CS334/ECE30834: Assignment #1 - Project it!
Linear Algebra + Perspective/Orthographic Projections

Out: January 24, 2022
Back/Due: February 7, 2022, 11:29 AM

Objective:
The objective of this assignment is helping you understand matrix transformations, coordinate systems in OpenGL and camera projection mechanisms. In this assignment you will learn to use linear algebra to transform the coordinates from one space to another coordinate space, apply matrix operations to objects for an interactive application, and set up cameras.

Summary:
In this assignment, you are provided with a scene with several OBJ models (e.g., a floor with a cube, a teapot and a pyramid on it). Your tasks for the assignment are to complete the framework so that it computes a model transformation for a given scene of three objects, viewing transformation, and a projection transformation. First you will need to construct the scene by applying transformations to the objects (rotation/translation/scaling). Then you need to place two cameras into the scene for two views, one using perspective projection and the other using orthographic projection, to show the difference. After setting these up, the application should also allow the user to walk around (including turning left/right, stepping forward/backward).

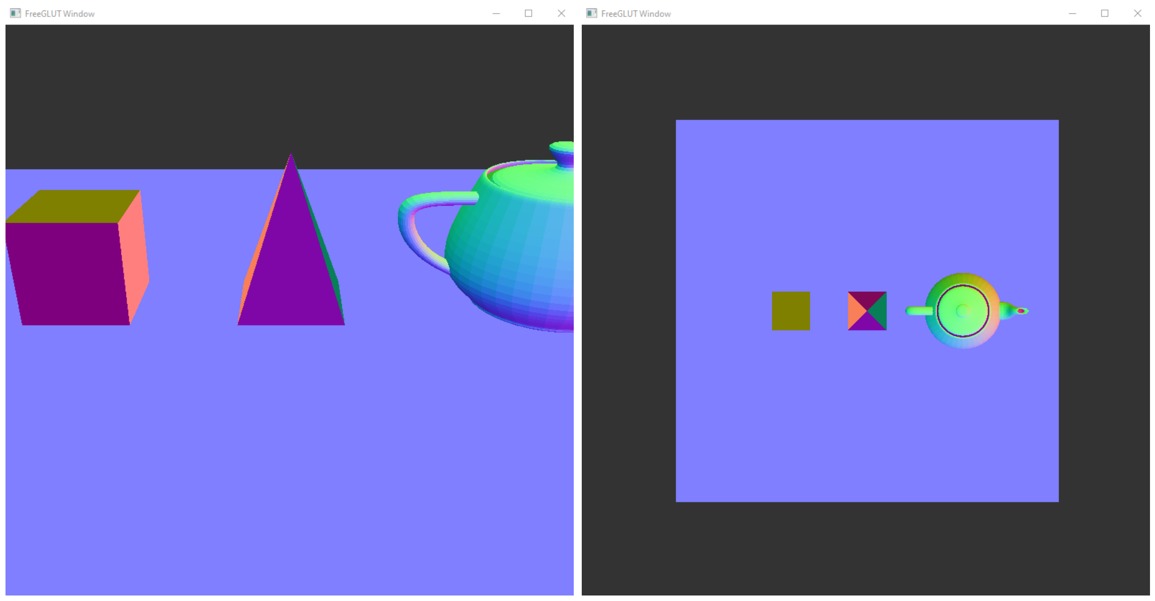
Specifics:

1. Start with the templates from the course website (choose FreeGLUT or Qt). The templates support Windows and Linux environments. Compile and run the templates. Note that this time most events (e.g., walking around the scene) are expected to be controlled by keyboard, so not many graphical user interfaces are provided. But you can always add UI components according to your needs.
2. **(15%) Model Setup.** When you successfully run the templates, you'll see four objects overlapping with each other (a cube, a teapot, a pyramid and a ground plane). The scene is a mess because the vertices of the objects are in their local coordinates, having not been transferred to world space yet. Your first task is to calculate a model matrix for each object so that after the transformation you construct the scene. To calculate the model matrices, you will need *scene.txt*, a file under *models* directory. The format is:

```
k
fn1
rMat1
tVec1
fn2
...
```

