Algorithm 3: Sceneview

Comp175: Introduction to Computer Graphics – Spring 2020

Algorithm due: **Friday February 28th** at 11:59am
Project due: **Monday March 9th** at 11:59pm

1 Introduction

In order to visualize a complex 3-dimensional scene, thousands of tiny triangles must be drawn to the screen. It would be incredibly tedious 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'll use an existing support library to parse an XML file containing information about the scene to be rendered, 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 (the data structure in which you stored the scene info) 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 the 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 `Camera` in order to accomplish these tasks.

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. This means that you are required to "flatten" the scene graph into an array-like structure (e.g fixed-length arrays or vectors, but not linked lists). As such, 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.

As usual, there is a good chunk of existing support code for this assignment. In addition, your code for this assignment will use your `Camera` and `Shapes` code. If your `Camera` or `Shapes` code isn't working (or is leaking memory), you will need to fix it!

For this assignment, you will be doing most of your work in `MyGLCanvas.cpp`. However, be sure to understand how the parser works, and the internal support structure that will help you figure out how to get information out of the parser.

You can find a whole bunch of test XML scenefiles bundled with the support code.

2.1 Support Functions in MyGLCanvas.cpp

There are a few functions in MyGLCanvas.cpp that you should utilize. One is the `applyMaterial` function, the other is the `setLight` function. As the names suggest, these functions help you set up the OpenGL context given information regarding the lights and the object materials.

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 `Camera` will be used automatically by the support code.

2.2 Parsing the Scene Graph

As noted earlier, the Parser for the XML files is written for you. The interface that you should be aware of are: `SceneParser.h/cpp` and `SceneData.h`. As of the low-level XML parser files (that is, `tinyxml, tinystr, tinyxmlerror, tinyxmlparser`), you should just ignore. There should be no reason for you to need to access these files directly.

2.3 Rendering the Scene

Your job is to fill in the methods to render the geometry of your scene using OpenGL. In particular, look for the words `TODO` in the `drawScene` function. You will have to traverse the data structure returned by SceneParser and render all the shapes by invoking your `Shapes` code. You will also need to use `glPushMatrix`, `glPopMatrix`, and `glLoadMatrix` to load the corresponding transformation matrices from your scene graph before rendering the geometry.

2.4 Generating a Scene

Lastly, as part of your assignment, you are required to create an XML file manually! Be creative! Use your imagination to come up with a complex scene entirely made up of the primitives (Cube, Sphere, Cone, and Cylinder). When you are done, put your XML file in `/comp/175/pub/a3` with your name on it (e.g., RemcoChang_Scene.xml). This directory is completely public, meaning that you should also check out this directory to render files that other students have come up with! Note that because of the "sharing" aspect of this requirement, you should avoid using special shapes in your XML scene. Also, when you put the file in the directory, please make sure that permissions are set correctly so that the file is readable by all (i.e. do a `chmod a+r RemcoChang_Scene.xml`).

Lastly, you need to submit your XML file as part of your final submission as well. In addition, in the submitted version, if you would like to make use of your special shape, do feel free to show off!

3 Final Notes

You will be using this code extensively for the rest of the assignments this semester. Please be careful about your design, it is important. Start early and come see us with any questions. The best way to test your `Sceneview` is to make very simple scene files that isolate particular things.

4 How to Submit

Complete the algorithm portion of this assignment with your teammate. You may use a calculator or computer algebra system. All your answers should be given in simplest form. When a numerical answer is required, provide a reduced fraction (i.e. 1/3) or at least three decimal places (i.e. 0.333). Show all work.

For the project portion of this assignment, you are encouraged to discuss your strategies with your classmates. However, all team must turn in ORIGINAL WORK, and any collaboration or references outside of your team must be cited appropriately in the header of your submission.

Hand in the assignment using the following commands:

- Algorithm: `provide comp175 a3-alg`
- Project code: `provide comp175 a3`

5 Extra Credit

Come up with a scene that blows my mind. See the chess.xml file for example. That's a pretty cool scene made up of very simple geometry. Try coming up with your own design that will get passed down to future students who take this class!

6 FAQ

6.1 Z-Fighting

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 nondeterministic 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.

6.2 Null vector in glm::rotate

One thing to be careful about is the `glm::rotate` function.

```
glm::mat4 resultMat = glm::rotate(
  glm::mat4 compositeMat, float radians,
  glm::vec3 rotVec)
```

If `rotVec` happens to be a null vector, `resultMat` will be undefined. In a lot of the examples (especially those in the `data/tufts/` directory), `rotVec` could be null. So don't forget to double check!

One way to check for the null vector is to use `glm::length` to make sure that the length of the vector is greater than `EPSILON`.