CS 355 Lab #5: Implementing Simple 3D Graphics

October 29, 2013

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 be given a shell to start with, which is an extension of the one you did for Labs 1–3. 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.

You will be given a new shell, similar to what you had in Lab #3 but with a few new things added.

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 with the addition of one new command: pressing the "h" key should take the camera "home" to the original position. (You are free to determine what that is, but this key should take the camera position and direction back to what it was when the program started.) This lab will omit orthographic projection, so you may also omit responding to the "o" and "p" keys.

Implementation Notes

As mentioned already, you will be given a new shell to use. You should integrate into it 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. It is recommended that you first reproduce your Lab #3 functionality with the new shell using stubs where necessary, then add the new functionality.

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 should use your 2D shape model code from Lab #3, but the viewer will also need to reference the static 3D model. You should use the `HouseModel` from Lab #4, but you are again free to build other models in that format if you wish.

Controller

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

`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.

`keyPressed`

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.

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 in 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 #3, this is the result of concatenating a translation matrix and a rotation matrix. You will need to implement the 4×4 matrix multiplication routines yourself.
- Apply this matrix to the 3D homogeneous world-space point to get a 3D homogeneous camera-space point. You will need to implement the 4×4 matrix times vector multiplication routines yourself.
- 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. Only reject a line if *both* points are outside the view frustum.
- 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 $512 \times 512 screenarea$. (Yes, you should be able to zoom and scroll this display independent of the camera movement and rendering.)
- 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.)

Submitting Your Lab

To submit this lab, again zip up your src directory to create a single new `src.zip` file, then submit that through Learning Suite. If you need to add any special instructions, you can add them there in the notes when you submit it.

Rubric

This is tentative and may be adjusted up to or during grading, but it should give you a rough breakdown for partial credit.

- To be added here

TOTAL: 80 points

Change Log

- October 29: initial version
- November 13: amended to correct the size of the drawable area