



Middle East Technical University



Department of Computer Engineering

CENG 477

Fall 2025

Introduction to Computer Graphics

Assignment 3 - Planetary Body Rendering with OpenGL

Date: The 2nd of January, 2026 23:55

1 Objectives

The purpose of this assignment is to give you hands-on practice to write a hardware-accelerated computer graphics program. This should complement the theory that you have learned in class and help you understand and solve various challenges of graphics programming.

In this assignment, you are going to implement a planetary body visualizer. It will render multiple planets. There will be a center planet (which will be the Earth) and its moon. And that moon also has its own moon. **At least these three bodies are required to be implemented in this way.** You can add more planetary bodies if you wish.

The center planet must be earth-like due to some shader effects applicable to this planet (Please see Section 2.2 for further information).

2 Specifications

2.1 General (40 pts)

- There will be a single directional light that simulates the sun.

Directional lights are light sources that are infinitely far away. So they do not have a position per se, only direction.

- The directional light will be rotated over time. This will give a simulated effect as if our scene is rotating around the sun.
- As mentioned above, there will be a center planet and its moon, and that moon's moon. So that you can tinker with relative transformations. This is the bare minimum.
- Orbit radius and size of these planetary bodies are up to you. You can set up your scene as you wish.

- There will be two types of camera, although one is a restricted form of the other. There should be a free-form movement camera (a camera like in the FPS games). While the user presses the **left mouse button** and moves the mouse through the application window, it should pan the camera. There are no specific restrictions on how it should pan, but it should be intuitive to use.
- On both camera modes **mouse wheel** should zoom in/out of the view. You can implement this by adjusting the camera position relative to the camera's gaze position.
- Additionally, in FPS camera mode, **W**, **A**, **S**, **D** should move the camera forward, left, backward, right, respectively.
- Other camera mode is the orbit mode, in which the camera's gaze point is fixed to a planetary body. If that planetary body orbits around another body, it should follow that.
- In orbit mode, **W**, **A**, **S**, **D** keys should be disabled.
- User can switch between these camera modes as follows:
 - given a list of planets P_0, P_1, P_2 , there will be a list of modes as P_0, P_1, P_2, FPS .
 - P_x means orbit camera over planet x , FPS means the FPS camera.
 - When the user presses the key **P**, it should select the next camera mode in the list. For key press **O**, it should select the previous mode.
 - Modes should rollover to the start as if it is a circular array.
- There will be another two button pairs that accelerate/decelerate time. **L** will accelerate and **K** will decelerate time. Successive presses to **L** should make the time flow backwards (So that eclipse events can be checked easily). Otherwise, it can be inconvenient to wait for planets to align for an eclipse.
- Bare minimum, these planetary bodies should be lit with diffuse and specular components and have a single albedo (diffuse reflectance/color) texture. They should have ambient (constant) illumination as well.
- There should be a background stars texture. You can render this texture over a large sphere that is centered on the camera's location.
- Additionally, the sun (as a sphere) should be rendered far away. You can implement this by translating a small sphere far away, translated relative to the camera's position towards the inverse of the direction of the directional light.

Unlike the planetary bodies, this background texture and the sun should be rendered with an orthographic projection to give an illusion that they are infinitely far away. If you try to render this with perspective projection, you may encounter problems when you zoom out of view via the mouse wheel.

2.2 Earth Shading Effects (30 pts)

The center planet will have specular and diffuse components, but also additional components, just as a basic atmosphere and a night city lights effect.

- Unlike other planetary bodies, Earth's specularity will not be constant (assuming you implemented the bare minimum for other planetary bodies). Water portions will have high specularity, and ground portions will have low specularity. To accomplish this, you will use a specular map which will have values between 0 and 1. You should interpolate your custom high/low specular power values using this texture.
- There will be a night map for Earth. This is a texture map that has lights over the large cities on Earth. According to the diffuse term (or both diffuse and specular terms), the color values in this map should be visible. If the processed pixel is in the dark side of the Earth, this map should be visible.
- This night map should be properly interpolated so that the light should slowly be visible as the Earth rotates and slowly diminish as that pixel comes towards the sun. Implementation is up to you, but it is required to be smooth.
- Finally, Earth also has a cloud map, which should be rendered separately. This cloud map will rotate at a different speed than the Earth itself. You can implement this phenomenon by rendering a slightly larger sphere after you draw the Earth. Then, enable alpha blending and render this sphere with this cloud map.

