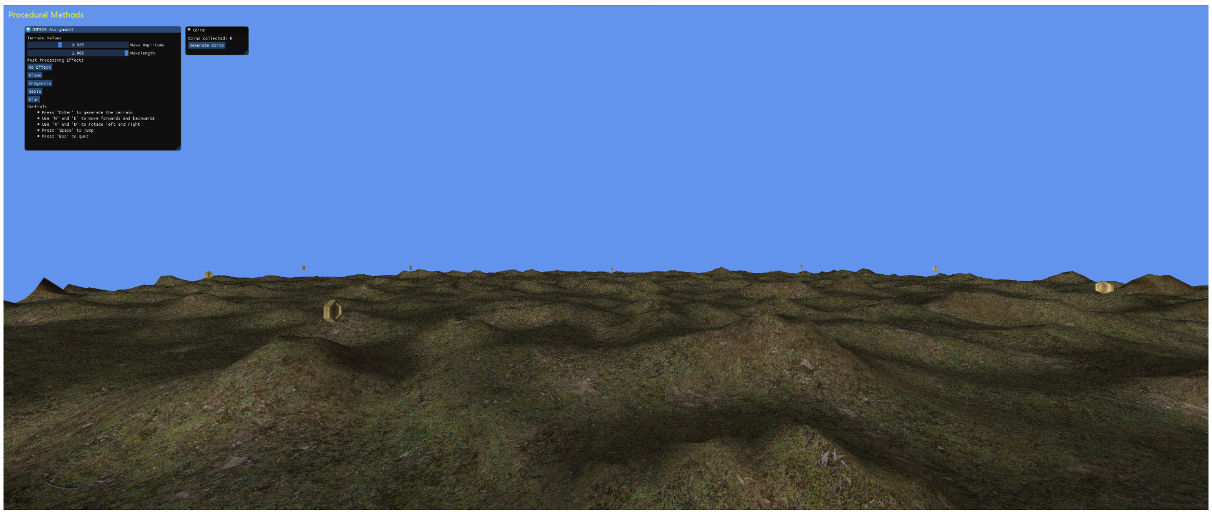
Kayla Wallace 2101726

CMP505: Advanced Procedural Methods

Introduction

The aim of this project is to create a playable computer game using DirectX 11 and DirectX Toolkit 2015. It must include procedural content generation, post-processing, and an advanced feature. This project showcases a live, procedurally generated terrain in which the player can walk and jump on, and scene bounds will not allow the player to walk off the terrain. The aim is for the player to collect all of the coins scattered across the terrain. The game is displayed in first-person. The player can alter the amplitude and wavelength of the terrain and regenerate it as many times as they would like. They can also choose to regenerate the collectable coins once they have all been collected. The player can also choose between four different post-processing effects to apply to the scene: monochrome, sepia, bloom blur, and gaussian blur. All of these actions can be achieved using the Dear ImGui displays.



0.935 Wave Amplitude
1.860 Wavelength

- Press 'Enter' to generate the terrain
- Use 'W' and 'S' to move forwards and backwards
- Use 'A' and 'D' to rotate left and right
- Press 'Space' to jump
- Press 'Esc' to quit

Coins collected: 6

Figure 1: Image of the final project

Controls

- 'W' and 'S' to move forwards and backwards
- 'A' and 'D' to rotate left and right
- 'Space' to jump
- 'Enter' to generate the terrain
- Walk into coins to collect them
- GUI sliders to change the terrain amplitude/wavelength
- GUI buttons to regenerate coins and choose post-processing effects

Implementation: Special Features

Jump

The special features implemented in this project are the player's 'jump' feature and the collision detection with the terrain. The jump feature is achieved using physics. There is a constant gravitational force applied to the player object, which is then also applied to the camera to maintain a first-person view. Then, tying into the collision detection with the terrain, a boolean *'grounded'* variable is set when the player is colliding with the terrain within a specific range. When the player presses the jump button, a large upwards force is applied. Each of these forces also use *'delta time'* to achieve frame rate dependent movement. To ensure the player cannot hold the jump button to continuously add this upwards force and fly, the Player class also has a boolean *'jumping'* property that is set to true when pressed. Then the player is only allowed to jump if that property is false, and the grounded property is true. The resetting of these two properties is handled in the collision detection of the player with the terrain.

