CAP 4730 Computer Graphics
Dr. Corey Toler-Franklin

Scene Manipulation
DUE: March 24th, 11:59 pm

Overview | Details | Resources | Getting Help | Submitting

Overview

You may complete this project alone or in groups of two.

The OpenGL pipeline is used to create interactive 3-D graphics. Specifically, it uses linear algebra and specialized hardware to convert 3-D scenes into 2-D raster images for display on your monitor. In this project, you will learn how to use OpenGL matrices to convert a 3D scene into a 2-D image on the screen. Matrix transformations are tools used to change the position, orientation and shape of objects in a scene. In this assignment, you will complete a short written exercise, create a scene graph and implement 3-D manipulators, selectable graphics that allow you to interactively control the position of objects. Source code is provided as a framework to help you. You may change/modify this framework however you like to complete the assignment. This assignment focuses on matrix manipulation in the traditional pipeline while your next assignment focuses on shader programing and the modern pipeline. Your work will be evaluated based on the following criteria (1) written answers 10% (2) source code completion and correctness 20% (3) render tree traversal 20% (4) camera matrix updates 20% (5) matrix transformations and manipulators 20% (6) written report 10%.

Details

Matrix Stack Written Exercise

1. The matrix transformation pipeline transforms points from *object space* to *world space* to *camera space* to a *canonical view volume*, and finally to *screen space*.

   (a) In $1-2$ sentences each, explain what happens in each of these pipeline spaces.

2. Assume a perspective projection, and the following parameters (Ch. 7 naming conventions):

   - Bounding planes for the orthographic view volume $[l,r]\times[b,t]\times[f,n]=[-5,5]\times[-5,5]\times[-30,-5]$
   - Screen size: $nx = 1000$ pixels, $ny = 500$ pixels
   - Camera: $e=(1,1,0), g=(0,-1,1), up=(1,1,0)$
   - Point of interest: $p=(5,-20,15)$

   (a) Starting from the world space point p, write the matrix required at each stage to transform the point p from its world space position to its screen projection. You should have three matrices that achieve the three transformations (world to camera, camera to canonical view volume, canonical view volume to screen space). Be sure to use the data provided to assemble the specific entries in your matrices.

   (b) Finally, compute the pixel coordinate of the world space point p.

Getting Started with The Framework

Extract the file proj3_scenemanip.zip to your local folder. The source code framework is in the *src* folder. Datasets (meshes, textures and scenes) are found in the *data* folder. Other helpful materials are in the *docs* folder. To build the proj3_scenemanip.zip application on linux, type *make* in the *src/mainsrc* folder. This also generates a visual studio c++ solution file. Use any platform but make sure your code compiles and runs on Ubuntu Virtual Box before you submit (see *docs* for instructions). Included in the package is libigl, a geometry processing library, and libst (by Syoyo Fujita) and librt from proj2_raytracer. The program is executed from the main function in the file src/mainsrc/main.cpp. **Search the project files for *TO DO: Proj3_sceneManip*. In the following sections, you will insert or alter code at these locations to complete the assignment.** It is suggested that you progress through the assignment in the order presented.

Render Tree Transversal

The *Scene* class stores objects in a tree structure. The implementation of the scene data structure in this assignment is slightly different from the one you were provided in project 2. In this case, objects are added to the scene by adding nodes to the scene class. In the scene graph, the root node has no parents. Each node in the tree holds a scene object. *LightNode*, *CameraNode*, *GeometryNode*, *TriangleMeshNode* and *TransformNode* are all subclasses of the *SceneNode* class which store lights, cameras, meshes and transforms respectively. Recall from lecture that groups are nodes with children. Transformations at the group level are applied to all children in the group. The current group's transform is an accumulation of all transforms in the path up to this group. Note that manipulators or draggable graphical interfaces (discussed later in the assignment) may be added to your object if you click on it. If this happens, for now, just toggle the *e* key until the manipulator disappears.

The initial scene is already created for you. The function *initializeScene* in main.cpp loads a cube triangle mesh in the center of the scene. Your first job is to transform the position of this object in the scene by initializing its parent *TransformNode* to something other than the IdentityMatrix(). To see the transformed results, you will have to complete the function *PropogateTransforms* that traverses the scene tree and propagates group transformations to child nodes. The order of matrix multiplication determines whether transformations occur locally, or in the parent coordinate system. The meshes in your start-up project have stored surface normals. Remember that for geometry with stored surface normals, you must preserve surface normal orientations correctly. **Take a screen shot of your transformed object** after *PropogateTransforms*.

Camera Matrix

To render the scene, you need a viewing transformation (camera transformation) to map world space points to camera space, and a projection transformation that maps the camera space to the conical view volume. Familiarize yourself with the camera controls provided. The camera orientation and position is updated by the mouse callback functions located in the main.cpp file. The third mouse button is used to zoom the camera in and out. You must modify the camera basis in the *Camera* class to implement the rest of its behavior. Examine the *Camera* class' *Orbit* and *Strafe* functions. The Strafe function moves the camera left and right and up and down by mapping changes in mouse motion to changes in the camera position. The *Orbit* function should either rotate a the camera about the viewpoint (*FLY* mode) or about an object it is inspecting (*ORBIT* mode). Press c to toggle between these modes and implement *Camera:: RotateFly* and *Camera:: RotateOrbit*. As rotation matrices are not cumulative, you must handle ambiguous mouse motions that involve multiple axes. Once you update the camera basis vectors, calls in main.cpp use OpenGL to create the camera view matrix. This call will make sure the aspect ratio of the camera viewport size is adjusted properly in relation to the 2-D screen size. **Take screen shots of your camera translation (Strafe), rotation and fly mode**.