where *k* is the number of objects (i.e., 4), fn_i is the filename of the ith object, $rMat_i$ is the 3x3 rotation matrix of the object (written in three lines and three columns), and $tVec_i$ is the 3D translation vector of the object (including three elements, written in one line). You need to read and parse this file to get the models and the matrices/vectors before applying the transformation (scaling, rotation & translation) on the objects. In *scene.cpp*, you need to complete the function *Scene::parseScene*. As given, the function already reads-in the mesh file fn_i into a mesh object. You need to implement reading in the rotation matrix and translation vector and storing it in the 4x4 matrix of each mesh object, e.g., *mesh->modelMat*. You might have to transpose the matrix depending on how you read it into the memory.

3. In one constructor of camera.cpp, *view* matrix is hard-coded so that you will get a better initial view when you get the template to run. Remember to delete it when you move to the step below.
4. **(25%) Projection Setup**. You will set up two camera objects, which are stored in the *GLState* object (*camGround* and *camOverhead*). One for a front view where you are looking down -z and +y is up (using perspective projection of vertical FOV 45 degrees), and one for an overhead view where you are looking down -y (using orthographic projection). In the camera class, there is *proj* and *view* matrix for you to fill out. According to the camType, you should update *proj* inside the *updateViewProj* function (the *view* matrix is set up as described later). For the aspect ratio needed for the projection matrices, you must compute the aspect ratio from the window size stored in the camera class. In main.cpp/app.cpp, the code swaps between camera views when the user presses ‘s’. For example, when correctly implemented the user expects to see the images below (left is front view and right is overhead view):



5. **(60%) View Setup.** Next, you must add to the camera class code to implement the functionality of “walking around” of the scene. When in the front view, the user should

look slightly downwards. But, the user walks forward/backward parallel to the floor, and rotates left/right about the y axis (i.e., the up axis), centered at the camera position. When in the overhead view, the user rotates left/right about the y axis (i.e., the same up-axis), and moves forward/backward (parallel to the floor). You must implement the functions *moveForward*, *moveBackward*, *turnLeft*, and *turnRight* in the class *Camera* that perform the aforementioned view functionality by altering the *view* matrix stored in the *Camera* class. The user controls are ‘s’ to shift views, ‘a’ to turn left, ‘d’ to turn right, ‘w’ to move forward, ‘x’ to move backward. In main.cpp/app.cpp, the code calls the above functions as needed.

6. In class *GLState*, the method *GLState::paintGL* makes uses of the above matrices you defines (i.e., mesh->modelMat, camera->view, camera->proj). This function will combine the transformations and pass it to the shader.
7. *GLM* provides methods to do mathematics, however you are only allowed to use *glm::perspective*, *glm::ortho*, *glm::sin*, *glm::cos*, *glm::radians*, *glm::cross*, *glm::dot*, *glm::normalize* and the basic data types (e.g., *vec3*, *vec4*, *mat3*, *mat4*). You should NOT use *glm::lookAt*, *glm::scale*, *glm::rotate*, *glm::translate* anywhere in your code. Implement them by yourself if needed.
8. The functions/methods requiring your implementation will be marked “TODO”; however, depending on your particular implementation there might be other places for you to change code. You are expected to add/modify the code as necessary to make sure the application runs smoothly.

Turn-in:
To give in the assignment, please use Brightspace. Give in a zip file with your complete project (project files, source code, and precompiled executable). The assignment is due BEFORE class on the due date. It is your responsibility to make sure the assignment is delivered/dated before it is due. If you wish to receive confirmation of receipt, please ask by email in advance.

Don’t wait until the last moment to hand in the assignment!

For grading, the program will be compiled on Linux and run from the terminal (with Visual Studio as a fallback – please try to avoid platform-specific code (e.g., don’t #include <windows.h>)), run without command line arguments, and the code will be inspected. If the program does not compile, zero points will be given. If you have more questions, please ask on Piazza!

Good luck!