Phong BSDF
6. Tone mapping: exposure, gamma correction, bloom, and saturation
7. Scene graphs with transformations
8. Triangle meshes imported from data files
9. At least three scenes that can be selected from the command line by specifying a string. These must include:
   a. “cornell”: Cornell box with a yellow sphere
   b. “figure”: A Stikfas figure in a plausible static human pose, like The Thinker, running, or a Karate stance. The figure must be constructed using a hierarchy. crate.ifs and wood-crate.jpg are provided to give your character a wooden crate to sit on or stand next to.
   c. One animated scene of your choice that takes advantage of a scene graph’s relative transformations, e.g., a walking figure, a snake, a moving train with a crane arm, a tank with a moving turret, etc.
10. Class and overview documentation including a results page with images and an architectural overview.
1.1 Restrictions

Your group submits a single solution through subversion. I encourage you to discuss the project with others outside your group and to help each other debug, derive algorithms, and understand the documentation. The code that you submit should be written solely by your group, except for trivial changes made while someone is helping you debug and code that you obtained externally in accordance with the guidelines from the syllabus. Any code not written by your group must be clearly noted in comments and documentation.

You may not directly use the VAR values inside ArticulatedModel::Part (you will need the CPU Arrays inside ArticulatedModel::Part, however.)

1.2 Evaluation

Your work will be evaluated for the following properties, in order of significance:

1. Satisfaction of the specification
2. Readability, including documentation
3. Mathematical correctness
4. Presentation of results
5. Design (including performance)
6. Programmatic correctness (compiling, memory management, safety)

In the event that you have known bugs, add a documentation section explaining what they are, how you attempted to debug them, and what you think the problem is. Use images as necessary. In general I will not deduct many points for bugs that you are aware of and investigated.

2 Advice

2.1 Getting Started

You’re going to start with a clean slate this week. Generate a new empty project using “icompile” with no arguments.

Since at this point you already know how to set up a scene, I provided interfaces for the BSDF, LambertianBSDF, Scene, and Entity classes, a partial implementation of LambertianBSDF, and a full implementation of ToneMap2. Add all of these to your project. Note that there are several data files provided with these.

Read the comments to see why we’re using ToneMap2 instead of G3D::ToneMap. Note that this class is very similar to the Film class from the ray tracer—it just takes its input from the GPU now. Declare ToneMap2Ref m_toneMap in your App, and be careful not to confuse m_toneMap and toneMap. Turn off G3D’s default tone mapper (it is off by default).

Delete all of the code that you aren’t using from the default G3D starter project. This includes Sky, Lighting, onConsoleCommand, and almost all of onGraphics. Your onGraphics should contain only ToneMap2::beginFrame, Scene::render, ToneMap2::endFrame, and 2D rendering code.
Most of the code that you write this week will not be used on the following project. This means that while you should still design well and document carefully, you do not need to design in a way that anticipates more general requirements in the future.

### 2.2 Architecture

You’re going to be working with the following major classes, some of which are designed by you, some of which are in the starter code, and some of which are in G3D:\(^1\):

**App**

The main application driving your project. The `onSimulation` method allows you to implement code that executes between rendering so as to create animations. If your animations are particularly “object oriented” in structure, you may want this to invoke a `Scene::onSimulation` method that invokes `Entity::onSimulation` for every object in the scene. The `App::onGraphics` method renders the entire image from scratch hundreds of times per second (although your monitor only updates at about 60 fps). You can use the `App::onEvent` method to detect key/mouse actions and use those to drive an animation to make it interactive. If you do so, you probably want to disable the default camera controller on `App`.

**Scene**

Contains all of the objects and lighting information for the world, and knows how to render it. Supports both insertion and removal of objects so that you can create animations in which objects are created and destroyed. The scene contains `Lighting` and a set of `Entities`.

**G3D::Lighting**

Scene uses this lighting environment class. The only part of this class that we’re using this week is the `lightArray` field and (optionally) the ambient fields. To create a point emitter, use `GLight::point(pos, lightPower, 0, 0, 1, true, true)` and append the new light to the array. The last five arguments are for constructing non-realistic lights under certain circumstances. Don’t change them from the values specified here.

**G3D::GLight**

Simple point emitter storing a position and a light power (it stores other fields as well, but we don’t need them this week).

**Entity**

Base class for objects in the world. You are only required to implement one subclass: `TreeEntity`.

**TreeEntity**

This is a subclass of `Entity` that implements a hierarchy of objects (part of a “scene graph”). Each `TreeEntity` should contain:

i. A reference frame (I suggest you use `CFrame` instead of `CoordinateFrame` [they are the same class] to save yourself some typing, and name your variable `m_frame`.)

ii. A reference counted pointer to a `TriList` (described below), which may be `NULL`.

iii. A pointer to the parent, which may be `NULL` (Use `TreeEntity*`, NOT a reference counted pointer `TreeEntityRef`; reference counted pointers don’t work if you have cycles in your data structures, and back pointers are not second class citizens in C++.

---

\(^1\) Given how much detail I’m presenting here, I expect your architectural overviews to be at least as detailed from now on! Feel free to copy parts of this text for your overview or class documentation.
are cycles. To get the pointer out of a Ref class, call the .pointer() method on it)

BSDF
The BSDF class this week has a very different interface than for the ray tracer. That’s because it will be implemented on the GPU as part of a rasterization pipeline, using shaders. A shader combines the EF, emitter geometry, and BSDF. This is awkward; we have to write one shader for every combination of EF and BSDF. This is a serious design flaw with the GPU architecture. We’ll deal with it now the way that many game companies do: assume a single EF type and geometry, which here is a point emitter.

