Middle East Technical University | Department of Computer Engineering

CENG 469

Computer Graphics II

Spring 2021-2022
Assignment 2
Cubemaps in OpenGL
(v1.0)

Due date: May 15, 2022, Sunday, 23:59

Figure 1: A sample rendering result for the assignment.

1 Objectives

In this assignment, you are going to implement environment mapping and reflection capabilities using cubemaps in OpenGL. The expected end product of this assignment is an OpenGL program which: *(i)* provides user interaction capabilities with keyboard and mouse buttons, and *(ii)* renders a scene composition that includes the items listed below:

- A textured skybox environment,
- A reflective (mirror) object that is placed into the center of the scene and rotates around the vertical axis.

- One or more diffuse objects that rotate around the center object in such a way that their reflections are visible on the center object. A sample result is shown in Figure 1.

The specifications are explained in Section 2. Note that you are free to implement your own design choices as long as you obey the requirements detailed in this assignment text. Design choices include but are not limited to:

- The skybox texture of the scene.
- The mirror mesh object (cube, teapot, or another mesh) as long as the reflections of the orbiting objects are clearly visible. For instance, the bunny mesh is too complex to understand if the reflection capability is working correctly.
- The orbiting mesh objects, their animation trajectory, and their colors, or textures (if you want to add textures to them).
- The color of the point lights that reside in the orbiting diffuse mesh objects' shader programs.

2 Specifications

1. Cubemaps will be used for the skybox and reflection capabilities.
    - For skybox, a cubemap texture will be generated that reads the textures of its faces from the disk as static image files. We call this cubemap the "static cubemap". The static cubemap's textures will then be used when rendering the skybox mesh.
    - For reflection, another cubemap, that we call the "dynamic cubemap" will be rendered at the beginning of each frame, by:
        - Placing the camera into the center of the mirror mesh and modifying its FOV to be 90 degrees and its aspect ratio to be 1, as we will render a dynamic texture to a square-shaped cube face. Set the distances of the near plane and the far plane appropriately to include all meshes of the scene.
        - 6 directions will be looked at by the camera and rendered to the corresponding face of the dynamic cubemap, which is attached as the color attachment to a pre-initialized framebuffer object. This means that these 6 renders will be off-screen. At each of those 6 renders, the up vector of the camera should also be set appropriately to have the final reflection visuals correct. A depth buffer must be attached to the framebuffer to enable depth testing during those 6 off-screen renderings. Note that we are still in the process of generating 1 frame of our main program.
        - During those 6 renderings, we ignore the mirror mesh itself as if it does not exist in the scene. All other meshes: skybox and the orbiting object(s) are rendered.
    - Then the main camera angle will be rendered. This will then conclude the process of generating 1 frame of our program:
        - Do not forget to reset the viewport correctly when switching from the cubemap face rendering to the main camera angle rendering.

- Render the skybox using the static cubemap. The depth writing should be disabled before the skybox is drawn to the screen. It can be enabled afterwards. As the skybox cube resides very close to the camera (each face of it has a distance of 1 to the camera position), it occludes the other objects in the scene that are positioned at a larger distance from the camera than this distance. Another note on the skybox is that while it is being rendered, any translation must be removed from the viewing matrix so that as the player moves, the cubemap translation stays the same: The player cannot approach the skybox, but can look around (rotation is not removed), giving the feeling of an extensively large world and boosting the immersion of the scene.
- Render the mirror mesh using the dynamic cubemap. At its fragment shader, "glm::reflect" function must be called with appropriately calculated parameters, and the result must be used to obtain the dynamic cubemap texture value as the fragment color output, making the mesh a perfect mirror.
- Render the orbiting diffuse meshes.
- Update the values of the rotation and orbiting animations.

2. The program should have the following user controls:

   - **W and S buttons** should move the camera in its Gaze vector direction forward and backward, respectively.
   - **A and D buttons** should move the camera in its Right vector direction backward and forward, respectively. Suppose Gaze $\times$ Up = Right for the camera.
   - **Q and E buttons** should move the camera vertically downward and upward, respectively.
   - **Right Mouse button**: Moving the mouse while pressing and holding the Right Mouse Button should make the camera to look around the scene according to the mouse movement direction. Information on the implementation of this capability can be found in this link.

3. The orbiting objects can have the default shaders that exist in the sampleGL.zip codebase provided to you. The skybox should have a shader that implements environment mapping by querying the pre-initialized and bound static cubemap. The mirror mesh should use the dynamically generated cubemap of that frame.

