PROJECT REPORT

CMP502 - Programming For Games

Author
Pablo González Álvarez
6 December 2023

Contents

1 Summary

The scene consists of a coastal area featuring a lighthouse, which serves as the primary light source illuminating the scene. There is also a dock on the beach, along with a boat floating in the water.

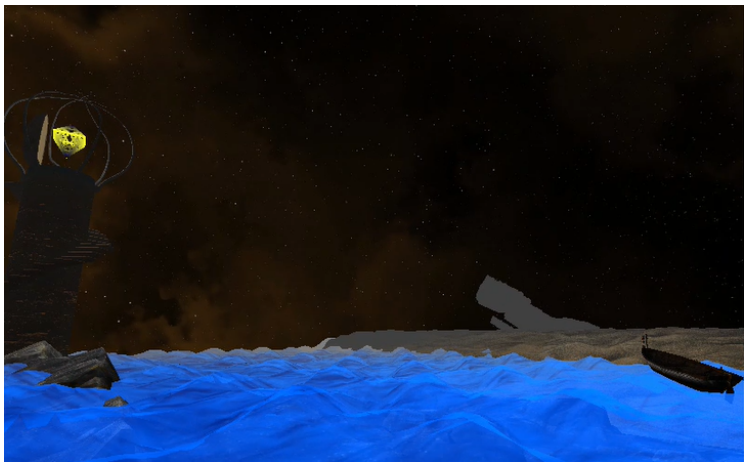

Figure 1: Game scene

2 Controls

Although, in the case of creating a game with the scene, the movement made would not make much sense, it has been decided to implement a free camera movement to view the scenario from all possible angles. Here is how it works:

- **W:** Move forward following the camera's direction.
- **S:** Move backward relative to the camera's direction.
- **D:** Move to the right relative to the camera's direction.
- **A:** Move to the left relative to the camera's direction.
- **Space Bar:** Move the player upward.
- **Shift:** Move the player downward.
- **Mouse left button:** Set movement speed to fast.
- **Mouse right button:** Set movement speed to normal.
- **H:** Unclog the player's ears with a delay of one second.

Additionally, the camera's direction is influenced by the mouse movement, taking into account its relative motion on the screen.

3 Technical aspects

3.1 Project Structure

In the project, the entities are divided based on their functions. On one side, there are the light and camera, each with its own. On the other side, there are shaders and scene entities that function through inheritance.

3.1.1 Entities

All objects in the scene inherit from the Entity class. This class encapsulates the basic behaviour of each object, such as handling translations, scaling, and rotation if needed. It also takes care of loading common elements shared by all objects, such as the mesh or shaders. Each object that inherits from this class has its behaviour in the update, if necessary, or loads new elements as required in the shaders.

3.1.2 Shaders

The shaders operate like the objects; they all inherit from a common shader class. Each shader that inherits from this class has specific behaviour based on what is intended for the entity to which the shader is applied. This could involve adding specific parameters to the vertex or pixel shader or modifying the creation of the shader itself.

3.2 Own geometry

With the learnings of the first few weeks of creating simple 2D figures and a bit of imagination, has been designed a 3D figure composed of points, where each face forms a Sierpinski triangle, to achieve this; this is the algorithm used:

$$P_s = random(P_a, P_b, P_c) \tag{1}$$

$$P_i = P_s + P_{i-1} \tag{2}$$

Where:

- P_s is the starting point of the new position to draw a.
- P_a, P_b, P_c represents the corners of the triangle.
- i is the number of pixel you are calculating and it start with a value of three because the 3 first points are the corners

Then this crystal is rotated with respect to its axis and in the update the cosine function is applied to the height giving a sensation of levitation. In addition, the middle points levitate in a different way than the highest and lowest points, giving a sense of deformation.

3.3 Depth buffer

To save the depth buffer, the framework has been modified to store this buffer in a texture, as it was not done up to this point. Additionally, a flag has been activated, allowing it to be read in the shaders for subsequent use.

3.4 Fog - Depth effect

A fog effect has been created in the objects so that when they reach a certain point, their colour begins to turn grey until it fades completely. The parameters are modifiable from the code, allowing the colour of the shadow to change when the player is underwater and gets closer to the player, giving a sensation of depth and immersion in the water.

3.4.1 Player position

To determine if the player is above the water, player's position needs to be check respect to the wave. To do this, a new point is created with the height of the water and the X and Z position of the player then a calculation is done checking the position he would have on the wave using the algorithm explained in the next point (6). With this position, is easy to know if the player is above or below the water just checking the Y position. This system does not work at all well since when it is at the limit, it can generate an unwanted effect; this is because it is still being calculated in a position in which it does not vertex and it can cause a point to be out of the water when it really is. It is inside, but due to the lack of vertices, it is seen outside.