Manipulators

Manipulators are user controllable 3-D graphics that map movements of an input device in 2-D screen space to 3-D matrix transformations in the scene. Figure 1 (right) shows an example of a manipulator. Figure 1 (left) diagrams the motion associated with the manipulator parts. For example, the red manipulator arrow translates the object forward and backward while the blue arrow translates it up and down. The circles are used to control rotations of the object about each axis. To conceptualize how this works, see the diagrams in the file docs/manipulators.pdf. For translation for example, the 2-D mouse position determines the position along the ray. Examples of similar manipulators are found here https://www.youtube.com/watch?v=KprOKi7OY18.

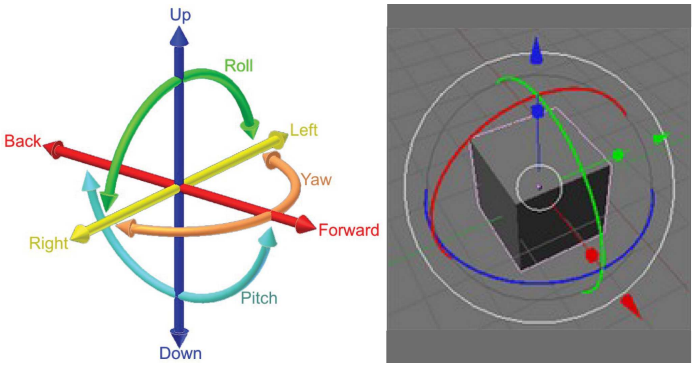

Figure 1: 3-D Manipulators

Familiarize yourself with the keypress and navigation tools provided. A left mouse click on the object will add the graphical manipulator interface. Toggling the *e* key switches between the no interface state and one with full, translation-only and rotation-only versions. You are responsible for implementing the remaining camera and manipulator transformations as follows.

Complete the *Rotate* and *Translate* functions in the *Scene* class to create the manipulator shown in Figure 1. There are two manipulation modes. One updates matrix transformations locally in the object coordinate system, while the other updates transformations in relation to the parent coordinate system. This is controlled by the order of matrix multiplications. Use the *k* key to toggle manipulation modes. When you select an object in *local* or *parent* mode, the manipulator appears centered around the object. Drag the manipulator geometry to update matrix transformations in proportion to the amount of mouse movement. Code is already implemented to select objects, add manipulators, detect when manipulators are selected and call your code when the user drags the manipulator parts. You will compute the changes to the matrix transformations of the object.

Rotation To control rotation in each of the three axes, the user selects and rotates one of three intersecting circles perpendicular to the axis of rotation. When the outer circle is selected, it rotates the object about the z-axis without changing the object tilt angle. To implement this functionality, you will complete the *Scene::Rotate* function for the given rotation axis. Tracking mouse positions for this manipulator is non-trivial. Therefore, you may map the vertical mouse motion directly to the rotation angle. Update the rotation angle by some reasonable constant value. **Take screen shots of your object with the manipulator attached showing how it has been rotated**.

Translation To control translation in a given direction, select and drag the red, green or blue arrow parallel to the x, y and z axes respectively. Note that **you will have to remove the other manipulator parts (*e* key toggle) so that the translation manipulator is selectable**. Recall the definition of a *ray* from proj2_raytracer. The translation manipulator will translate the object along a ray from some original position e along an interval t. To implement the *Scene::Translate* function, you should map the change in the mouse motion to the coordinates of the selected transform. **Take screen shots of your object with the manipulator attached showing how it has been translated**.

Geometry Finally, upgrade your code to handle more complex models than the default model initialized in *InitializeScene*. You have several options. Your model may come from the Data directory,

be created in your code, or consist of more than one simple object (like a cube) acting as a group. You also have the option to add support for more than one triangle mesh group node. **Take screen shots of your object with the manipulator attached showing it being moved and manipulated**.

To summarize, screen shots should demonstrate (1) PropogateTransforms, (2) camera translation, orbit and fly. (3) manipulator translation and rotation and (4) a more complex model being manipulated (rotated, translated). Include these, along with a description of how you completed the implementation in your report. Section *Submitting* lists what to submit.

Resources

The openGL Programming Guide, available online https://www.opengl.org/sdk/ is a good reference. Appendix A also provides information about GLUT and GLEW. Chapter 12 of the textbook, Fundamentals in Computer Graphics, covers meshes. Information about the *.obj* file format is in the *docs* folder.

If you are not in the CISE department, and would like computer lab access, you can register for a CISE account at https://www.cise.ufl.edu/help/account. The source packet proj1_mesh.zip will run on machines in the computer labs on the first floor of the CSE building. Labs are open 24 hours (see https://www.cise.ufl.edu/help/access). **Remember to back-up your work regularly!!!** Use version control to store your work. DO NOT PUBLISIZE SOLUTIONS.

Getting Help

Source Code Please do not re-invent the wheel. Use the source code framework (and comments) and review the notes in the docs folder.

Discussion Group Post questions to Canvas (everyone benefits in this case), or send me an email at ctoler@cise.ufl.edu. I will check both daily.

Office Hours Stop by my office, CSE 332 (or lab CSE 319) during office hours MWF 11:45am to 12:30pm.

Collaborating Work independently. Remember to always credit outside sources you use in your code. University policies on academic integrity must be followed.

Submitting

Upload your proj3_4 _OpenGL.zip file to Canvas. It should include:

- Source code with make file. Before you submit, build and run on *thunder* or *storm*.
- One written report that includes:
  - answers to written questions
  - screen shot demonstrating PropogateTransforms
  - screen shots demonstrating manipulator translation and rotation
  - screen shots demonstrating camera translation, orbit and fly
  - screen shot of a more complex model being manipulated (rotated, translated)
  - a few lines that explain your implementation
  - anything you would like us to know (how to run your code, bugs, difficulties)