CMP305- Procedural Methods
– Coursework –

Francisco Diaz
1902654

Contents

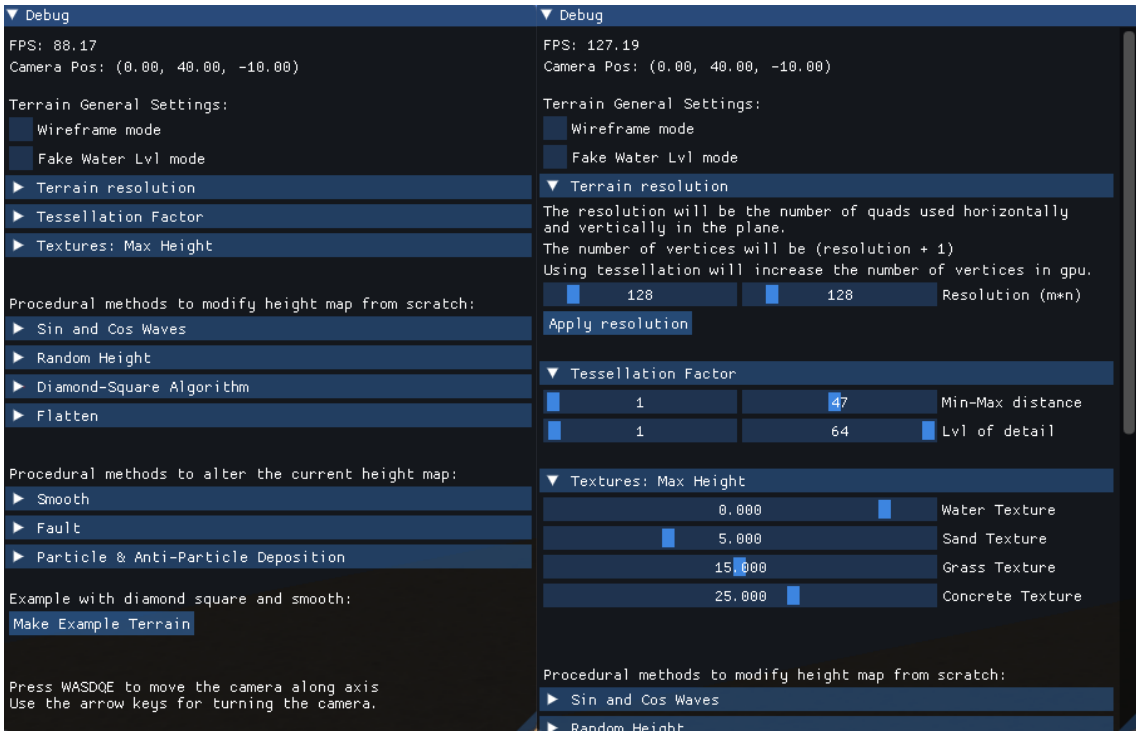
1. How to use the program

1.1. Controls

Initially the application camera is looking at the plane.
Pressing the keys WASDQE the camera will move forward, left, backward, right, up, and down respectively along the axis. To rotate the camera, use the arrow keys or the mouse.

1.2. UI

The application uses IMGUI to easily manipulate the parameters which affect to the procedural methods and to run these.
The UI has been split in different sections: On the top there is information about fps and camera position followed by the terrain general settings to modify the resolution, tessellation factor and the height of each texture in the terrain. Additionally, there are two toggles which will activate the wireframe mode or will fake the water level (levelling up the water vertices to 0).
In the middle of the UI you can find the actual procedural methods which are divided in two parts, firstly the methods which modify the height of the terrain completely without taking into account any previous height and secondly the methods which modify the terrain taking into account the current height.
To finalise there is a button which run an example terrain with diamond-square algorithm followed by the smooth method. Clicking on this button will set a predefined values to run the methods (terrain resolution, texture heights, diamond-square height offset, faking water level) these values will be displayed on the UI automatically.