3.5 Waves

3.5.1 Blending

The blend state defines how the output of the fragment shader is written and blended into the render target; here, it has been created to allow the use of translucent textures (Microsoft, 2020b) and to make the bottom of the sea visible from above.

3.5.2 Raster State

The rasterizer state is used to control the rendering of points, lines and triangles. Here it has been used to allow the water texture to be seen on both sides (Microsoft, 2022) and not look like the water has disappeared when you are in it.

3.5.3 Algorithm

For the movement of the waves, the vertex shader has been modified, requiring a new struct with the necessary parameters for the waves to work, the time to generate the movement and the amplitude and frequency to give the waves their shape.

$$K = Vector2(1, w) \tag{3}$$

$$XZ_w = X0 - K * A * sin(dot(K, X0) - (F * T * S)) \tag{4}$$

$$H_w = A * cos(dot(K, X0) - (F * T * S)) \tag{5}$$

$$\begin{aligned} X+ &= \frac{XZ_w.\text{first}}{u} - 0.5 \\ Z+ &= \frac{XZ_w.\text{second}}{v} - 0.5 \\ Y+ &= H_i \end{aligned} \tag{6}$$

Where:

- w is the number of the wave.
- XZ_w is the X, Y position after the wave w.
- $X0$ is the starting position of the pixel.
- A represents the amplitude of the wave, in this case 0,01.
- F represents the frequency of the waves, in this project 0,45.
- T Represent the time since the creation of the shader.
- S Represent the speed of the wave, in this project 10.
- H_w is the height position after the wave w.
- u is the size of each wave in X axis.
- v is the size of each wave in Z axis.
- X, Y, Z represents the final position of the pixel their starting value is 0.

The base algorithm (Imperat, 2016), and the one being used, is similar but with a different type of pixel shader and the vertex shader with more customizable parameters and few changes to make it more suitable for the scene.

3.6 Height Mapping

The entities in the scene can use height maps to have personalised vertex geometry without having to use some type of 3D modelling programme based on flat ground with many polygons. To make this effect work, the texture of height is added to the project, and the vertex shader changes the height of the pixel according to the values of the height map, which goes from black to white.

3.7 Lighting

The lights provided by the framework itself have not been modified, but certain improvements have been added to enhance the user's visibility of scene elements.

3.7.1 Shadows

To implement the creation of shadows in the scene, the approach taken is to create an additional view representing the direction in which the camera is looking. With this view, the scene is rendered from the perspective of the light source, and the depth buffer is saved. Later, this depth buffer will be used in the pixel shader of the objects to determine whether a pixel is in shadow or not.

Once the depth buffer is calculated, the pixel shader first determines if it is part of the camera's view. If it is inside, it calculates whether the depth is less than the value in the pixel's depth buffer. This indicates that there is an object in front if it and some entity obstructing it Rastertek, 2019.

This method of generating shadows caused some issues because part of the lighthouse was visible, creating an undesired shadow. Additionally, the sand, which has a height map, produced strange shadows due to the lack of pixels in the mesh. To address this, in the rendering from the light, the drawing of the lighthouse is skipped, and the sand ground is rendered without taking the height map into account, just for the light view, solving both problems.

There is also some aliasing issue in certain parts of the shadows, as higher resolution would be needed for nearby objects (Microsoft, 2020a).

3.7.2 Normal Mapping

Normal mapping is a technique used to enhance the lighting of a scene, allowing objects to have better illumination and a sense of relief.

To use this type of texture, upon loading the model, the tangent and bitangent of the triangles forming each of the mesh triangles are calculated. These values are then sent to the pixel shader, which computes the TBN matrix formed by the tangent, bitangent, and normal obtained when loading the mesh. Once this matrix is obtained, the final normal mapping value is calculated by multiplying the matrix by the value that pixel would have in the normal texture. The light intensity at that point would then become a blend of this previously calculated value and the intensity of the light (opengl-tutorial, 2018).

3.8 Sounds

To add sounds to the game, the first thing that has been done is to update DirectX to the 2019 version, as the previous version being used was outdated and not functioning properly.

To create the sound system, an audio engine is developed, sounds are loaded, and they are added to a sound emitter, which is responsible for executing the sounds (Walbourn,

2022). Additionally, functionality has been implemented to ensure that the application can react appropriately in case the audio output is changed.

3.8.1 2D Sounds

There is a 2D wave sound in the background, simulating a directional 3D sound so that as the player goes further into the beach, the sound of the waves gradually fades away.

In addition, the game checks whether the player enters or exits the water by triggering a water entry or exit sound. When the player enters the water, the left ear is partially blocked, resulting in significantly reduced sound on that side until the 'H' key is pressed or 10 seconds elapse. During the time spent in the water, a distorted wave sound is played to simulate being underwater, also working as a directional sound but in the Y axis. This effect is achieved by applying a parametric equaliser with emphasised low frequencies, along with a compressor and reverb.