Collision Detection
The collision detection between the player and the terrain is the second special feature implemented. For this, a ray-triangle intersection function was implemented. This is achieved in several separate steps. First, a *'Triangle'* class was created. This simply allows a *Triangle* object to hold the positions of its three vertices. Then in the terrain class, a vector of *Triangles* is initialised. When the terrain's buffers are initialised, the positions of the triangles are set and pushed back onto a vector of triangles, which is emptied before this process so that the terrain can be continually generated by the player and the triangle positions will remain accurate. Then, a '*RayCollisionDetection'* class was created. The ray-triangle intersection method was from Ray Tracing: Rendering a Triangle (Ray-Triangle Intersection: Geometric Solution), n.d.. A ray is cast downwards from the position of the camera, and the three vertices of a triangle are used to perform these calculations. First the triangle's normal is calculated, and then checks are run that determine if the triangle and ray are parallel.

$$
\begin{aligned}
&Ax + By + Cz + D = 0\\
&D = -(Ax + By + Cz)
\end{aligned}
$$

$$
\begin{aligned}
&P = O + tR\\
&Ax + By + Cz + D = 0\\
&A * P_x + B * P_y + C * P_z + D = 0\\
&A * (O_x + tR_x) + B * (O_y + tR_y) + C * (O_z + tR_z) + D = 0\\
&A * O_x + B * O_y + C * O_z + A * tR_x + B * tR_y + C * tR_z + D = 0\\
&t * (A * R_x + B * R_y + C * R_z) + A * O_x + B * O_y + C * O_z + D = 0\\
&t = -\frac{A * O_x + B * O_y + C * O_z + D}{A * R_x + B * R_y + C * R_z}\\
&t = -\frac{N(A,B,C) \cdot O + D}{N(A,B,C) \cdot R}
\end{aligned}
$$

Figure 2: Two plane equations (Ray Tracing: Rendering a Triangle (Ray-Triangle Intersection: Geometric Solution), n.d.)

Then both *D* and *t* are calculated in order to compute the intersection point of the ray and triangle using the equations of a plane as shown above in Figure 2. In the code, *D* is calculated by taking the dot product of the triangle's normal and a point on the triangle, and *t* is calculated exactly as shown above, with *R* being the direction of the ray. Then using these, the point of intersection is calculated. After this, checks are performed to ensure the point is inside the triangle and the triangle is not behind the ray. Providing all these checks pass, the ray is colliding with the terrain and true is returned.

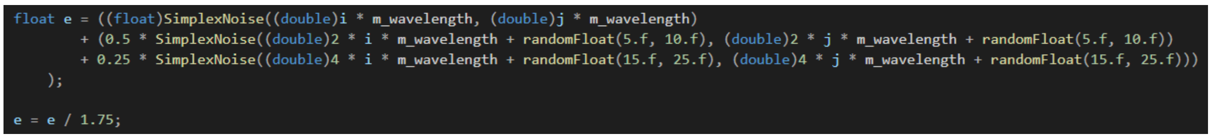
In the *Terrain* class, the above function is performed for each triangle in the terrain. If a hit is detected, then the intersection point is calculated in Barycentric coordinates (methods derived from Ray Tracing: Rendering a Triangle (Barycentric Coordinates), n.d. and Point within a triangle: barycentric co-ordinates, 2019) and this is used to compare the distance between the camera and its intersection point on the triangle. Then if this is in the range of a radius of 2, then the position of the camera on the y-axis is set to that of the intersection point plus 2, so that the camera maintains a position of slightly above the terrain at all times. It is at this point that the player's *'grounded'* property is set to true.

For the collisions between the player and coins in the scene, the "*DirectXCollision*" header was included so that types such as *BoundingSphere* could be used. In this case, a BoundingSphere was added to each coin and the player and then the *Intersects*() method from the *DirectXCollision* header was used to detect collisions between the two.

