Project 4 Sceneview

Introduction to Computer Graphics, Fall 2015

1 Introduction

In order to visualize a complex 3-dimensional scene, thousands of tiny triangles must be drawn to the screen. It would be senseless to require the manual placement of each one of these triangles; instead, we usually define the scene in terms of the various primitives that compose it. Even better, we allow for primitives to be grouped, and then we reference those groups as user-defined primitives.

In Sceneview, you use a TA support library to parse a cs123-specific scene file format, and then interface with that library to extract all of this scene information for rendering. After you’ve imported all of the data, you must traverse the scene graph (That’s what the data structure in which you stored the scene info is called) and make a list of objects to draw to the screen. Each object will have certain properties needed for rendering (color, for instance), and a transformation that will move vertices and normals from object space into world space for the specific object. Once you’ve made this flattened list, you will iterate through the list, set up the appropriate state information, and render each object to the screen. You’ll be making good use of your code from Shapes and Camtrans in order to accomplish these tasks.

1.1 Demo

As usual, we have implemented the functionality you are required to implement in this assignment. Your results should be identical to the results in the demo; this includes camera positioning, camera parameterization, aspect ratio, object locations, object colors, light positions, light attenuations, and light colors.

One caveat: some scenes have primitives that have similar Z-buffer values and this causes an effect called Z-fighting. The effect causes random pixels to be rendered with the color of one primitive or another in a non-deterministic manner. Don’t worry if this happens and your pixels don’t exactly match the demo. As long as the primitives are in the correct place (within epsilon) then you have done it correctly.

You can find test files with the “.xml” extension in /course/cs123/data/scenes/. You can feed it any of the files there, but the ones designed for Intersect and Ray will probably be boring in Sceneview.

2 Requirements

You are required to implement a program that will build a scene graph internally and then efficiently render the scene to the display using calls to OpenGL. You will probably want to use the STL (Standard Template Library) in various places within this assignment; there are places where maps and lists will make your life much easier. See the Intro to C++ presentation for more information about the STL. You can also use Qt’s built-in data structures (such as QList) which have additional conveniences. See the Qt documentation for more information on that.

As usual, there is extensive support code for this assignment. Don’t forget that in Qt you can hold ctrl and object/methods names will become hyperlinks that will take you to their definitions. This can be very useful when trying to figure out what the support code is doing.

In addition, your code for this assignment will use your Camtrans and Shapes code. If your Camtrans or Shapes isn’t working (or is leaking memory), you will need to fix it now! But don’t say we didn’t warn you before…

For this assignment, you will be doing most of your work in Scene.cpp/h and SceneviewScene.cpp/h. These files can be found in the scenegraph folder.

2.1 The Scene Classes

SceneviewScene (which you will complete in this assignment) implements an OpenGLScene. It will hold a list of objects that are to be rendered to the screen. In addition, the SceneviewScene will contain the cumulative transformation matrix for each object. In the setLights method, the lights will be set up. Then, in the renderGeometry method, each object will be rendered (you’ll want to call upon your Shapes code for this).

Recall from class that a scene graph’s purpose is to deal with hierarchical transformations. Therefore, we have made a distinction in this assignment: objects and lights are part of the scene graph, and the camera is not. The camera is part of the canvas. You will find this distinction to be handy when it comes time to implement additional rendering interfaces.

In this assignment, you are only responsible for setting up the objects and the lights. We take care of setting up the camera for you; your implementation in CamtransCamera will be used automatically by the support code.

Hint: You should use the setLight and applyMaterial functions in OpenGLScene, which take care of a lot of tricky OpenGL stuff for you.
2.2 Parsing the Scene Graph

When a scene file is opened, the support code instantiates a new SceneviewScene (which you will be implementing in this assignment) and calls the static Scene::parse method, passing an instance of the CS123XmlSceneParser. At this stage, the parser has already parsed the scene. You can get the root node of the scene graph by calling parser->getRootNode(). A pointer to a newly-allocated instance of SceneviewScene is also passed into Scene::parse.

In Scene::parse, you should set up the global data for the scene, as well as the lighting. You should also traverse the parse tree and add all the objects in the parse tree to your scene. You do not need to set up the camera in this method; this is already handled for you. As a design tip, we are providing stencils for Scene::setGlobal, Scene::addPrimitive, Scene::addLight functions. You may implement these functions if you want to use them.

Because the Scene::parse function is designed to be a generic scene setting-up function, it should not do anything specific to an OpenGL scene. Ideally, you will be able to reuse Scene::parse for Intersect and Ray without having to make any modifications. Think about how you can take advantage of polymorphism and virtual functions. Specifically, you should not need to make any calls to OpenGL in the Scene::parse function.

Note that each CS123SceneNode may have multiple modeling transformations. You should multiply the matrices for each of these modeling transformations in the order in which they appear in the scene file. Do not enforce any other ordering of the transformations (e.g. scale, rotate, translate).

Additional documentation for the different types of transformations can be found in the source code.