3.8.2 3D Sounds

Creating 3D sound involves modifying the sound creation process to incorporate both an emitter and a listener. The listener corresponds to the player, represented by the camera's position, while the emitter signifies the source location of the sound. Subsequently, specific sound parameters are adjusted to constrain the listening range to the intended distance. Alternatively, the sound could have been configured to act as a focal point by manipulating other parameters, but this approach did not align seamlessly with any specific area within the scene constructed. This sound can be found in the scene at the top of the lighthouse.

3.9 Skybox Rendering

Following the instructions provided in one of the theory sections, a skybox has been developed. To achieve this, the normals of a cube are inverted so that the texture is visible from the inside. Although in my project the texture is visible from both sides, it may not be necessary, but if the rasterization changes, it would be necessary. Then, when rendering the cube, the depth buffer ID is disabled for this mesh so that it is not taken into account, allowing it to be drawn at a constant distance in the background.

3.10 Postproccesing effects

This part of the project actually does not fit at all with the actual scene due to the size of the objects, which are limited by the number of vertexes they can have, and it will look better in a much bigger scene.

3.10.1 Depth of Field

This postprocessing effect gives focus to close objects in the project, trying to recreate the effect when you are in the desert and you cannot see the far objects very well.

To make this effect, there is a pixel shader that checks how far the pixel is from the camera using the depth buffer, and in case the pixel is inside the threshold, it is mixed with the ones that are close to him, in this case, with 9 pixels. This effect is done following the

Separable Blur (Barsky and Kosloff, 2015) algorithm, which is a Blur effect done first in the X-axis and then in the Y-axis, this is done to reduce the cost of the function and make it more efficient.

4 Limitations

Regarding the framework's foundation, there are limitations on the number of vertices allowed for 3D models, restricting the size of the scene. Increasing the size of the ground and sea would result in an imperfect display of the effect with numerous peaks. Additionally, when it comes to added functionalities, the custom geometry created is also limited to 2048 points. Thus, when getting too close to it, a noticeable lack of points becomes apparent.

Due to the post-processing used, the screen only looks good in a small window. Changing it to full screen does not work properly, and there does not seem to be a simple way to fix it. Moreover, the sound does not have many functions, so implementing effects on the sounds like an equaliser or reverb within the game is not possible, and a different library would need to be used to create the effects in the code.

Additionally, the scene does not allow for more shadows or lights since each shader has a light as input, requiring modifications in too many parts of the code to add more lights and shadows. Furthermore, the shadow mapping works in a single direction, so it is not possible to create a global light that maps in all directions. To achieve this, rendering the scene in all directions would be necessary, but this is an option that both the light and shaders lack.

5 Future work

Looking ahead, it could be possible to enhance the shadow system to address aliasing issues. Efforts might also be directed towards overcoming current project limitations to make the scene larger. Additionally, physics could be integrated into the project to achieve a more realistic effect for the boat on the water and collisions between the water, the dock, and the lighthouse.

Furthermore, it could be worthwhile to incorporate a particle system to simulate the movement of sand by the wind or to add effects to the lighthouse glass (Zink and Hoxley, 2011). This functionality was started but was not completed in time. And finally, the lighting system could be improved to address the limitations explained earlier.

References

Barsky, B. and T. Kosloff (2015). "Algorithms for Rendering Depth of Field Effects in Computer Graphics". In: *https://cgvr.cs.uni-bremen.de/teaching/cg_literatur/*. Accessed: 25 November 2023.

Imperat (2016). "Rendering ocean water with C++ and DirectX11". In: *https://github.com/Imperat/DirectX-CourseWork-2016/tree/master*. Accessed: 25 November 2023.

Microsoft (Aug. 2020a). "Common Techniques to Improve Shadow Depth Maps". In: *https://learn.microsoft.com/en-us/windows/win32/dxtecharts/common-techniques-to-improve-shadow-depth-maps*. Accessed: 25 November 2023.

— (Aug. 2020b). "Configuring Blending Functionality". In: *https://learn.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-blend-state*. Accessed: 25 November 2023.

— (Jan. 2022). "Getting Started with the Rasterizer Stage". In: *https://learn.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-rasterizer-stage-getting-started*. Accessed: 25 November 2023.

opengl-tutorial (2018). "Tutorial 13 : Normal Mapping". In: *https://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/*. Accessed: 25 November 2023.

Rastertek (2019). "Tutorial 40: Shadow Mapping". In: *https://github.com/matt77hias/RasterTek/tree/master/Tutorial%2040_Shadow%20Mapping*. Accessed: 6 December 2023.

Walbourn, C. (Nov. 2022). "Creating and playing sounds". In: *https://github.com/microsoft/DirectXTK/wiki/Creating-and-playing-sounds*. Accessed: 25 November 2023.

Zink J. Pettineo, M. and J. Hoxley (2011). *Practical rendering and computation with Direct3D 11*. CRC Press LLC.