LambertianBSDF
The spatially varying, perfectly diffuse BSDF with a single point emitter. The C++ side of this is implemented for you, and the GLSL shader code is stubbed out. Just implement the GLSL main() bodies.

PhongBSDF
Normalized Blinn-Phong BSDF with a single point emitter. Since this is a superset of LambertianBSDF, just subclass LambertianBSDF instead of repeating implementation code. Note that PhongBSDF has three parameters: lambertian, glossy, and gloss exponent (shininess). Each can have its own wavelength-varying texture map.

TriList
A description of a single, rigid part of the scene. It contains a BSDF and an indexed triangle list. Recall that an indexed triangle list contains a vertex array, a normal array, a texture coordinate array, and an index array. The index array will be Array<int> and stored on the CPU. All other arrays will be “vertex arrays” stored on the GPU using the VAR class.

G3D::Shader
Class that manages the run-time loading, compiling, and argument matching for GPU programs. It is really hard to debug shaders. You can’t access the code from printf or from a debugger, the error messages are generally inscrutable and often have the wrong line numbers, and the compilers are flakey anyway. Make tiny changes to your shaders and test frequently. Seek clever ways of drawing intermediate results to the screen to figure out what is going wrong.

G3D::ArticulatedModel
G3D class that can load hierarchical models. This is useful for building TriLists / TreeEntities. You are only required to implement importing an ArticulatedModel with a single part (at the root) and converting it to a TriList. However, if you choose to implement recursive importing of a whole ArticulatedModel into a TreeEntity, you’ll be able to make much better scenes (e.g., the entire Sponza).

2.3 Creating TriLists
To create a TriList, you must transfer geometry from the CPU to the GPU. First, allocate a block of memory on the GPU using VARArea::create (this stands for “vertex array memory area”). For this call, you must compute the size in bytes of the area that you’ll need. Your data will take the same space on the GPU as the CPU, although you should add 32 bytes to the total to allow for memory alignment padding. Use the C++ sizeof operator to find out how big individual classes
are, e.g.: sizeof(Vector3). Note that you can’t use sizeof to figure out how many bytes an array of objects takes because sizeof(Array<Vector3>) tells you how large the array object is, but the array object is just the size of n int and a pointer into the heap—you need to know how many elements are in the heap, which is Array::size().

Now create VAR. A VAR is a very small class that contains a pointer into GPU memory. The VAR constructor takes the G3D::Array that you want to upload and the VARArea into which you want to load it. (Although you won’t have to do so in this program, be aware that you can pass VAR values by value, just like you would for pointers. This isn’t the actual “array”, since the memory is on the graphics card. There is no need to free VAR data structures; G3D manages them for you once created.) You’ll need one VAR for vertices, one for normals, and one for texture coordinates.

I don’t want you to re-use the VAR data from inside ArticulatedModel because I want you to experience writing the code to upload data to the GPU for yourself.

The TriList should be able to draw itself. So far you’ve been using RenderDevice::begin…end to draw using the GPU. That method is very slow because it requires one function call per vertex. Now we’ll execute much faster. Use G3D::RenderDevice::beginIndexedPrimitives…endIndexedPrimitives to create a block that processes whole vertex arrays. Within that block, set the vertex, normal, and texture coordinate arrays on the RenderDevice. GPUs support multiple texture coordinates per vertex. Just set texture coordinate 0. The last call within your block should be RenderDevice::sendIndices. Pass it your index array and indicate that you want to render TRIANGLES.

Somewhere back near the beginning of your TriList rendering method, you must configure the shader on the GPU. Use BSDF::setShader to handle this.

### 2.4 Other Tips

When constructing your Scenes, only insert the root TreeEntitys into the scene. Do not insert the child TreeEntitys—they should be rendered by recursing down the TreeEntity structure. While implementing the methods to build scenes I found it useful to go back and add overloaded TriList::create methods to build TriLists from files, from Arrays in memory, and from an existing TriList with a different BSDF.

Set the clearColorValue of the RenderDevice to something other than black so that you can see your triangles even when they render as all black. Remember that triangles are one-sided and you won’t be able to see them from the “back” side.

Remember you can debug shaders by drawing values to the screen, for example, to see the texture coordinates or normal vectors.

I left some commented out code in a Scene.cpp file. This code gives you the magic constants you need to build the stick figure. I created my Cornell box by loading square.ifs (rescaling it by sqrt(2) in the process) and using CFrame::fromXYZYPRDegrees to flip it to the appropriate orientation for each wall.

As previously, you can load any 3DS, IFS, OFF, or PLY2 model to make your scenes. I provided a bunch of these in the /usr/local/371/G3D…/data directory (something in the nightly update scripts keeps changing the permissions—let me know if you can’t see these files on a given day!)
You can also download free models from the web, e.g., from the Princeton Shape Benchmark, the Brown Mesh Set, 3D Cafe, or TurboSquid. Programs like Blender, Poser, SketchUp Pro, Maya, and 3DS Max will let you create these files manually.

Project 0x03 Groups

red
09cmz
09twb

orange
10dpf
09sb

yellow
09ajs
09jmc

green
09kaw_2
09ack_2
10msl

blue
09msg
09hc

indigo
09wkj
10kl

svn co svn://graphics-svn.cs.williams.edu/371/3/xform-GROUP
Your username and password are the same as last week.