Assignment 3: Basics of 3D Graphics

CS 4600 Computer Graphics
Fall 2018
Ladislav Kavan

The goal of this assignment is to code up an interesting 3D graphical application using OpenGL. You will display with the famous Utah teapot model served as a standard reference object in 3D computer graphics. Your program will incorporates shading, animation and dolly zoom. This is an individual assignment, i.e., you have to work independently. All information needed to complete this homework is covered in the lectures and discussed at our Canvas Discussion Boards. You shouldn't have to use any textbooks or online resources (except for Visual Studio and OpenGL documentation), but if you choose to do so, you must reference these resources in in your final submission. It is strictly prohibited to reuse code or fragments of code from textbooks, online resources or other students -- in this course this is considered as academic misconduct (https://www.cs.utah.edu/academic-misconduct/). It is fine to use material from lectures, lecture slides and online documentation of OpenGL functions (and C++ / Visual Studio if necessary), but please add references (e.g. URL for online help) in your PDF writeup (see Submission Instructions below).

The framework code is written in C++ with the following dependencies:

- OpenGL 1.0
- C++ STL
- OpenGL Extension Wrangler Library (GLEW)
- GLFW3

The recommended IDE is Visual Studio 2017 Community Edition, which is available free of charge for educational purposes. The framework code provides precompiled dependencies for Visual Studio 2017. If you choose to use a different platform or IDE version it is your responsibility to build the dependencies and get the project to work. The provided source code, after being successfully compiled, linked, and executed, should display a static teapot with constant gray color (no shading).

The assignment consists of three parts: normal computation, view matrix computation and projection computation. All three parts should be implemented in the *main.cpp* file using specified subroutines. No other source code / dependencies / libraries are needed or allowed for this assignment.

Introduction

This assignment uses the very basic, yet easy to learn fixed function pipeline of OpenGL. In modern OpenGL, using the programmable pipeline gives more freedom to the programmer, but it also makes it more difficult to understand. OpenGL is a state machine. Once you set a state anywhere in the code, which runs in a thread of an OpenGL context, it remains in that state until set otherwise. The OpenGL states can be changed by calling procedures with prefix “gl”, such as *glEnable, glMatrixMode, glColor3f*, etc.

When the program starts and the main function is called, a new window with an OpenGL context is initialized using the GLFW library. The program continues by changing a few states of OpenGL, i.e. enabling lighting, setting background color and finally setting the projection matrix. Next, the Utah teapot mesh is loaded from a file TEAPOT.OBJ into a std::vector of floats, *g_meshVertices*, so that it contains the sequence of vertex coordinates $(x_1, y_1, z_1, x_2, y_2, z_2, \ldots)$. Vertex indices for each triangle of the mesh are also stored sequentially in a std::vector of integers, *g_meshIndices*, as $(a_1, b_1, c_1, a_2, b_2, c_2, \ldots)$, where a_1, b_1, c_1 are vertex indices of the first triangle and so on. Finally, a rendering loop is called. This loop sets the OpenGL ModelView matrix, renders the teapot using OpenGL immediate mode (i.e. by calling *glBegin, glVertex3f,* and *glEnd*) and checks for I/O events until ESC key is pressed or the window is closed. But do not worry too much about the technical details. Instead, let’s code up some 3D graphics!

1 Compute Normals (33 points)

The goal of this part is to compute normal vectors for each vertex of the Utah teapot mesh in order to get basic lighting to work. OpenGL fixed function pipeline uses Blinn–Phong reflection model that describes the way a surface reflects light as a combination of an ambient, diffuse, and specular reflection (we will cover reflection models later in this course). In this task, the main thing is that this reflection model requires a normal of a surface point at each vertex in order to compute the color of reflected light. Lighting enhances realism of the rendering significantly as shown in the figure below:

(a) Default normals (b) Color coded normal directions (c) Correct normals

Normals of each vertex are stored sequentially in a std::vector of floats, *g_meshNormals,* in order of the vertices, $(n_{x1}, n_{y1}, n_{z1}, n_{x2}, n_{y2}, n_{z2}, \ldots)$. If a vertex is shared by multiple triangles, the normal for such vertex is an average of all surface normals of all triangles sharing this vertex. All normal vectors should be normalized, i.e. length(normal) = 1. Your task is to complete the *computeNormals* function.

