Assignment 5: Sceneview

Written Homework Due: Friday, April 3 (9:30 AM)
Programming Project Due: Friday, April 3 (5:00 PM)

1 Introduction

When you look at a moderately complicated computer generated image, you realize that no person (who has a life), would sit down and hand place every single one of the triangles necessary to create that image. The triangle is simply at too low a level to be much good for large-scale scenes. Displaying a scene of moderate complexity would require the placement of thousands of triangles, if not tens or hundreds of thousands. To help bridge the difference between low-level triangles and the complicated scene, we will interpose a layer which is a medium-level description language. This language will deal with shapes, not triangles, and will deal with applying transformations to those shapes, as opposed to specifying the coordinates of those shapes absolutely. Color will also be dealt with on a per-shape basis, as opposed to a per-triangle basis. As you have probably realized, you have all the tools necessary to deal with such a scene description. In this assignment, you will be taking a hierarchical scene description (which you will read from a text file), and transform that scene description into its visual representation. To do this you will need your shape tesselators from Assignment 2, linear algebra, and your camera from Assignment 3. To get a better idea of what you will be doing, check out the demo. Switch to sceneview and open a .sc file. Example scene files can be found in `data/scenes`. Take a look at both the scene description files as well as the output, and become familiar with how they relate. You'll be creating a scene of your own to make sure that the concept behind the scene description is understandable before diving into the programming side of this assignment.

2 Requirements

The first thing you need to do is make a scene (you can use the demo program to display it). This scene must contain at least one subgraph example and at least a half-dozen shapes. **Turn in the scene file (.sc) with your lab**.

Second, you are required to implement a program that will read in, parse, process, and display a three-dimensional scene file that conforms to the CSC 321 scene file format specification. Most of the parsing is already done for you (camera, lights, and parts of the object and subgraph descriptions). You should design some sort of data structure for holding and accessing the scene data in memory. The use of STL (Standard Template Library), to create your data structure is highly encouraged. By using STL, you may be able to skirt some grunt work.

For details about the file format, see the additional handout "The CSC 321 Scene File Format". The result of correctly processing a scene file should be an image shown in a window, with objects in the correct positions and with correct attributes (*i.e.,* color, size, etc.). Note that this assignment does not require you to allow the user to manipulate objects in the scene. However, you will need to make sure you are viewing the scene from the correct location.

You are required to implement subgraphs. By implementing subgraphs, one may instantiate several copies of the same object without having to declare the same object's graph more than once

in the scene file. This also lets you compose the modeling task into sub-tasks. For example, an arm composed of two cylinders can be declared as a single subgraph labeled "arm". Since the arm is a subgraph, it can then be instantiated multiple times. The relative positions of the two cylinders that compose each arm remain the same, but the two arms may be placed in different positions. Please see the scene file format for more details.

Many of the attributes described in the file format document are not used in this assignment, but will be in later ones. When designing your data structure/parsing algorithm, take into account this extra data; don't just discard it because it isn't useful at this point in time. For example, you will not be implementing texture mapping in Sceneview, but your parser should handle storing the texture map name. In summary, you will need what you write in this assignment for the later assignments, so design for flexibility. It isn't too difficult; I just want to make sure you understand that if there is one place where you should absolutely not hack something together with spit and bailing wire, this is it. The skeleton of the parsing and data structures has been provided for you, but you will need to study the provided code carefully and fill in the missing methods appropriately. The purpose of the provided inner classes to `Scene` are explained below.

- `Object` This is the base level of the tree. It stores shape and material information.
- `Tree` Your scene is composed of trees. A tree is simply a list of nodes.
- `Node` A node corresponds to a trans block. It is associated with a transformation matrix and has either a child object or child tree.
- `MasterSubgraphs` This data structure keeps track of named mastersubgraphs so you can instantiate them later. It is up to you how you wish to represent this mapping.

In this assignment you are essentially building a tree that represents the scene. Some parts of the tree may be duplicated. The leaves of the tree will point to an instance of the corresponding shape class (you should only have one copy of each tessellated shape). In this assignment you will be traversing the tree in order to draw the elements. In the ray tracing assignment you will be traversing the tree (possibly multiple times) performing ray intersections with each object. Keep this in mind when you build your tree data structure.

3 Support Code

Look through the file `MyScene.h` to see exactly what has been provided for you. You will need to start by augmenting the inner classes with variables to actually store what is parsed. The parsing code is located in `MyScene.cpp`. When you are able to load files without errors, you can add drawing code that traverses the tree starting at the root. The setup of the lights has been provided for you. You need to set up the camera. Note that this code uses your camera class from the camera assignment. This must be working properly or else you will not get proper results.

As you go down the tree, you will need to aggregate the effect of the transformations you encounter. OpenGL handles this with a matrix stack, which is implemented with `pushMatrix` and `popMatrix`. Be careful that for every matrix you add onto the OpenGL stack, you remove it somewhere else, otherwise OpenGL will silently stop displaying anything (or worse depending on how well-behaved your video drivers are). The matrix class is correctly set up so that `glLoadMatrixd( &mat(0,0) )` and `glMultMatrixd( &mat(0,0) )` will work properly.

A portion of your grade (10 pts) will be determined by a sample scene file you create from scratch. The data directory has example scene files. Your scene does not need to be super-elaborate, but you will not get points for "Cube, at (0,0,0)" as your new scenefile. Create one that is educationally rewarding for yourself, and everything will be fine. This just means make a scene that utilizes the constructs of subgraph, and teaches you about how such files are created.

4 Coding style

Memory management: There should be no memory leaks in your code — every object that is created should also be deleted. You will lose points if your memory allocation strategy consists of allocating a huge number of objects and hoping you don't run out of them... Also, you should not have multiple copies of each of the shape classes or of the subgraphs. (The exception to this is if you write a "flatten" routine.)

Bad data: Your code will be run on scenefiles that are incorrect — for instance, trans nodes without objects, multiple objects in a trans node, missing subgraphs, etc. You should handle these cases gracefully, not seg fault. The syntax checking is mostly handled for you already, but your code should, again, deal gracefully with parse routines that end early due to bad syntax.

Correct defaults: The user is not required to supply all data; the defaults are given in the scenefile format document.

Data correctness: Obviously, the easiest way to check if you're parsing the scene correctly is to compare it to the demo. There are some suggestions below for how to test your scenefile code piece by piece. How do you guarantee that you've exercized every part of your code and every scenefile option? Describe your solution to this in your ReadMe file. Don't forget to test both your parsing code and your drawing code.

5 Additional Notes

Included here are a few additional notes that are of no relevance to your grade, but that you may find useful.

- Just so you know, you will be using your parser for the ray tracing assignment. This is also a difficult lab; get started early.
- Start with a camera on the z-axis looking at the origin, with an object at the origin. Add a transformation to the object. Add a second transformation. Add a transformation to the camera. Then check subgraphs.
- I recommend putting break points in every switch and if statement option. Don't take them out until you've verified that the corresponding piece of code is correct. This also lets you know which part of the code you haven't reached yet.
- Just because you parsed the data correctly doesn't mean you *stored* it correctly. One way to check that the data is correct is to write a recursive print method for each element of your tree. Call this *after* you've parsed the tree and when you go to draw it.

6 Extra Credit

For extra credit, try adding extra keywords/objects or implement texture mapping in OpenGL. For instance, you may wish to augment the scenefile specification to add simple animation support, and in turn support it in your sceneviewer. Make sure that regardless of what extra features you may add to your viewer, it is still capable of reading the standard CSC 321 file format.