CS 355 Lab #5: Implementing Simple 3D Graphics

Overview

Now that you're familiar with the basic pipeline for 3D geometry, it's time to do it yourself. In this lab you will implement the world-to-camera, projection, and viewport transformations that OpenGL did for you in the last lab.

You will use the shell you were given for Labs #2–3 and add functionality for simple 3D rendering. This is solely so that when you're done you can have all of the parts you did in one program. *You will not be graded on whether all the parts from Labs #1–3 work, so long as what you have integrates with the new 3D rendering.*

All rendering of the model should be done *on top of* anything drawn with the 2D drawing tools (if any). Think of what you draw in 2D as forming a backdrop that always stays behind the rendered model.

User Interface

In addition to the GUI elements from Labs #1–3, there is a button that toggles display of the 3D model. With that "off", the program should work as in Lab #3. With that "on", you should render the model on top of the 2D content and respond to keyboard commands for camera movement as in Lab #4. *Keyboard presses that occur while the model display is off should not cause the virtual camera to move.* (Otherwise, it's easy to move around and lose the model while not displayed.)

Camera movement uses the same keyboard commands as in Lab #4. This lab will omit orthographic projection, so you may also omit responding to the "o" and "p" keys.

Implementation Notes

You should integrate into your program all of the functionality from Lab #3, which is probably most easily done by using your previous model, view, and controller, then adding to the view and controller the new methods necessary to implement the new interface.

The controller and view code should be separate, as you did in Lab #3. Your controller should maintain the 3D camera state, and the view should query your controller to get it for drawing, much as you probably did for Lab #3 for zooming and scrolling.

Model

For this lab the 3D "model" in the model-view-controller sense is static. You *can* open new scenes by going to "`File->Open Scene`", but scenes cannot be edited like 2D shapes.

You should use your 2D shape model code from Lab #3.

Controller

In addition to the event handlers your controller provided in Lab #3, you will have three new methods:

```
public void toggle3DModelDisplay()
```

This method should toggle your internal state for whether to draw the 3D model. It should initiate a screen refresh to update with the model appropriately drawn or not drawn.

```
public void keyPressed(Iterator<Integer> iterator)
```

The main shell will call this method at regular intervals whenever a key is pressed or held down. A list of the pressed keys will be passed to it, and you should respond and update your camera state like you did in Lab #4. It should also initiate a screen refresh to update the screen with the rendered model after all of the keys have been processed. If the 3D model is not currently being shown, any key presses should be ignored.

```
public void openScene(File file)
```

The shell will call this method when the user tries to open a scene. It should simply pass the `File` parameter to `CS355Scene.open(File)`. The `CS355Scene` class will open and parse the *.scn and *.obj files and build a scene for you.

View

When asked to draw the screen, your viewer should first draw any 2D shapes that the user has drawn, moved, rotated, resized, etc. (You shouldn't have to write anything new to do this.)

After drawing the 2D background, you should then render the 3D model according to the current camera position and direction. *Moving the virtual 3D camera around does not need to make the 2D background change.*

This part is the real heart of the lab. You should render the 3D model by using 2D line-drawing commands after first determining the projected 2D locations of each 3D line's endpoints. You should determine these projected 2D locations by implementing each of the pipeline stages we've talked about:

- Convert the 3D (X, Y, Z) coordinates to 4-element $(X, Y, Z, 1)$ homogeneous coordinates.
- Build *a single matrix* that converts from world to camera coordinates. As you did in Lab #4, this is the result of concatenating a translation matrix and a rotation matrix. *You will need to implement storing the 4 $\times$ 4 matrix yourself, including routines for multiply 4-element vectors by them.*
- Apply this matrix to the 3D homogeneous world-space point to get a 3D homogeneous camera-space point.
- Build a clip matrix as discussed in class and in your textbook. Pick appropriate parameters for the zoom_x, zoom_y, near plane distance n, and far-plane distance f.
- Apply this clip matrix to the 3D homogeneous camera-space point to get 3D homogeneous points in clip space.

- Apply the clipping tests described in class and in your textbook. Reject a line if *both* points fail the same view frustum test **OR** if *either* endpoint fails the near-plane test. For this lab, we'll let Java's 2D line-drawing handing any other clipping.
- Apply perspective by normalizing the 3D homogeneous clip-space coordinate to get the $(x/w, y/w)$ location of the point in canonical screen space.
- Apply a viewport transformation to map the canonical screen space to the actual drawing space (2048×2048, with the origin in the upper left).
- Apply the same viewing transformation you use to implement zooming and scrolling of the 2D graphic objects to map from a portion of the 2048×2048 to the 512×512 screen area.
- Draw the line on the screen. (This is where rasterization of the primitive would take place, but we'll just use the familiar 2D drawing commands to do this.)

Note that you are rendering to the entire 2048×2048 area. You should be able to zoom and scroll this display as in Lab #3 independent of the camera movement and rendering. Be careful: if you start moving the camera around while scrolled, the movement of the rendered model will be counterintuitive. *For testing, I recommend zooming all the way out so that the entire drawing / rendering area is displayed, then moving around in 3D, and then seeing if you can zoom in and scroll correctly. If you wish, you can set the default zoom level to 25% so that your program starts up with this view.*

Important: For some reason, the shell's z-axis is the opposite direction as OpenGL's, so be aware that you will need to flip the z-axis on all of your Point3D's. This includes camera position and vertex positions.

Restrictions

In addition to the restrictions on all previous labs, the code in this lab has the following restriction:

- You *cannot* modify any code in the `cs355.model.scene` package.

Submitting Your Lab

To submit this lab, zip up your src directory to create a single new `src.zip` file, then submit that through Learning Suite. We will then drop that into a NetBeans project, copy in the originals for any files you are not allowed to modify (see the Restrictions section of the Lab 1 specification), do a clean build, and run it there.

If there are compile errors in your project because you modified files that you were not allowed to modify, we will grade your lab accordingly.

If you need to add any special instructions, please do so in the notes when you submit it. If you use any external source files, please make sure to include those as well.

Rubric

- Correct rendering of the wireframe house with perspective (30 points)
- Correct rendering from multiple viewpoints (30 points)
- Correct clipping (10 points)
- Correct navigation relative to current position and orientation (10 points)
- Scrolling and zooming of the 2D rendering (10 points)
- Generally correct behavior otherwise (10 points)

TOTAL: 100 points