2 Make the teapot rotate (33 points)

Your second task is to make the teapot rotate about the vertical axis, as shown in this figure:

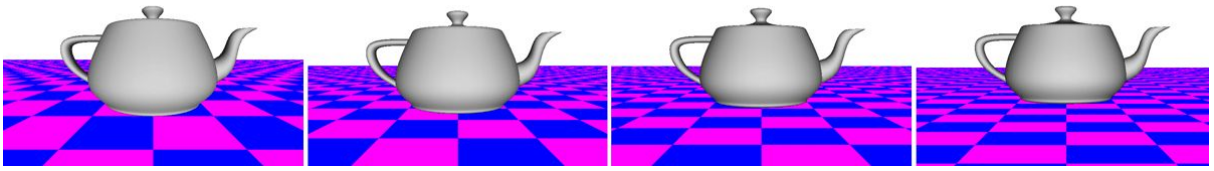
Rotating Utah teapot

In OpenGL, this can be done using a ModelView matrix, which describes the relative transformation between the model (teapot) and the camera. There are several ways to understand this, but perhaps the easiest way is to remember that the OpenGL camera is centered at [0,0,0], with x-axis (i.e. [1,0,0]) pointing to the right, y-axis ([0,1,0]) pointing up and z-axis ([0,0,1]) pointing *behind* the camera. The latter is slightly counter-intuitive, but it is like that: the camera is looking in the direction of the *negative* z-axis, i.e. [0,0,-1].

The ModelView matrix can be understood as a (homogeneous) matrix which is used to transform the coordinates of your object into the camera space. If you set this matrix to identity, you will see nothing because the camera is exactly inside the teapot. So the provided framework translates the teapot -4.5 units along the z-axis (controlled by global variable *distance*), so the teapot is nicely visible to the OpenGL camera (which is, again, pointing in the negative z-axis). Your task is to modify the function *updateModelViewMatrix()* in such a way that the teapot starts to rotate on the screen about the y-axis. You may want to use the function *getTime* to query the current system time.

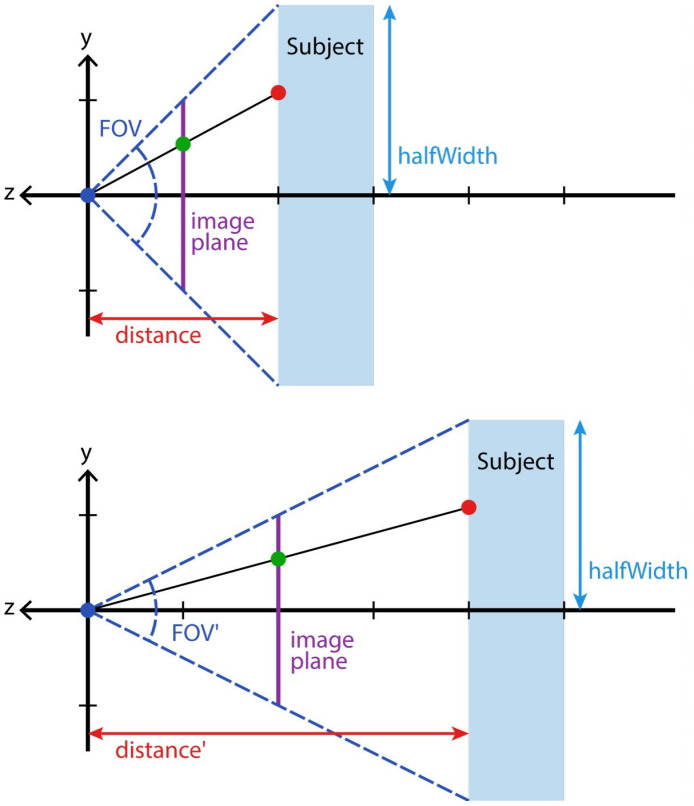
3 Dolly Zoom Effect and Orthogonal Projection (34 points)

In this part, first, you will implement an interesting effect of perspective projection called “dolly zoom”, and then, you will experiment with orthographic projection. This task helps you better understand the concepts behind perspective and orthographic projections. In order to change the projected size of our object under perspective projection, we can either change the field of view (FOV) or move (“dolly”) the camera toward or away from the subject. The “dolly zoom” effect is achieved by adjusting the FOV while moving the camera at the same time in order to keep the size of the object approximately the same after perspective projection, as shown in the figure below:



Sample frames showing the dolly zoom effect

Considering the figure below, we can explain the math behind the dolly zoom effect. In order to achieve the effect, first we need to place the subject (blue rectangle) in a desired distance (red arrow) from the camera (the center of the camera is in the origin):



Two different states in dolly zoom

Note that moving the subject in some direction is equivalent to moving the camera in the opposite direction, so we can consider the camera is fixed at the origin. For a given FOV angle, we can compute the corresponding width of the visible part of the scene (*halfWidth* shown in the figure) as below:

$$halfWidth = distance * tan(FOV/2) \qquad (1)$$

Note that the “tan” function assumes that angles are in radians (not degrees). The idea is to keep the width of the visible part of the object the same while zooming. As you can see in the figure, the width in the image plane (purple vertical line) remains the same after zooming. Notice the red point on the subject is projected to the same area (green point) on the image plane in both states.

Once we have computed the desired width (*halfWidth*) using the initial distance and FOV, we can change the FOV over time and update the distance like this:

$$distance' = halfWidth / tan(FOV'/2) \qquad (2)$$

In the equation above, *distance'* is the new distance of the subject from the camera in *z* direction, *FOV'* is the new FOV (changed over time), and *halfWdith* is computed according to Equation 1.

Your task is to first compute *halfWidth* with the initial FOV and *distance* by implementing Equation 1 in C++. Next, you need to complete the part inside the condition for dolly zoom in the function *togglePerspective()* to change the FOV over time and compute the new distance using Equation 2. You may use *getTime()* to periodically change the FOV. In order to keep the subject visible in your program, you may want to make sure the *distance* remains positive and in a good range of *zNear* and *zFar* in the *gluPerspective()* setup. To better see the effect, a piece of code to render a checkerboard plane is provided for you in the framework and you can show/hide it using the key 'C'. You can also enable/disable the dolly zoom by pressing the key 'D' in the framework.

You last job is to code up orthographic projection using OpenGL inside the function *togglePerspective()* where orthographic mode is activated. You can switch between perspective (key 'P') and orthographic (key 'O') projections to see the difference.

4 Extra Credit (up to 20 bonus points at instructor's discretion)

When you're finished with Task 3, you might notice the rotation axis is not perfectly aligned with the teapot. This is because the teapot rotates about its center of mass, which also accounts for the handle and the spout. The animation would be more visually pleasing if we make the teapot rotate about the center of its body *only*. You can also make the teapot stand still when the program runs, and gradually spin it in a smooth transition, as if the teapot was rotated by physically-realistic torque (force/torque always needs time to translate into velocity, it never happens on a dime; no matter how strong your car brakes are, you have to apply them *before* the car is in contact with a wall). In order to have more controls over the animation, you can write code to pause/resume the spinning of the teapot (it should start again at the same angle where it paused) or pause/resume the dolly zoom effect (it should start again at the same stage of zooming where it paused). The key callback is provided in the framework for you (you can use the key 'S' to toggle the spinning and the key 'D' to toggle the dolly zoom).

If you are curious what else we can do with normals, you can also experiment with basic OpenGL shading models (i.e. how colors of the vertices are interpolated during rasterization). The default shading model is *GL_SMOOTH* which is an implementation of Gouraud shading. There's also a *GL_FLAT* shading model which you can set using the *glShadeModel* function. You can also try to change the teapot's color. A particularly creative real-time rendering or animation of the Utah teapot will be especially appreciated!

5 Submission

When you're finished with Tasks 1,2 and 3 (and possibly the optional Task 4), you'll need to submit the following:

- Source code (you should only modify main.cpp so that's all we need)
- PDF document describing what you did, ideally with several screenshots; if you used any textbooks or online resources that may have inspired your way of thinking about the assignment, you must references these resources in this document. Please reference also OpenGL or C++ documentation if you used them, it is a good practice to give credit to external resources. No need to reference the lecture materials or slides.
- The PDF document should also include a link (URL) to a video which you need to upload on your YouTube channel. The video will be a screen capture of your final rotating teapot. There are many free screen capture utilities that can help you with this.

Please pack the source code (main.cpp) and the PDF document to a single ZIP file named Lastname_Firstname_HW3.zip.

Please submit this ZIP file via Canvas.