2.3 Shadow Mapping or Shadow Volumes (30 pts)

In order to capture eclipse events (when a planetary body is in front of another planetary body, its shadow will be cast on the planetary body that is behind), you need to implement one of the two shadow casting techniques, namely shadow mapping or shadow volumes.

The technique that is used in the reference implementation was shadow mapping. In this section, we'll try to explain the shadow mapping technique.

This technique relies on the fact that, if a light does not “see” a world location, it should be in shadow. Thus, you will render the scene from the viewpoint of the light and render visible positions to a texture.

Since the sun is infinitely far away, it is hard to imagine a proper camera (with gaze and position). The view transform for that camera will use the sun direction as the forward vector. The up vector should be the vector that is perpendicular to the orbit plane of the planetary bodies. The final right vector can be calculated via the cross product. Such directional lights are rendered with an orthographic projection, and this orthographic volume encapsulates the entire scene.

The step-by-step explanation of the method is as follows:

- Create a view matrix with the forward direction as the direction of the sun.
- Up direction is the Y axis (assuming your planets orbit with a rotation relative to the Y axis)

- Orthographic projection parameters (left, right, top, bottom, near, far) should tightly cover the scene.
- Create a framebuffer with a depth texture and a single-channel 32-bit floating point color texture.
- Clear the color values with a large value.
- Render your objects with their own model transforms and these view and projection matrices to the framebuffer that you've created. The shader only writes the transformed z value to the color texture.
- Save these two view and projection matrices (or a combination of these two as a single matrix). Also, save the color texture (this is called the shadow map). These will be required while rendering planets.
- This concludes the so-called shadow pass. We have rendered the planets from the point of the light and saved the closest points visible from the light.
- During normal rendering, pass the rendered color texture and saved transformation to the **fragment shader(s)** of the planets.
- While rendering your planets, pass the world space positions to the pixel shader (which you should already be doing for calculating the lighting).
- Transform the fragment's world space position with the **light's** view/projection matrix. This will give you the normalized device coordinates (NDC) of the light.
- Convert x,y values of this point (which are in NDC and should be between $[-1, 1]$) to $[0, 1]$ to access the passed shadow map (texture coordinates are between $[0, 1]$).
- Access the shadow map using these UV coordinates. Compare the queried value with the transformed z value. If our z value is less than the queried value, we are in front, so do the light calculations; if not, only use the ambient term.

Try to create a high-resolution framebuffer for shadow mapping calculations. In the reference implementation 2048, 2048 shadow map is used. A higher resolution shadow map will reduce blocky artefacts and make the shadows more realistic.

You may (or will) encounter problems when these two z values are very close to each other. This means we are shading a point either exactly the same or close to the projected pixel. An example of this is given in Figure 1.

The quite primitive but simple fix is to just bias the calculations with an epsilon (similar to the self-intersection prevention on a ray tracer). In short, $z + \text{epsilon} < \text{shadow}_z$; where epsilon is a small number. With this approach, these numerical precision artefacts should disappear. A fixed version of the previous image is given in Figure 2.