A few additional tips:

- Be careful about the order in which you multiply the composite modeling matrices when traversing the scene graph. Refer to the algo if you are having trouble.
- Resources owned by CS123XmlSceneParser are freed shortly after the Scene::parse method returns. Therefore, you should make copies of CS123ScenePrimitive objects (and any other objects inside the parse tree) rather than just copying pointers.

2.3 Rendering the Scene

The rendering of your scene starts in the paintGL method found in Canvas3D. This method will invoke the render method in your SceneviewScene. Your SceneviewScene will call context->getCamera() to get the current camera from the canvas, load the appropriate matrices into the shader, and then call renderGeometry. This is all done for you.

Your job is to fill in the renderGeometry method to render the geometry of your scene using OpenGL. You will have to traverse the data structure in your SceneviewScene and render all the shapes by invoking your Shapes code. You can use glUniformMatrix4fv to pass the corresponding transformation matrices from your scene graph before rendering the geometry. (Remember Lab 2?)

2.4 Materials and Global Lighting

You are not required to use any of the material properties of scene objects except the “diffuse” and “ambient” color properties. These are the only properties that determines the color of an object in Sceneview. You should make use of the CS123SceneMaterial struct and use the applyMaterial function in the support code, which wraps a few OpenGL calls for your convenience. This will handle using the material color properties for you. Remember to multiply by $k_d$ and $k_a$!

You will also find that the CS123SceneParser returns some global lighting information including lighting constants. For the purposes of Sceneview you only need to apply the diffuse lighting constant to your diffuse color, but feel free to store everything in your scene graph. They’ll come in handy in future projects.

3 Grading

Be sure to test your program on as many scenes as possible! Also, this is the time to fix buggy Shapes and Camtrans code. You’ll lose points again on Sceneview if your Shapes and/or Camtrans code doesn’t work. Also, be sure to pay particular attention to the following areas of your implementation (although this is a non-exhaustive list):

- Does your program leak memory?
- Is your implementation efficient?
- Do all the scenes load correctly?
- Is the lighting correct?
- Is the camera correct?
4 Extra Credit and Half-Credit

4.1 [Optional] Extra Credit

Remember that half-credit requirements count as extra credit if you are not enrolled in the half-credit course.

Texture mapping and other more advanced effects are not part of the basic requirements for this assignment. You may implement them for extra credit. Extra credit will also be awarded for particularly interesting scenes.

4.2 Half-Credit Requirements

Remember that half-credit requirements count as extra credit if you are not enrolled in the half-credit course.

1. You must implement OpenGL texture mapping in your scenes. Your implementation must produce scenes that match the demo. Scenes in the /course/cs123/data/scenes/ray directory have textures.

2. You must also implement adaptive level of detail based on the complexity of the scene. This means that for scenes with many objects in them, each object should be rendered with fewer triangles than if the scene contained a small number of objects. Look up "level of detail" for more information. Note: many of the examples you will find discuss distance-based or view-dependent level of detail, which you are not required to do!

5 Final Notes

You will be using this code extensively for the rest of the assignments this semester. Please be careful about your design, as it is important. See a TA with any questions. The best way to test your Sceneview is to make very simple scene files that isolate particular things. Remember, you are allowed to share your scene files with the class! You can put any scene files you create in the /course/cs123/data/scenes/shared directory (the folder is world-writable). Extra credit will be awarded for good, compelling scenes.

To hand in your assignment, type cs123_handin sceneview in the directory containing CS123.pro.

6 Troubleshooting/FAQ

Please check these FAQ entries before coming to TA hours. You should be able to answer any of the questions in the appropriate section before a TA will help you. If you come to hours unprepared, you may be asked to come back the next day, after you have prepared sufficiently.

6.1 Scene too bright

Did you remember to multiply the ambient object color by $k_a$, and the diffuse object color by $k_d$? Are your normals correct? Are they normalized? Is the homogeneous coordinate correct? A fault in any one of these can cause the problem.

Your Camtrans may also be buggy. Look back at Camtrans and make sure your shapes all look correct, including the normals.

Finally, your lighting may be incorrect. Make sure the lights are in the correct place and all required attributes have been set. Look around the support code we have given you and try to find some useful examples for how to do lighting in OpenGL (there definitely are some in there!).

6.2 Objects incorrectly positioned

Make sure the homogeneous coordinates of all your vectors and points are correct (particularly those loaded from the CS123SceneParser). Generous use of assert statements will help isolate these problems.

Are you multiplying your matrices in the correct order? Are your transformation matrices correct? Is your camera in the right place? Start out by loading simpler scene files, such as ball.xml (which contains a single blue ball at the origin). Create other simple test scene files, and go from there.

Note: While OpenGl expects matrices in column-major format and the scene files store matrices in row-major format, the parser will take care of the conversion automatically - you should not have to worry about transposing them.

6.3 Why is the background gray?

The background is gray if you have “draw wireframe” turned on, because if it was black you might not be able to see the black wires against the black background. If you turn off “draw wireframe” the background will change to black.