Assignment 2: Scene

CS 175: Introduction to Computer Graphics – Fall 2024

Algorithm due: **Friday Oct 11th** at 11:59am (noon)
Project due: **Wednesday Oct 16th** at 11:59pm (midnight)

1 Notice

The timing of this assignment is a little unusual. Instead of doing the algorithm worksheet first, the algorithm worksheet will be released a week later. This is because this assingment has two parts. Part 1 is on Camera, Part 2 is on Scene Graph. However, instead of giving the Camera lecture first, we're doing the Scene Graph lecture first. Because of this unusual setup, you have a couple of extra days to complete this assignment (note the due date of Wednesday at midnight).

For your Assignment 2, you should start working on Part 2 (Scene Graph) first. Note that without a proper camera, the system will just assume a camera that's positioned at (0, 0, 0) and looking down the negative z-axis (using orthographic / parallel projection). See Figures 1 and 2 for examples of loading a scene file without a working camera. You can use this test case (Text\robot.xml) to make sure that your Scene Graph is working correctly. Once you have implemented Part 1 (Camera), you'll then be able to load a scene graph with the camera oriented and calibrated properly.

2 Introduction

When rendering a three-dimensional scene, you need to go through several stages. The first step is to take the objects in the scene and break them into triangles. You did that in Assignment 1. The next step is to define how the triangulated objects in the three-dimensional scene are displayed on the two-dimensional screen. This is accomplished through the use of a camera transformation: a matrix that you apply to a point in three-dimensional space to find its projection on a two-dimensional plane. While it is possible to position everything in the scene so that all the camera matrix needs to do is flatten the scene, the camera transformation usually incorporates handling where the camera is located and how it is oriented as well. Lastly, in a scene where there are multiple objects, we can organize them in a hierarhical fashion in a "scene file." Parsing the scene file, traversing the hierarchy, and rendering each object in its correct position (and with the appropriate lighting and surface properties) will result in a rendered scene.

As such, there are two components to this assignment. First, you will need to implement a Camera. Second, you will implement a hierarchical scene parser/traversal function to render a scene.

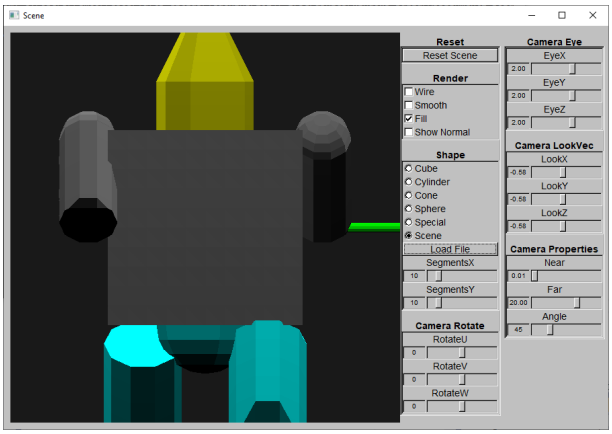

Figure 1: Loading Test\robot.xml without a working camera

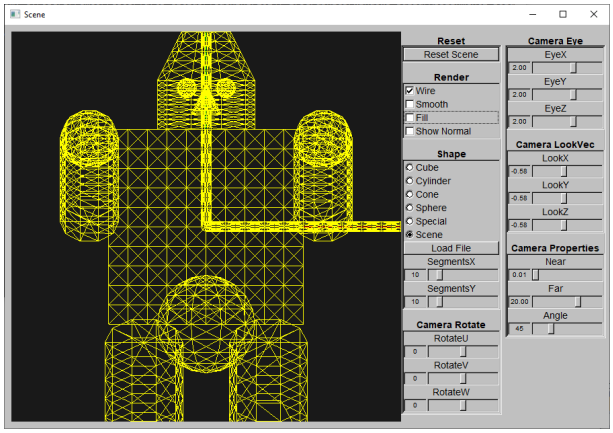

Figure 2: Loading Test\robot.xml without a working camera, rendered in Wireframe mode

For Part I (the Camera), it is entirely possible to simply throw together a camera matrix which you can use as your all-purpose transformation for any three-dimensional scene you may wish to view. All that you would have to do is make sure you position your objects such that they fall within the standard view volume. But this is tedious and inflexible. What happens if you create a scene, and decide that you want to look at it from a slightly different angle or position? You would have to go through and re-position everything to fit your generic camera transformation. Do this often enough and before long you would really wish you had decided to become a sailing instructor instead of coming to Tufts and studying computer science.

For this part of the assignment, you will be writing a Camera class that provides all the methods for almost all the adjustments that one could perform on a camera. The camera represents a **perspective** transformation. It will be your job to implement the functionality behind those methods. Once that has been completed, you will possess all the tools needed to handle displaying three-dimensional objects oriented in any way and viewed from any position.

For Part II (Scene Traversal), 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 flattened list (e.g., an array) 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 will be making use of your code from Assignment 1 (`Shapes`) to accomplish these tasks. In particular, you will need to copy-and-paste the CPP and H files you wrote for rendering each of the basic shapes (Cube, Cylinder, Cone, and Sphere) from Assignment 1 to this assignment.