Implementation: Other Features

Terrain Generation

In the terrain class, the simplex noise function from Gustavson (2005) was implemented and used for the generation of the terrain. This works by picking a pseudo-random gradient at each coordinate point on the terrain, which is blended with its neighbours in a similar way to bilinear interpolation. The input space also has to be skewed to determine which simplex cell is currently being operated on, and then unskewed for further calculations. This function is then called in a nested for loop, iterating through the entire width and height of the terrain, generating the height map. At this point, some further improvements were made to the terrain, deriving from Making maps with noise functions (2015). In doing so, three octaves were added to the generation of the terrain to give the hills a more rugged appearance.



```
float e = ((float)SimplexNoise((double)i * m_wavelength, (double)j * m_wavelength)
        + (0.5 * SimplexNoise((double)2 * i * m_wavelength + randomFloat(5.f, 10.f), (double)2 * j * m_wavelength + randomFloat(5.f, 10.f))
        + 0.25 * SimplexNoise((double)4 * i * m_wavelength + randomFloat(15.f, 25.f), (double)4 * j * m_wavelength + randomFloat(15.f, 25.f)))
    );

e = e / 1.75;
```

Figure 3: Octaves applied using the SimplexNoise function

This value, *e*, with the octaves of generation applied was then divided by 1.75 to ensure it stayed within the range of 0 to 1. Then, in order to give some parts of the terrain a flatter appearance, a redistribution factor was added. The higher the redistribution factor, the more flat land will occur within the terrain. To achieve this, a curve function must be used to push/pull the middle elevation values of the terrain up/down. In this case, the power function was used, where *e* is raised to the power of the redistribution factor. Additionally, a *'fudge factor'* was added into this process to keep the calculations consistent.

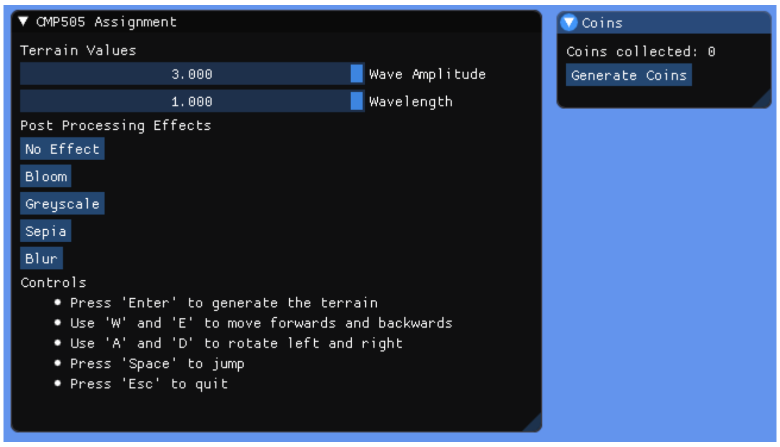
Additionally, texture blending based on the height and slope of the terrain was implemented in the pixel shader for the terrain (derived from DirectX11: Height based texture blending, (2015)). Four constants were set for this process, the upper and lower height bounds and the upper and lower slope bounds. First, the texture colour was set depending on the slope of the terrain. If the slope was lower than the lower slope bound, then the texture was linearly interpolated (lerped) between the middle colour and the low colour. If the slope was in between bounds, the texture colour was lerped between the low colour and the middle colour. If the slope was higher than the upper bound, the texture colour was set to the middle colour. With the lower colour being a dirt texture and the middle colour being a grass texture, this provided a much more interesting and visually pleasing terrain. Next, the height values were checked against the height bounds, lerping the current texture colour with a snow texture colour if in between the bounds, and simply setting the texture colour to the snow texture if above the upper bound.

Post-Processing

For the post-processing effects within the project, the provided RenderTexture class was used alongside the DirectX *'PostProcess'* header. The scene is rendered to a texture in the *RenderTexturePass* method in the *Game* class, and then this is used to create the post-processing effects in the *Render* method. The render pass is set as the source texture for the post-process effect, and then using the built-in functions of the *PostProcess* header mentioned above, effects such as bloom blur, monochrome, sepia, or gaussian blur are applied according to player input. Additionally, a 'copy' effect is available when using this header, and this is used when the player opts for 'no effect'.

