Middle East Technical University | Department of Computer Engineering

CENG 469

Computer Graphics II

Spring 2023-2024
Assignment 2 (v1.0)
Environment Mapping & HDR Lighting

Due date: May 13, 2024, Monday, 23:59

Figure 1: Objects are illuminated by the environment radiance.

1 Objectives

In this assignment, you are going to implement environment mapping and HDR tone mapping operations 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 sphere object at the center of the world which can be illuminated by an environment map, or can reflect or refract its environment.
- A textured HDR skybox environment that surrounds the scene and can also illuminate the sphere object in the scene.
- Tone mapping capability to render and display the HDR scene composition in LDR monitors.

Some sample visuals are provided in Figure 1. More visuals can be found in the YouTube video we shared. The specifications are explained in detail in Section 2. You are free to implement your own design choices as long as you obey the requirements discussed thoroughly in this assignment text.

2 Specifications

1. The scene will consist of a skybox that surrounds the scene, a sphere object that is located at the world origin, and the camera that looks at the origin at all times but from a distance. Alongside its task of producing the skybox of the scene, the image of the skybox will also be used by the sphere object as an illumination source.

2. An HDR equirectangular projection image will be read by the program from the disk and the 6 cubemap face textures will be produced from it mathematically. You can see this mathematics in Environment Mapping slides on ODTUClass course page.

3. The environment's illumination will be calculated by implementing the algorithm described in the paper "A Median Cut Algorithm for Light Probe Sampling" and running it on the HDR equirectangular image. The algorithm will produce a number of light positions and their radiances on this image's coordinates. You should then revert the mathematical process of producing cubemap faces from the equirectangular image to find the directional light positions in OpenGL's 3D world so that this lighting information (3D light positions and their radiances) can be used in the OpenGL to illuminate the sphere object. More detail can be found in **Hints**.

4. While rendering the skybox, the depth writing should be disabled before the skybox is drawn to the screen. It can be enabled afterward. 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 camera moves, the cubemap translation stays the same: The camera cannot approach the skybox, but can orbit around the center object inside the skybox, giving the feeling of an extensively large world and boosting the immersion of the scene. The color that is looked-up in the skybox shader from the cubemap texture is set as the fragment's output color, but the vector that we poll this texture with should first have its Z axis negated to account for the cubemap textures being in a left-handed coordinate system.

5. All the rendering should be done in HDR and into an off-screen floating-point framebuffer texture. The skybox values should be in HDR (floating-point). The shaders of the center object and the skybox should return log() HDR color in this first draw step. The result of this draw's framebuffer color texture is then bound as the texture for the next draw step: As the second draw step, a quad object is then drawn onto the monitor's screen without any matrix projection (spans our whole OpenGL window), which, in its shader, looks-up the bound HDR floating-point texture, that has the scene visual we expect to see rendered in log domain, and runs exp() on the looked-up HDR texture value and tone-maps it before returning it as the final LDR color for that fragment position of our monitor screen. Both of those draw steps, when chained together, will render, tone-map and show the HDR scene in our LDR monitors. We will use a basic tone mapping operator whose starting code is available in "HDR lighting tone mapping" course slides. The modification we'll make to this code is described in **Hints**.

6. **Visualization Modes:**

   The program should have 5 visualization modes to render the scene:

- **LightProbe mode:** The center object has a grayscale kd and ks value. The radiances of the directional light sources calculated by the algorithm are used for diffuse and specular illumination and those diffuse and specular colors are accumulated. Ambient color is not used as the idea of using the algorithm is to calculate the environment light. kd and ks values of the sphere should be set appropriately to have a sphere that is well-lit and blends with the scene's background (skybox) after being tone mapped and then shown on the monitor.
- **Mirror mode:** The center object is rendered as a perfect mirror object. None of the algorithm lights are used in this mode. Diffuse and specular lighting is also not used. A simple **reflect** call in the shader is used to query the skybox texture for that position to have a mirror effect. Reflected vector's z coordinate should be multiplied by -1 before querying the skybox texture to make the conversion from the right-handed OpenGL to left-handed cubemap texture.
- **Glass mode:** The same steps are used as was in the Mirror mode, but this time issuing a **refract** call. This mode's kt (transmissive k) value can be decreased to have a darkened glass visual. We will not implement a second refraction for the first refraction's exit from the object's inside. We will query the skybox using the first-and-only refract call result. This will still give us a glass visual, especially better visual in complex meshes.
- **Glossy mode:** Some mirrorness is appended on top of the color calculated in Light-Probe mode. Mirrorness can be set by changing the km (mirrorness k) in light calculations of the shader.
- **SpecularDisco mode:** A fun debug mode **which we will also use during the grading** to see if the algorithm-calculated lights have correct color and if the are set to their appropriate 3D locations by inspecting their specular visuals on the sphere. That means in this mode, a specular lighting calculation will be done with an exceptionally high Phong exponent to have sharp specular effects for each algorithm-light. The only color source will be this distinctly-visible specular component calculated using the algorithm-lights (so no diffuse coloring). The video we have shared can be inspected.

7. **Keyboard and Mouse Buttons:** The user should be able to orbit the camera around the center object as follows:

   - Holding the **Right Mouse Button** and moving the mouse. A simple yaw-pitch rotation can be implemented to find a gaze vector as if the camera is looking around while not moving from its position. This gaze vector can then be used as the "camera position" as if the camera is at that gaze location, and then the camera can be made looking at the center's (0,0,0) coordinate. Any other implementation idea can also be used.

   The user should be able to change the exposure as follows:

   - Initially, it starts at the value of 0.18.
   - Tapping **W** button should multiply the exposure by 2.
   - Tapping **S** button should divide the exposure by 2.

   The user should be able to rotate the skybox and algorithm's light positions in the 3D world during runtime as follows:

- Tapping **A and D** button rotate the skybox around the center object towards left and right. Center object is located at (0,0,0) and itself does not move at all in this mode. The algorithm-lights are also rotated appropriately together with the skybox. See **Hints** for more information on this.

The user should be able to turn off the specular lighting effects globally as follows:

- Initially, specular effects are visible in all relevant visualization modes.
- Tapping **F** button toggles the visibility of the specular light effects. The video we have shared can be inspected to see this button in action.

The user should be able to increase/decrease the "n" parameter of the algorithm to have 2^n directional lights during runtime as follows:

- Tapping **E** button should decrease the value by 1, down to the minimum value of 1.
- Tapping **R** button should increase the value by 1, up to the maximum value of 7.

The user should be able to change the visualization mode of the center mesh during runtime as follows:

- Tapping **1** button should change the visualization mode to *LightProbe* mode.
- Tapping **2** button should change the visualization mode to *Mirror* mode.
- Tapping **3** button should change the visualization mode to *Glass* mode.
- Tapping **4** button should change the visualization mode to *Glossy* mode.
- Tapping **5** button should change the visualization mode to *SpecularDisco* mode.
- All those buttons are the numeric ones right above the W key in the keyboard.
- Initially, the mode is set to *lightProbe* mode. The video we shared shows all modes in action by also showing when the buttons are pressed using the keyboard overlay visual.

8. **Hints:**

   (a) **Equirectangular-to-Cubemap Generation:**

   - The mathematics in the course slides can be used as was discussed above. The vector with xyz components should be normalized before use.
   - The vector's rotations can be done using glm::rotate() calls.
   - The face images this process produces should be directly set to OpenGL's Cubemap textures without vertical flipping as both the OpenGL's cubemap's and our calculation's image starting points are top-left of the image.

   (b) **Median Cut Algorithm:**

   - The path of the equirectangular image can be hard-coded into your program, and the image can be bundled together into your submission.
   - The check for which dimension of a region is longer than the other should use the cos of the inclination angle of center pixel of that region towards the middle equator line of the image. Each pixel's value should also be first scaled by their cos of inclination before any calculation.

