Middle East Technical University    Department of Computer Engineering

# CENG 469

Computer Graphics II

Spring 2022-2023
Assignment 2
Cubemaps
(v1.0)

Due date: May 7, 2023, Sunday, 23:59

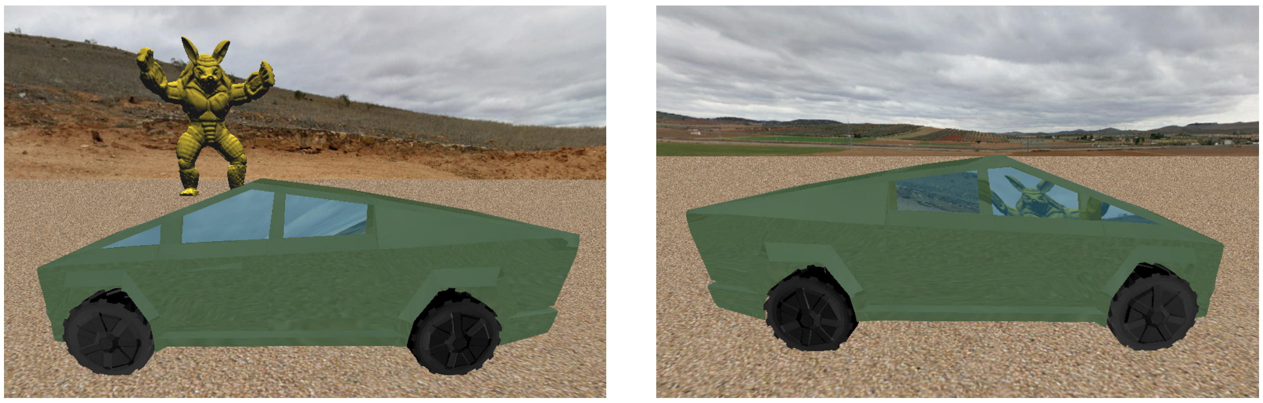

Figure 1: The reflections of the sky, ground, and statue are visible on the car.

## 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 buttons, and *(ii)* renders a scene composition that includes the items listed below:

- A textured skybox environment,
- A colored car object whose body and windows reflect its environment.

- A large-enough scene composition with a textured ground and one or more big-enough diffuse statues/landmarks whose reflections are visible on the car's reflective parts when the car travels in the scene. Sample visuals are 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 textures of the skybox and the ground of the scene. Some images are shared to you.
- The meshes of the statue/landmark object(s), their colors and positions in the scene. Some meshes are shared to you (armadillo, bunny, and teapot).
- The colors of the car's body and windows, which are to be combined with the reflection colors in their own shaders.
- Driving another object in the scene (teapot, etc.) instead of the car as long as the reflections are clearly visible (for instance, the bunny mesh is too complex to understand if the reflection capability is working correctly, but it can be used as a statue object in the scene). However, we recommend driving the given Cybertruck mesh as the car, as its parts are already separated for you to easily read and create the meshes and apply different vertex and fragment shaders (OpenGL programs) to each of them. The same transformations can be applied to all parts of the car (body, windows, and tires), as they complement each other and produce the complete car visual in the world space without requiring any additional transformation.
- The color, power, and location of the lights that reside in the shaders of each of the statue, car body, car windows, and car tire meshes.
- Optionally, rotating each tire of the car appropriately as the car moves. For this, you might want to separate each tire from the car tires object into a separate .obj file by keeping the vertex positions, and then read in each separated tire object in your code and apply appropriate transformations to it to animate it, such as translating it to the origin of the scene using the center of mass coordinate value of all of its vertices, rotating the mesh around the axis passing thru the wheel hub, and translating it back to its original coordinate position, at each frame.

## 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 to the same world position as the car mesh and modifying the camera's 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 all parts of the car mesh itself as if they do not exist in the scene. All other meshes: skybox, ground, and statue object(s) are rendered.
  - Then the main camera angle will be rendered (the main camera always follows the car's position). 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 (the provided cube mesh) 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 car moves, the cubemap translation stays the same: The car cannot approach the skybox, but can rotate around (rotation is not removed), giving the feeling of an extensively large world and boosting the immersion of the scene.
    - Render the reflection color on the meshes of the car's body and windows 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 and be added to the fragment color output, making the car's body and windows have some reflection of its environment. Fragment color also includes some lighting of your choice, such as a dark greenish ambient light, to make the car have some color together with the reflection effect.
    - Render the ground and statue meshes.
    - Update the values of the car's speed and position; the camera's gaze and position, and optionally the tires' rotation angles.