Dear IMGUI

The *'ImGui'* graphical user interface library was used to display the GUI seen on the screen when running the project. This allowed for the implementation of two small GUI windows in the project. The first allows the user to alter the amplitude and wavelength of the terrain, choose between all the post-processing effects available, and view the controls for the game. The second allows them to see how many coins they have collected, and to regenerate the coins on a button press once they have collected all the current coins in the scene.



```
▼ CMP505 Assignment
Terrain Values
            3.000            Wave Amplitude
            1.000            Wavelength
Post Processing Effects
No Effect
Bloom
Greyscale
Sepia
Blur
Controls
    • Press 'Enter' to generate the terrain
    • Use 'W' and 'E' to move forwards and backwards
    • Use 'A' and 'D' to rotate left and right
    • Press 'Space' to jump
    • Press 'Esc' to quit

▼ Coins
Coins collected: 0
Generate Coins
```

Figure 4: Image of the GUI shown in the project

Coin Collection System

The coin collection system is intertwined with the *Game* class. Upon initialisation of the scene and each generation of the terrain, the *GenerateCollectables* function is run, which generates 10 coins at random x and z coordinates on the terrain, initialises the *BoundingSphere* collider for each, checks the coin is not colliding with the terrain, and adds the coin to a vector of *Coin* objects. The collision detection of the coins is discussed above, and when this returns true, the coin will cease to be rendered and will be erased from the vector of *Coin* objects. To add some life to the coins, a sine function is used in the *RenderCollectables* method to apply a bobbing motion to each coin, and a rotation on the y-axis is created using time to apply a constant rotation to the coins.

```
float newY = sin(m_timer.GetTotalSeconds() * 3.f) * .25f;
Matrix position = Matrix::CreateTranslation(Vector3(coin.GetPosition().x, coin.GetPosition().y + newY, coin.GetPosition().z));
```

Figure 5: Sine function applied to the coins, where 3 is the speed and 0.25 is the height of the bobbing motion.

Code Organisation

Classes:

- Camera
- Light
- ModelClass
- RenderTexture
- Shader
- Terrain
- Collectable
- Game
- Input
- Main
- Player
- RayCollisionDetection
- Triangle

Critical Appraisal

Code Organisation

In terms of code organisation, more could have been done to separate the collectable coins from the *Game* class. The *Coin* class stores all the data and getters and setters for a single coin, but a *CoinManager* class could have been implemented to store the vector of coins in the scene, and perform the generation and collision detection for the coins, as opposed to these methods being a part of the *Game* class. This would have allowed for more organised code and less coupling in the code, keeping the *Game* class separate from the collectables entirely. In addition to this, the player's jump should have been handled in the *Player* class instead of the *Game* class - further improving the structure and organisation of the code. Besides this, the code is quite organised in terms of class structure.

Strategies

Throughout this project, a large amount of time was spent on features that didn't make it into the final game, such as a third person camera. Due to an approach of trying to implement a feature first, and then researching afterwards, this meant that a lot of time would have already been spent on said features before realising that it was difficult to find resources to help that were easily translatable to the framework being used. This was a very unproductive strategy that cost a large amount of time to the project that could have been spent improving the features that already worked. In addition to this, the decision to add physics to the game was made very late into development. In future, more planning should be done at the beginning of the project so that critical decisions such as this are not left too late, leaving the mechanic slightly lacking.

Physics

The jump physics in the project does not work in a very polished manner and is in a very basic form. The gravitational force works well. Although it is a bit floaty, this allowed for a reduction of the jittery movement that occurs when jumping. The upwards force is applied all at once when the jump button is pressed, which is a jarring motion for the camera. It is believed this is due to the way that the 'jumping' and 'grounded' properties of the player are set, alongside the collision detection, meaning the upwards force isn't gradually applied. This could have been improved with the implementation of a velocity variable. This could have allowed for the gradual addition of upwards force while the velocity was positive, and more control over the falling state of the player as well. However, the addition of calling the *Camera Update* function in the *Render* method of Game as opposed to the *Update* method of *Game* acted as a late update for the camera, allowing for much smoother movement along the terrain. As with the addition of gravity and jumping, it had become slightly jittery at times. This made a big difference to the polish of the final project.

