Middle East Technical University | Department of Computer Engineering

CENG 469

Computer Graphics II

Spring 2022-2023
Assignment 3
Over the Clouds
Perlin Noise & Ray Marching
(v1.1)
v1.1: Deadline extended, new button to turn off clouds, added/edited text for clarification.

Due date: May 31, 2023, Wednesday, 23:59

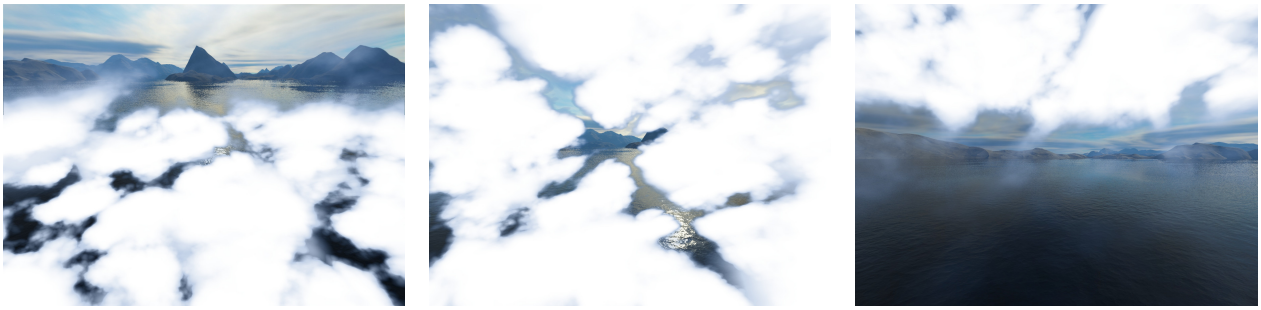

Figure 1: The plane-like camera flies over, in, and under the clouds, respectively.

1 Objectives

In this assignment, you are going to implement Perlin noise and Ray Marching algorithms in a scenario where a plane-like camera flies over/in/under the procedurally generated clouds in a textured skybox environment. The expected end product of this assignment is an OpenGL program which renders a scene composition that includes the items listed below:

- A textured skybox environment,
- A plane-like camera that moves like a plane when controlled via keyboard buttons.

- Procedurally generated clouds that are leveled at a height range over the ground, meaning that they have a start height and end height and they are clustered in between those heights. The camera can fly over, in, and under them. Sample visuals are shown in Figure 1, and also in the YouTube video we have shared.

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 texture of the skybox of the scene. Some images are shared to you.
- The camera is the "plane" in this homework. There is no plane mesh. However, you are welcome to add a real plane mesh to the scene and make the camera follow it.
- Enhancing the Perlin noise with the Turbulence idea to have more detailed clouds.
- Instead of having the emissive cloud model, enhancing the cloud lighting with ideas like scattering, etc.

2 Specifications

1. Cubemap will be used for the environmental 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. The cube mesh we shared can be used for the skybox implementation.
2. The camera flies in the scene. A quad mesh is put right in front of the camera, as if the camera wears glasses, to render the clouds. The view angles should be arranged carefully to make the quad fill the camera's view all the time and obey the camera's transformations appropriately. The given quad mesh's .obj file also includes texture coordinates. Therefore, be sure that your code parses it correctly.
3. Draw commands can be called on the skybox and the quad mesh in the respective order. The quad mesh will produce clouds in its fragments. As the camera is like "wearing glasses" – with the quad mesh being right in front of it – the final effect will be as if the camera is flying in a cloudy skybox environment.
4. The **blending** is needed to be enabled in OpenGL. The fragment shader of the quad can return fragment colors with different alpha values so that they are blended onto the already rendered skybox on the screen.
5. All transformations should be implemented using Quaternions. Please don't forget to normalize the quaternions (or their n vectors) in appropriate places before using them in calculations.
6. 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.

7. The vertex shader of the quad mesh should pass the interpolated texture coordinates and vertex (now fragment) positions to the fragment shader. This fragment shader should implement the Perlin noise and Ray Marching. At each fragment, a ray, centered from the camera position, is started to be followed step by step until a maximum distance. The step size and the march maximum distance is up to your design. At each step, the evaluated Perlin noise value at that 3D location is blended to the current color of the fragment of the current fragment shader. The fragments in this shader start at a default sky color, for which (0.2, 0.4, 0.69) values can be used. A fragment can be **discard**ed in the fragment shader if its color is still the initial sky color after the Perlin value blends. Discarding this fragment results in the previously written environment skybox color (if you drew the skybox first) to be kept as the final color of the fragment, simulating a transparent quad mesh except the clouds. Note that the color accumulated in this fragment shader is to be blended to the screen's buffer at the very end when this fragment's shader returns, which is a separate blending than calculating Perlin values at each march step and blending them to the current shader's fragment which was previously initialized to a default sky color. Some other thresholding can also be applied, such as smoothing out the ending of the clouds instead of suddenly cutting their visual, in cooperation with the discarding idea to have a nice blend between the skybox texture and clouds.

8. The ray march distance and march step size is up to your design. We expect a similar visual to the teaser images and the video, with visually appealing clouds.

9. The clouds should only be visible at a height range above the ground. The camera should be able to go above/inside/under them. A simple thresholding can be applied to the quad's fragments to remove clouds if the view the plane sees is not in the cloud-height range, whose size and the start height is up to your design. However, your program should start right under the clouds initially. You can calculate the position of the maximum reach of the current ray to cast or remove clouds for that fragment, by first checking if the position's vertical height exceeds your cloud start or end heights.

10. The program should have the following user controls (their effects can be seen in the demo video we shared, and tapping-only or pressing-and-holding actions are both okay for the buttons during the grading):

    - **W and S buttons** should speed up and slow down the plane, respectively, in the plane's gaze direction. After a complete stop, S button speeds up the plane in the reverse direction. No upper limit exists on the speed.
    - **A and D buttons** should cause the plane make a Roll rotation around the plane's current gaze axis to left and right, respectively.
    - **Q and E buttons** should cause the plane make a Yaw rotation around the plane's current vertical axis to left and right, respectively.
    - **U and J buttons** should cause the plane make a Pitch rotation around the plane's current horizontal axis to up and down, respectively. This axis can be thought to be ranging between the plane's wings, and it is orthogonal to both the plane's current gaze and the plane's current vertical axes.
    - **T button** should toggle the rendering of the clouds on the screen. When the cloud rendering is turned off, the camera should still fly like a plane inside the environment.

11. 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.

12. You may want to have separate fragment and vertex shaders for each mesh (skybox and the quad). The default shaders of the sampleGL.zip codebase can be leveraged as a starting point.

13. The size of the window might affect the performance of your program, as the fragment count on the screen increases. A (Width x Height) size of 800x600 is enough for grading.

14. 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);
    ```

15. There is no frames-per-second (FPS) constraint in the homework but the plane's movement inside the clouds at a windows size of 800x600 should be seen as continuous and smooth. A frame-per-second (FPS) value which is at least 30 would be enough. 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.

16. **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:

    - An FPS comparison of your code in different scenarios (implementing Turbulence or scattering-like lighting, versus the baseline implementation).
    - Comparison on how different modifications on the cloud generation procedure (turbulence, etc.) affect the visual appeal of the resulting cloud scenery.
    - How the change of the window's height and width affects the performance of your program.

    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 "HW3 Blog Post". For code submission, create a "**tar.gz**" file named "hw3.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 hw3.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.