2. The program should have the following user controls:
  - **W and S buttons** should speed up and slow down the car, respectively, in the car's gaze direction. After a complete stop, S button speeds up the car in the reverse direction. No upper limit exists on the speed. The camera follows the car appropriately, as can be seen in the demo video we shared. Tapping-only or pressing-and-holding actions are both okay for the buttons during the grading.
  - **A and D buttons** should rotate the car around its vertical axis to left and right, respectively. The camera that follows the car also gets its position and gaze vectors updated appropriately, as can be seen in the demo video we shared. Tapping-only or pressing-and-holding actions are both okay for the buttons during the grading.

- **Q, E, R, and T buttons:** Q and E buttons change the camera angle so that it shows the left-side and right-side of the car, respectively. R and T buttons show the car from its back (which is the default camera angle) and from its front, respectively. The camera gets its position and gaze vectors updated appropriately, as can be seen in the demo video we shared.

3. 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.

4. The ground can be produced by reading the ground.obj file we shared to you, rotating it to have a horizontal surface, and scaling it on the surface axes to have the ground mesh, on which then a detailed-looking texture can be applied. **Hint:** Use U-V coordinates larger than 1 for the vertices of the ground mesh, and set the texture sampler to GL_REPEAT to repeat the sand texture on the scaled-up ground mesh, preventing the blurry texture that would otherwise occur if the sand texture was stretched over the large surface of the ground mesh due to the U-V values being in range [0, 1].

5. Each mesh should have its own fragment and vertex shaders. The default shaders of the sampleGL.zip codebase can be leveraged as a starting point.

   - The skybox should have a shader that implements environment mapping by querying the pre-initialized and bound static cubemap. The file paths of the face images of the static cubemap can be embedded into the code as your main executable will be run without any arguments during grading.
   - The car's body and windows meshes should use the dynamically generated cubemap of that frame in their shaders. The body and windows shaders should differ from each other visually. The body might have a scaled down reflection color and more ambient color, whereas the windows might have a more intense reflection color. The tires mesh should not have reflection in its shader, but just some tire color.
   - The shaders of the ground mesh should correctly sample the sand texture on the mesh. A different sampler than the one used for rendering the cubemap might be used, as the ground texture can be repeated (GL_REPEAT) onto the scaled up ground mesh, requiring a different sampler GL_TEXTURE_WRAP setup. **Hint:** You might want to inspect the "glActiveTexture" function if you use multiple samplers.
   - The shaders of the statue mesh(es) have no reflection, but only diffuse, ambient, and/or specular effects. The light(s) positions/colors/intensities can be set however you like, as long as the mesh is illuminated well.

6. You can try changing the "glm::reflect" call to "glm::refract" call in the shader of the car's body and windows meshes 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.

7. **Some Additional Hints:**

   - The car's new gaze can be calculated after a rotation by using sin and cos functions on the new value of a "car rotation angle" variable. Different methods can also be used instead of this. The calculation of the gaze vector of the camera can then leverage the car's new gaze vector's value.

- "glm::lookAt" function can be used with appropriate arguments to calculate the new viewing matrix whenever its needed (after the camera's position and gaze get updated, or while capturing the static cubemap's faces).
- In the shaders, the light positions, whose values can be embedded into the shader, might be multiplied with the modeling matrix to have them transformed, too, in case you need such a capability; inside the shaders of the moving car, for example.
- Information on properly setting the Up vector of the camera while rendering a cubemap: https://www.khronos.org/opengl/wiki/Cubemap_Texture.

8. 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 and also the ground 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);
   ```

9. There is no frames-per-second (FPS) constraint in the homework but the car's movement 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: Especially the keyboard button actions might behave slowly.

10. **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 a scene composition you have prepared and which meshes you have used.
    - The resulting visual of the reflection when the dynamic cubemap's texture resolution is decreased harshly (maybe combined with an **FPS** (frames-per-second) analysis).
    - 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 there are other reflective/mirror objects, and the car mesh and those mirror objects do reflect each other, a discussion on how you have implemented this in your code, and the resulting visuals.

    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++ is highly recommended. You also must use gcc/g++ for the compiler if you use those programming languages. Any C++ version can be used as long as the code works on Inek machines. If you use any other programming language, be sure that the Inek machines can compile and run your code successfully and also include a simple Readme file to describe how we should run your code on Inek machines during the grading (if you don't use C/C++).

2. **Additional Libraries:** GLM, GLEW, and GLFW are typical libraries that you will need. You should not need to use any other library. But if you still want to use some other library, please first ask about it in the ODTUClass forum of the homework.

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

4. **Submission:** Submissions will be done via ODTUClass. You should submit your blog post link to Blog Links forum on ODTUClass, so that the URLs are publicly available for everyone's access, 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, shader files, Makefile, and additional meshes and texture images. 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. **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).

6. **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.

7. **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.

8. **Grading:** Your codes will be evaluated on Inek machines. We will not use automated grading, but evaluate your outputs visually. 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 when launched. This might require reducing/optimizing per frame calculations in the code. Blog posts will be graded by focusing on the quality of the content.