Graphics Programming HW 2

Due: 4/6 11:59 PM

In this assignment, there are four problems, but can be implemented as a single program. So you do not have to make a program for each problem. Just fill in `main.cpp, camera.cpp, texture_cube.h` and shaders of the given skeleton code. For problem 1 and 2, you can copy LearnOpenGL site's code. But we hope you to at least understand and learn something from doing the homework.

Prob 1. Interactive 3D Camera (30pt)

Your task here is to implement a movable camera so that you can walk through a scene. In this problem, you have to implement followings.

- You have to render 10 textured cubes in 3D space. The initial position of each cube is already provide. Please add rotation to each cube such that individual cubes rotate in different orientations. Do not change its size. Actually this is the same as your lab session assignment, so you can make use of it.
- Now let's go further from the lab session; make the camera movable. The camera should be moved in following ways.
  - Camera position: WASD (Forward / Left / Backward / Right) (Here, the forward direction is same as current camera direction. This means that if camera looks up the sky, the camera should go up when we press W.)
  - Camera yaw and pitch: Mouse cursor move. (Ignore camera roll)
  - Camera FOV (or zoom): Mouse scroll.

  Note that the camera position, yaw and pitch affect the view matrix, while the camera FOV affects the projection matrix. Also don't forget that pitch should be under 90 degree and over -90 degree. Please set appropriate sensitivity when you rotate the camera or change the FOV.

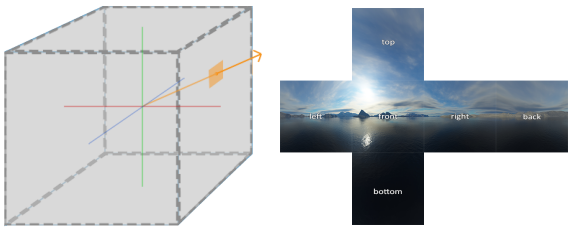
Prob 2. Cubemap Texture (30pt)

Now, let's incorporate more fancy features with texture. A cubemap is basically a set of textures that consists of 6 individual 2D textures that each form one side of a cube. You can sample a texture value of the cubemap with a direction vector. A skybox, which is widely used in many games, also uses a cubemap.

In this problem, you will implement a simple skybox using a cubemap. Overall steps are similar to rendering a textured cube, but there are several differences. One is that the translation part of the view matrix should be removed. If not, a player may go out of the skybox, which is not desired. Also, you have to use three-dimensional uv to fetch texture from a cubemap (Fig. 1 (a)). You also have to disable depth

masking when drawing a skybox, so that it will always be drawn at the background of all the other objects. For the details, it is strongly recommended to read LearnOpenGL cubemap chapter.

Fill in the `CubemapTexture` class, then use it to render a skybox. You also have to implement an additional shader `shader_skybox` that renders the skybox. In addition, you need to implement mixing two skybox textures. Proportion of each texture (`dayFactor` in code) should be changed when we press O and P on the keyboard.



(a) (b)

Figure 1: (a) A direction vector is used for cubemap sampling (b) An example of a skybox cubemap

Prob 3. Billboard (25pt)

For this problem, you will implement billboards. Billboards are 2D planes that always face toward the camera (or perpendicular to the camera's forward direction), like health bars in 3D games. Billboards are also used to approximately render a bunch of grass or faraway trees. In this section, you will render a small field covered by grass using billboards. Here, one important thing is that you do not want your grass to be bent: it MUST be vertical. So you need to rotate them only around y-axis (Fig. 2).

There are many ways to implement billboards, but in this problem, we want to evaluate whether you have fully understood 3D transformation. So **DO NOT ADD** additional shader or VAO/VBO to render billboards, but manipulate the view matrix or model matrix to make grass face toward the camera. Use a quad as a VAO. Ignore transparent texture (there is a hint in skeleton code). Also please render the ground, so that your grass do not float on the air.

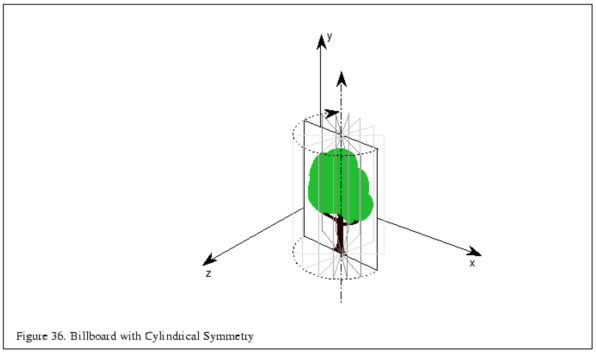


Figure 2: Billboards of tree or grass should rotate only around the vertical axis.

(Hard) Prob 4. Camera Interpolation using Quaternions (15pt)

The last problem is an optional problem. It is a challenging and time-consuming task, but you will get only few additional points. So try it only if you are confident that you have fully understood the concept of quaternions and coordinate system.

In this problem you will implement two simple camera interpolation systems.

- If we press key Q, remember the current position and orientation of the camera. Initialize these values with the camera's initial values.
- If we press key E or R, make the camera move back to the saved position/orientation smoothly. For position interpolation, use linear interpolation for both cases. For orientation interpolation, use a different strategy for each key.
- Key E: Use yaw/pitch linear interpolation. Also handle the cases that the interpolation covers the range beyond $[-\pi, \pi]$. Choose the shorter path. For this method, the roll of the camera should remain zero (or the vertical component of camera's right direction should be zero) while interpolating.

$$yaw(t) = yaw_1 \cdot (1-t) + yaw_2 \cdot t$$

$$pitch(t) = pitch_1 \cdot (1-t) + pitch_2 \cdot t$$

$$roll(t) = 0$$

- Key R: Use quaternion for spherical interpolation (`slerp`). For this method, roll may not remain zero while interpolating.

$$q(t) = \texttt{slerp}(q_1, q_2, t)$$

- To verify difference between the two methods, print camera's right direction during interpolation. You can just print on terminal, or use `TextRenderer` which was given in HW1, to print on a screen.
- Make sure to choose the **shortest path** for orientation interpolation. For example, if yaw_1 is -170° and yaw_2 is 170°, you should interpolate by 20°, not 340°. To evaluate this, please also print cameras yaw as well as cameras right direction.
- For simplicity, assume that there is no acceleration. Set interpolation time to a constant value, maybe a few seconds, regardless of interpolation path length. Also during interpolation, ignore all mouse/keyboard inputs.
- You do not have to implement quaternion from scratch. Use `glm::quat`.
- Make sure that billboards remain to work fine during both interpolations. This would be a challenging task.

Figure 3: Final scene of HW2

To get a full credit of HW2, you have to submit a program that can render a scene like Fig. 3.