4. You can try changing the "glm::reflect" call to "glm::refract" call in the shader of the center mesh and see the visual result of refraction in your program. You can put a screenshot or video of the refraction results into your blog post. However, your submission should show reflection when compiled and run.

5. You are advised to use the **sampleGL.zip** file shared to you on ODTUClass course page as the template code and build your homework code on top of it.

6. You can use an image reader library to read the image data from any image file type (JPG, PNG, etc.) to be used as the skybox texture. As an example, you can use the header-only **stb_image.h** library as follows:

   (a) Download and include the file:

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

(b) Load the image:

```
int width, height, nrChannels;
unsigned char *data = stbi_load(faces_img_names[i].c_str(),
                        &width, &height, &nrChannels, 0);
```

7. There is no frames-per-second (FPS) constraint in the homework but your animation should be seen as continuous and smooth. Please test your code on Inek machines before submitting, as the animations may slow down due to the performance capacity of the Inek machines.

8. **Blog Post:** You should write a blog post that shows some output visuals from your work, and explains the difficulties you experienced, and interesting design choices you made during the implementation phase. You can also write about anything you want to showcase or discuss. Below we provide some discussion ideas:

   - What kind of orbiting animation you have implemented and how.
   - An analysis of **FPS** (frames-per-second) of your code.
   - The resulting visual of the reflection when the dynamic cubemap's texture resolution is decreased harshly.
   - If you try refraction, the visuals of it with different ratio of indices of refraction, where air's index of refraction is 1.00: *(i)* Water: 1.33, *(ii)* Glass: 1.52, *(i)* Diamond: 2.42.
   - If you render a scene composition where two mirror meshes reflect each other (for example the orbiting mesh is also a mirror), or if the center mirror mesh can reflect itself in your code, a discussion on how you have implemented this in your code.

   The deadline for the blog post is 3-days later than the deadline for the homework. You can submit your code before the deadline, and finalize the blog post during the 3-day late submission period without losing late days. You consume your late days only if you submit your "code" late. However, submitting the blog post more than three days later will incur grade deductions.

3 Regulations

1. **Programming Language:** C/C++. You also must use gcc/g++ for the compiler. Any C++ version can be used as long as the code works on Inek machines.

2. **Changing the Code Template:** We provide you a code template (sampleGL.zip on ODTUClass). You are free to edit, rename or delete any file from the given code template as long as you comply with the submission rules below.

3. **Additional Libraries:** GLM, GLEW, and GLFW are typical libraries that you will need (these are already included in the sampleGL source file). You should not need to use any other library except for reading images. But if you still want to use some other library, please first ask about it in the ODTUClass forum of the homework.

4. **Submission:** Submissions will be done via ODTUClass. You should submit your blog post link to "Blog Links" forum on ODTUClass, using the title "HW2 Blog Post". For code submission, create a "**tar.gz**" file named "hw2.tar.gz" that contains all your source code files, texture images, meshes, and a Makefile. The executable should be named as "**main**" and should be able to be run using the following commands (any error in these steps may cause a grade deduction):

   tar -xf hw2.tar.gz
   make
   ./main

5. **Groups:** All assignments are to be done individually.

6. **Late Submission:** You can submit your codes up to 3 days late. Each late day will be deduced from the total 7 credits for the homeworks of the semester. However, if you fail to submit even after 3 days, you will get 0 regardless of how many late credits you may have left. If you submit late and still get zero, you cannot claim back your late days. You must e-mail the assistant if you want your submission not to be evaluated (and therefore preserve your late day credits).

7. **Cheating: We have zero tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0 from the homework. You can discuss algorithmic choices, but sharing code between students is strictly forbidden. Please be aware that there are "very advanced tools" that detect if two codes are similar.

8. **Forum:** Any updates/corrections and discussions regarding the homework will be on ODTUClass. You should check it on a daily basis. You can ask your homework related questions on the forum of the homework on ODTUClass.

9. **Grading:** Your codes will be evaluated on Inek machines. We will not use automated grading, but evaluate your outputs visually. Implementing the following will get you 100 points from the homework part: A program that correctly implements the user-interaction capabilities and renders a smoothly-animated scene composition where there is a correctly textured skybox environment, a mirror mesh in the center which reflects every object other than itself, and one or more diffuse objects orbiting around the mirror mesh, whose reflections are visible on the mirror mesh. Note that you should test your code on an Inek machine before submission, as the frame rate might not be as high as your home computer if you have a powerful PC, and might cause hangs during the animation. This might require reducing/optimizing per frame calculations in the code. Blog posts will be graded by focusing on the quality of the content.