- Instead of the center, the **centroid** position of a region **must be** used as the light position in image coordinates. Light's color should be set to the pixels' cos-weighted RGB sum without any averaging.
- For the process of finding the cut line, you can implement an approach similar to binary search to speed up the calculation vastly.
- The calculation of finding the 3D light position should do the inverse of the "equirectangular to cubemap" mathematics using the regions's centroid pixel location. Height index should be reversed as is in the course slide's figure. After phi and theta are found, a vector of (1,0,0) can be rotated using glm::rotate() calls around +Z by theta amount, and then around +Y by -phi amount. This gives the light position in 3D in the right-handed OpenGL system.
- When the user rotates the cubemap for some angle via the buttons along Y axis, the lights' positions should be rotated by the negative angle along Y axis.

(c) **HDR Rendering and Tone Mapping:**

- As the equirectangular image is in **.hdr** format, the internal format of the Cubemap texture should be set as **GL_RGB32F**. The framebuffer onto which we will render the scene in HDR will also have the same format.
- The center mesh's and the sky's shaders add some small epsilon value to the log() calls before returning log(color).
- The shader of quad object should calculate the tone mapping as was discussed. It first looks-up the HDR scene color from the rendered and bound framebuffer texture. Then, the tone mapping is done per fragment as follows:
  - exp() of the HDR RGB texture color is first calculated after getting it from the texture.
  - The scene's mean HDR RGB value is found by calling **textureLod()** function with a high number as the argument so that a 1x1 mipmap (mean HDR RGB) of the image is looked-up by this call automatically in shader (of course, we take exp() of this looked-up value, too). We should generate mipmaps of the rendered scene's texture by calling glGenerateMipmap(GL_TEXTURE_2D) and glTextureBarrier() before rendering the quad.
  - A scaled luminance is calculated by dividing the luminance of the exp()-applied HDR RGB texture value by the mean luminance and then multiplying with the exposure uniform. Luminances are found by interpolating the RGB's by (0.2126, 0.7152, 0.0722).
  - The rest of the tone map code lines are the same as the code in the slides.

(d) **Additional References:**

- You can inspect the following links as additional reference:
  https://learnopengl.com/Advanced-OpenGL/Framebuffers
  https://learnopengl.com/PBR/IBL/Diffuse-irradiance

9. You are advised to use the provided support files. You can 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. The skybox can be made using "cube.obj" mesh. The drawing to the screen can use the "quad.obj" mesh. You can try changing the center object from sphere to some other mesh

and the environment image to one of your own images to see some different visual results in your program. You can put a screenshot or video of those to your blog post. However, your submission should use the provided **"sphere.obj"** mesh and one of the following provided HDR maps when compiled and run: **"Snowstreet.hdr"** or **"Thumersbach.hdr"**, whose visuals can be seen in Figure 1, respectively. You can find some HDR maps on this website: https://www.openfootage.net/hdri-panorama/

10. You are advised to use the header-only **stb_image.h** library to read the HDR image. (Hint: see *stbi_loadf* function).

11. There is no frames-per-second (FPS) constraint in the homework but the overall flow should feel continuous and smooth. During the new calculation of the algorithm if the user changes the algorithm's **"n"** parameter, some short-duration freezes might occur, which does not affect your grade.

12. **Please test your code on Inek machines** before submitting, as the animations may slow down due to the performance capacity of the Inek machines. We will use Ineks for grading.

13. **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:

    - Different environment maps, or the visual of environment mapping on different center meshes,
    - Visuals for different ratio of indices of refraction, where air's index of refraction is 1.00: *(i)* Water: 1.33, *(ii)* Glass: 1.52, *(iii)* Diamond: 2.42,
    - How the scene visual gets affected if the cubemap's texture resolution is decreased or increased (might be combined with an **FPS** (frames-per-second) analysis).

    The blog website you write your blog on should have the ability to automatically show the last edit date for your post which shouldn't be able to be edited by the writer.

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

    If you would like to put some piece of code or a link to the online repository of your homework implementation into your blog post, please do so by editing your post 3 days after the homework's code deadline, so that your code is not visible to the students who use 3 late days for the coding part. The rest of the post should already be published before the blog post deadline, which is 3 days after the original homework deadline. We will check the blog pages the moment the blog post deadline ends.

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 **".zip"** file named **"hw2.zip"** that contains all your source code files, shader files, mesh files, any other content your program needs, 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):

   unzip hw2.zip -d hw2
   cd hw2
   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.