2.1 Demo

As usual, we have implemented the functionality you are required to implement in this assignment.

The sliders control the following:

- RotateU, RotateV, RotateW refer to controlling the camera's roll, pitch, and yaw (think of the orientation of how you would hold a camera).
- The position of the camera (EyeX, EyeY, EyeZ)
- The look vector of the camera (LookX, LookY, LookZ)
- The distances to the near and far clipping planes
- The view angle (Angle) of the camera (in degrees)

Note that the rest of the program (and the code) is built upon your Assignment 1. However, in addition to the basic objects, there is now a button to load a scene file. Selecting the "scene" radio button and loading an xml scene file will render the scene from the associated file. Part II will be to implement this functionality.

3 Part I (Camera) Requirements

3.1 Code from Assignment 1

You should copy-and-paste your code from Assignment 1 to this assignment to render the basic shapes (Cube, Cylinder, Cone, Sphere). Note that the definition of the Shape.h file has changed – now there is a new possible value for `OBJ_TYPE` called `SHAPE_SCENE`. That value is used to denote rendering a scene file (more on this in Part II below).

3.2 Linear Algebra

Your camera package depends heavily on linear algebra. In the past, students implemented an entire linear algebra package from scratch in this assignment! However, we believe you have better things to do than writing vector addition functions in C++. Therefore, for this class, we're using the GLM library.

3.3 Testing your Linear Algebra

Before you implement your Camera, you should test your transformation matrix generators. We leave it up to you to determine how to do this. Either way, make sure that you are comfortable and familiar with the use of GLM to perform linear algebra operations. You will be using it extensively.

3.4 Camera

You will need to complete the `Camera.cpp` and `Camera.h` files. Your camera must support:

- Maintaining a matrix that implements the perspective transformation
- Setting the camera's absolute position and orientation given an eye point, look vector, and up vector
- Setting the camera's view (height) angle and aspect ratio (aspect ratio is determined through setting the screen's width and height).
- Translating the camera in world space
- Rotating the camera about one of the axes in its own virtual coordinate system

- Setting the near and far clipping planes
- Having the ability to, at any point, spit out the current eye point, look vector, up vector, view (height) angle, aspect ratio (screen width ratio), and the perspective and model view matrices.

Finally, while not all the functions are called explicitly in the program, you are still required to fill in all of the empty functions.

3.5 ModelView and Projection Matrices

OpenGL requires two separate transformation matrices to place objects in their correct locations. The first, the `modelview` matrix, positions and orients your camera relative to the scene[1]. The second, the `projection` matrix, is responsible for projecting the world onto the film plane so it can be displayed on your screen. In `Camera`, this is your responsibility. You must be able to provide the correct `projection` and `modelview` matrices when your `getProjectionMatrix` or `getModelviewMatrix` functions are called from main.

4 Part II (Scene Traversal) Requirements

You are required to implement the funcionality that takes in an XML file to build a scene graph internally and then efficiently render the scene to the display using calls to OpenGL.

4.1 Flatten the Scene Graph into an Array

You are required to "flatten" the scene graph into an array-like structure (e.g fixed-length arrays or vectors, but not linked lists). The resulting array-like structure should contain: (1) all n objects in the scene file, and (2) for each object, its corresponding composite transform matrix[2]. The reasoning for this is that traversing the entire scene graph at 60 fps can be very expensive as the scene graph becomes large. Storing the composite matrices for all objects trades space for efficiency and will make the rendering much more efficient. You will probably want to use the STL (Standard Template Library) in various places within this part of the assignment; there are places where maps and lists will make your life much easier.

For this part, 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.

4.2 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.

4.3 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.

4.4 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.

4.5 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). You need to submit your XML file as part of your final submission as well.

5 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,

[1] Or, if you prefer, orients the world relative to your camera.

[2] Note that each object will maintain its "material" information (e.g., color, surface reflection property, etc.). This is done for you by the parser.

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 on Canvas.

6 FAQ

6.1 GLM and Matrix Order

GLM uses column-order matrices (just like OpenGL). To clarify. Given:

$$M = \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$$

When using glm, let's say that M is of type `glm::mat4`. Then:

$$\begin{aligned} M[0][0] &= a \\ M[0][1] &= b \\ M[0][2] &= c \\ M[0][3] &= d \\ M[1][0] &= e \\ &\vdots \end{aligned}$$

In other words, the ordering of the arrays in `glm::mat4` is $[col][row]$.

6.2 The Camera LookVec Sliders are Updated

When the camera rotates (i.e. when the user interacts with the RotateU, RotateV, RotateW sliders), the camera's look vector will change. You can see this behavior in the demo of A2 – when you interact with those rotation sliders, you should see the values of the look vector sliders change accordingly.

As part of the support code, in `main.cpp` lines 174-179, you will see that, in the function cameraRotateCB() – as a part of the callback to RotateU, RotateV, and RotateW – the interface will ask your camera to give an updated look vector. The values of the look vector are then used to update the values of lookXSlider, lookYSlider, and lookZSlider.

This means that if you have successfully implemented `setRotUVW()` in your Camera class but haven't implemented `getLookVector()`, your system will be buggy (and possibly crash). If this is happening, comment out the lines (174-179) until you have implemented `getLookVector()`.