Collision Detection

In general, the ray-triangle collision detection in this project is not very efficient. Improvements could have been made to this by calculating the normal of the triangle only once, upon generation of the terrain, and storing this in the *Triangle* class, as opposed to calculating the normal to the triangle each time the collision detection was run. Furthermore, the terrain could have been subdivided such that the collision detection would only be running through a half or a quarter of the triangles in the terrain at a time. This too would have improved the efficiency of the collision detection. These factors influenced the decision to use the DirectXCollision helper class for the coin collection mechanic. In order to avoid further performance loss by casting more rays in the scene and being under the assumption that using this would be more efficient than using/adapting the ray-triangle collision detection already written, this helper class was chosen as it provided pre-defined collider data types and methods to detect intersection. However, the collision detection implemented is very consistent and effective.

Procedural Content
When using a simplex noise function to generate the terrain, even with a smoothing algorithm, the terrain was very rough and jagged. Although this is not good for performance, the decision was made to double-smooth the terrain, calling the smoothing function twice after each generation of the terrain. This allowed for time saved to work on other aspects of the project and there was not a noticeable change in performance. This approach provided much smoother hills and more space between high and low hills on the terrain. In addition to this, texture blending was implemented in the pixel shader for the terrain. This was a great way to improve the appearance of the terrain without involving much computational power. This looks much better at low amplitudes, providing an almost realistic looking terrain, but can look jagged at high amplitudes. Overall, the procedural content generation in the project may not be the most efficient but provides a decent result in the end.

Reflection
One of the main takeaways from this project is that procedurally generating a terrain is not difficult given the resources available. However, taking that terrain/noise function and altering it so that the terrain looks objectively 'good' or realistic is very difficult. Through making small changes to the inputs/outputs of the simplex noise function in this project and the implementation of texture blending, the appearance of the terrain was significantly improved but by no means looked game-ready or realistic. Additionally, even with the understanding of the underlying maths of collision detection, this is very difficult to translate into code. The circumstances in which collision detection is used is never as 'simple' as in mathematics problems and applying this knowledge is something that was found very difficult throughout this project.

Conclusion

The aim of this project is to create a playable computer game using DirectX 11 and DirectX Toolkit 2015. This project showcases live procedural terrain generation, basic physics, post-processing effects, and collision detection allowing for first-person traversal of the terrain and the collection of coins. Throughout the process of implementation, specifically the coin collection system, the *Game* class became more bloated and if more time was available, the coin collection system would be migrated into its own manager class to reduce the coupling in the *Game* class. Further, the physics implemented is very basic and does not work in a polished manner. Again, had more time been available, this could have been improved with the use of a velocity value in the player movement. A lot of development time was spent on features that had been under-researched by the developer and so never made it into the final project. In future, more planning should be done at the beginning of development to allow for more refined and developed mechanics to make it into the final project. However, the effectiveness of the collision detection is a fine addition to the project. It is consistent and ties in well with the gravitational force applied to the player. The terrain is not the most realistic looking terrain at high amplitudes, but at low amplitudes the texture blending provides great visual interest to the terrain.

References

Gustavson, S. (2005) *Simplex noise demystified* Linköping University, Sweden
Red Blob Games. (2015). *Making maps with noise functions*. [online] Available at: <https://www.redblobgames.com/maps/terrain-from-noise/>
Scratchapixel.com. n.d. *Ray Tracing: Rendering a Triangle (Barycentric Coordinates)*. Available at: <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates>.
Scratchapixel.com. n.d. *Ray Tracing: Rendering a Triangle (Ray-Triangle Intersection: Geometric Solution)*. Available at: <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution>.
Stack Overflow. 2015. *DirectX11: Height based texture blending*. [online] Available at: <https://stackoverflow.com/questions/34344677/directx11-height-based-texture-blending>
Stack Overflow. 2019. *Point within a triangle: barycentric co-ordinates*. Available at: <https://stackoverflow.com/questions/25385361/point-within-a-triangle-barycentric-co-ordinates>.