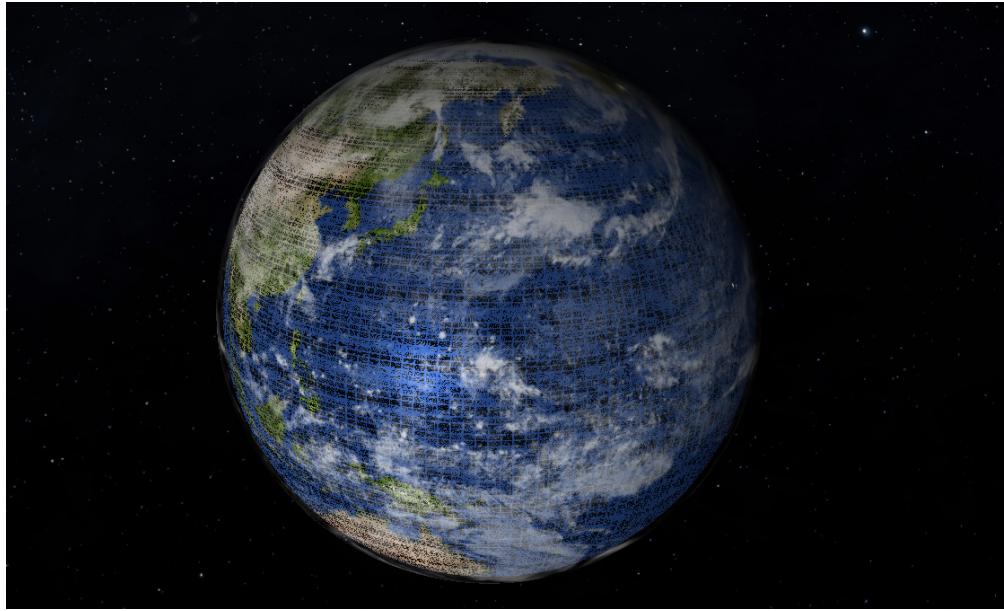


Figure 1: Shadow Acne.



Figure 2: Shadow Acne Fixed.

3 Hints and Tips

1. Shadow mapping can be complicated; you may either want to render the shadow map to the screen (using a large quad that covers the entire screen and render the shadow map over it). Or you can use external tools such as [Renderdoc](#). I will give a demonstration on how to use these tools at the recitation.
2. Do not try to get the shadow mapping perfect; it is an uphill battle. Some artefacts may remain, and you may still get full points (this excludes the problem in Figure 1).

3. Most of the textures are given to you (the night map, specular map, cloud map, etc.) in the starter pack. You can change this if and only if the shader effects are still applicable. Additional planetary textures (from our solar system) can be found [here](#). You can change the texture of other planets if you wish.
4. Try to start early. Although homework looks complicated, all of the statements can be implemented in a straightforward way.
5. Starter implementation contains texture loading, .obj file loading, and GLFW setup. It draws a single triangle on the screen with texture mapping. You can press the buttons **P** and **O** to render the triangle's different components.
6. Starter implementation uses CMake as a build tool. `cmake -B build .` (assuming you call this command inside the main folder) command will create makefiles for you inside `build` folder. You can then call `make` inside this build folder and generate the binaries over `working_dir` folder. All shaders, textures, and obj files should be in this folder. Thus, while evaluating your code "`cmake -B build . && make -C build && ./working_dir/PlanetRenderer`" command can be used.
7. Please check the explanation video in the starter kit. Additionally, an explanation of the HW will be given at the recitation.

4 Regulations

- **Programming Language:** C/C++ for the main program and GLSL for shading computations. You can use any C++ version as long as you set up your CMake properly. CMake build is used so that you can even easily port your code on Windows machines. **Do not forget that in the end, your submissions will be evaluated on an inek machine.** You can use GLSL version 430, which is the version in the template code given to you.
- **External libraries:** You can use C++'s standard template libraries, such as `std::vector`. Additionally, you can use GLFW for window management, GLEW for OpenGL extensions, and GLM for matrix operations. The template code given to you illustrates the usage of these libraries. These libraries are already available in source form, and the starter code automatically compiles them.
- **Submission:** Submission will be done via ODTUClass. To submit, create a "`tar.gz`" file named "`PlanetRenderer.tar.gz`" that contains all your source code, data CMake files. The executable should be named as `PlanetRenderer`. It should be extracted and compiled as follows:

```
tar -xf PlanetRenderer.tar.gz
cmake -B build . && make -C build && ./working_dir/PlanetRenderer
```

- **Late Submission:** Standard late submission rules defined in the syllabus apply. Do not expect postponement.
- **Cheating:** We have a 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 groups or using third-party code is strictly forbidden.

- **Forum:** Check the ODTUClass forum regularly for updates/discussions.
- **Evaluation:** Your submissions will be evaluated based on the expected render features and on the overall quality of your renderer. There will not be any black box testing; each submission will be compiled and visually checked manually.