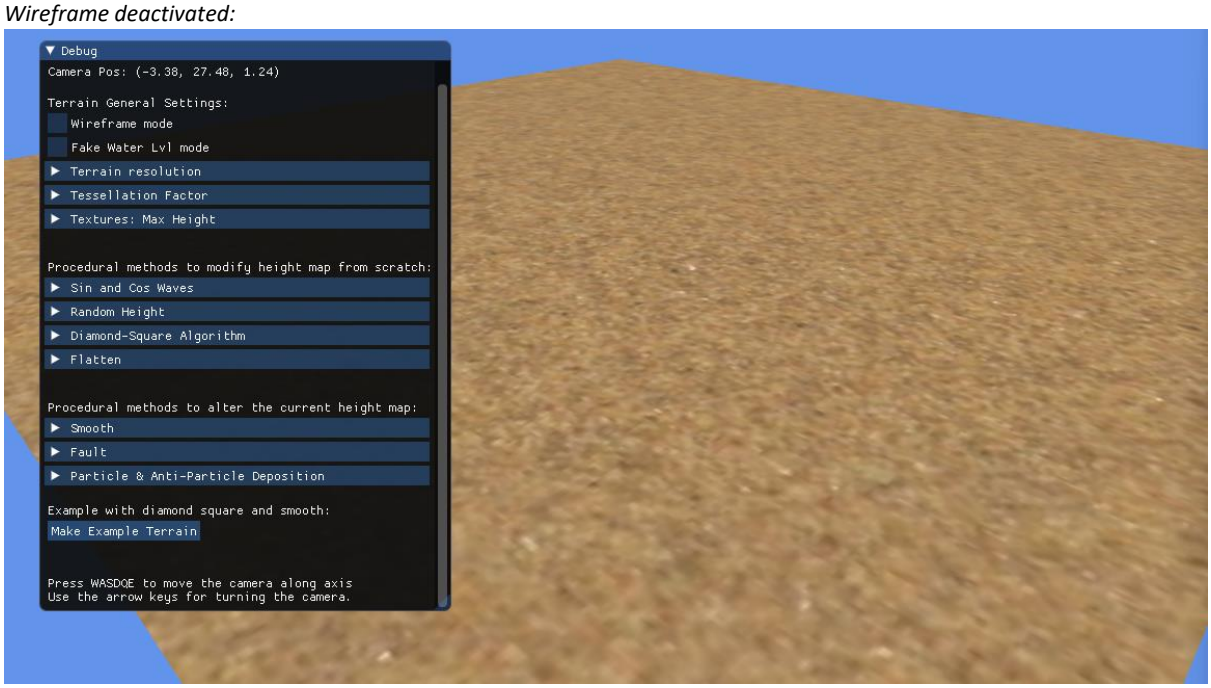
2. Outline of the general features

2.1. Terrain General Settings

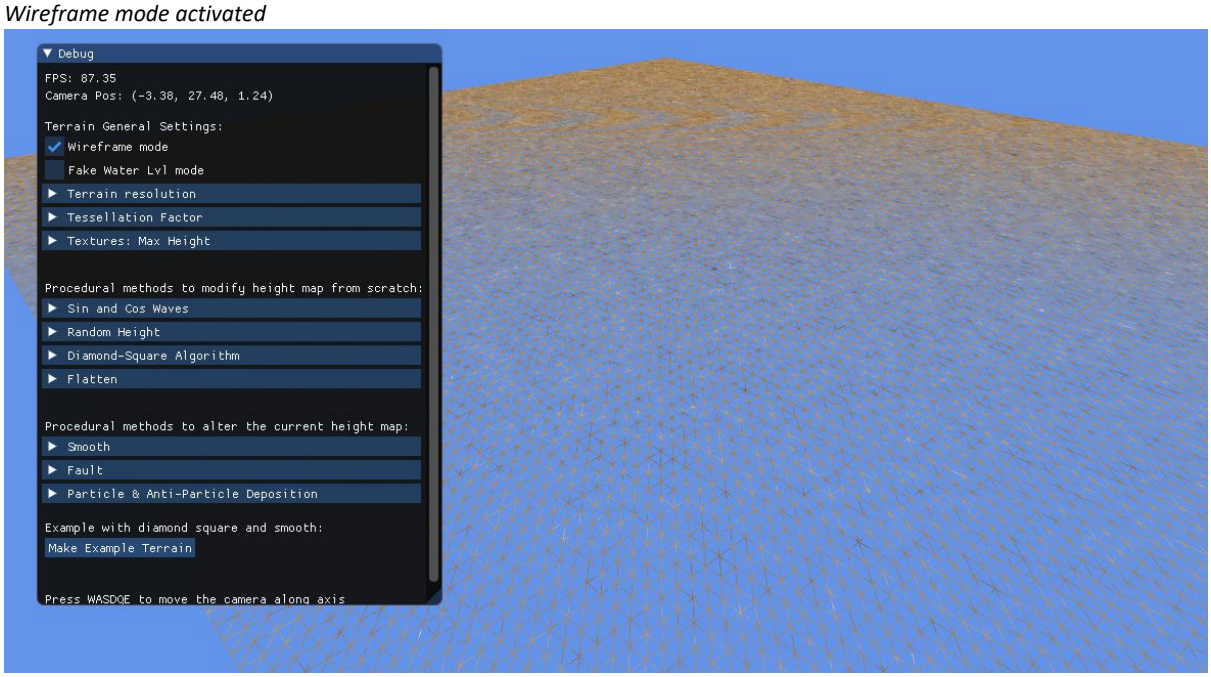
2.1.1. Wireframe mode

It allows to see the plane skeleton, lines which connect the vertices of the plane.

Wireframe deactivated:



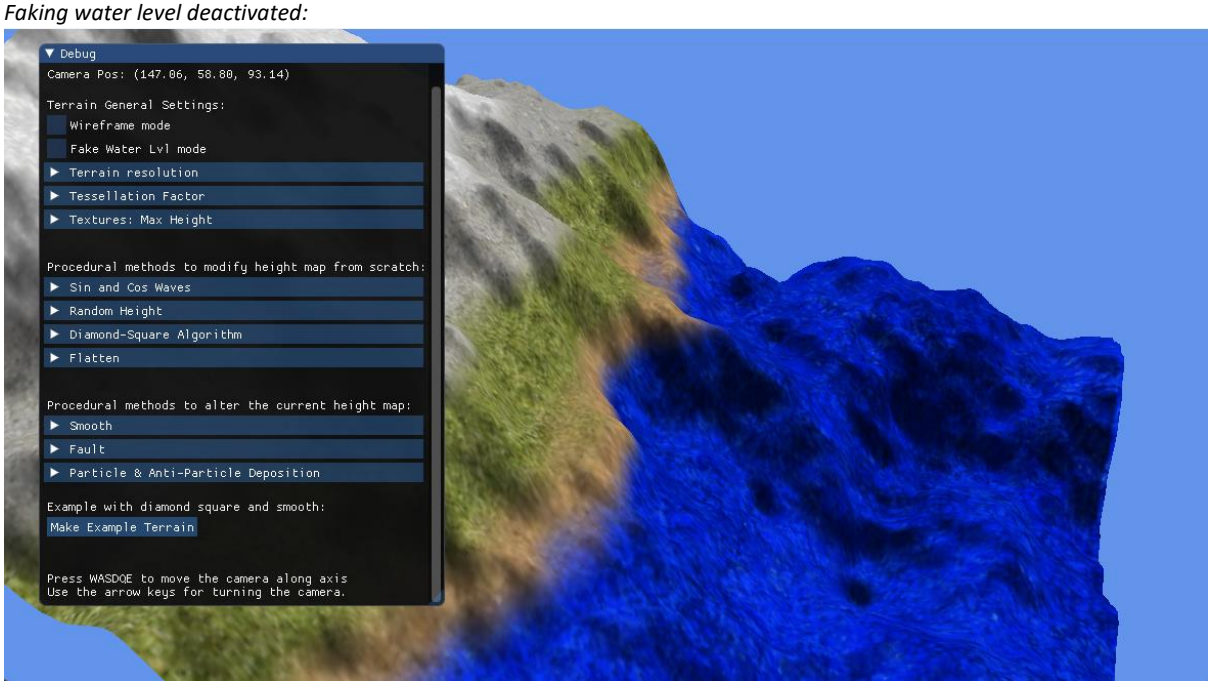
Wireframe mode activated



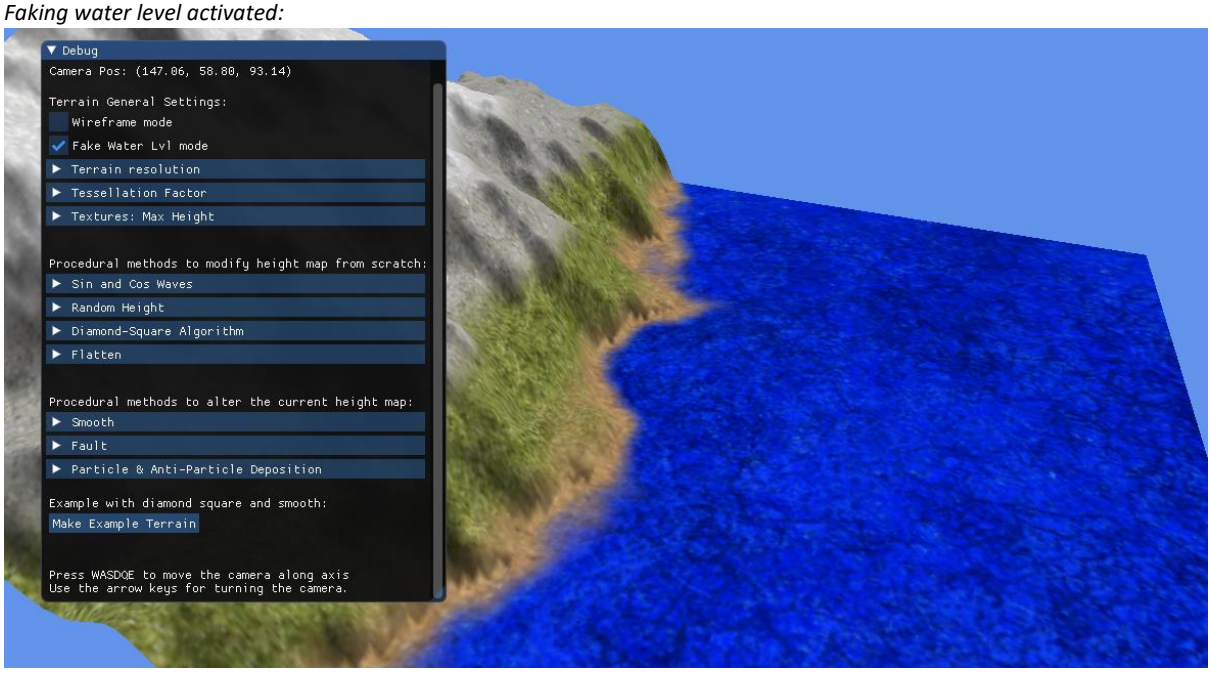
2.1.2. Faking Sea level

It elevates the terrain to zero where it is below zero.
This process is done in the domain shader, sending a float to detects if the user wants to fake the sea level or not. If they want, then the vertex in y is set to 0 and the normal is set facing up y = 1.
The intention of this feature is giving a more realistic effect for the purpose of this application. In a real game this should be done differently as the player might be able to dive in the sea for example.

Faking water level deactivated:



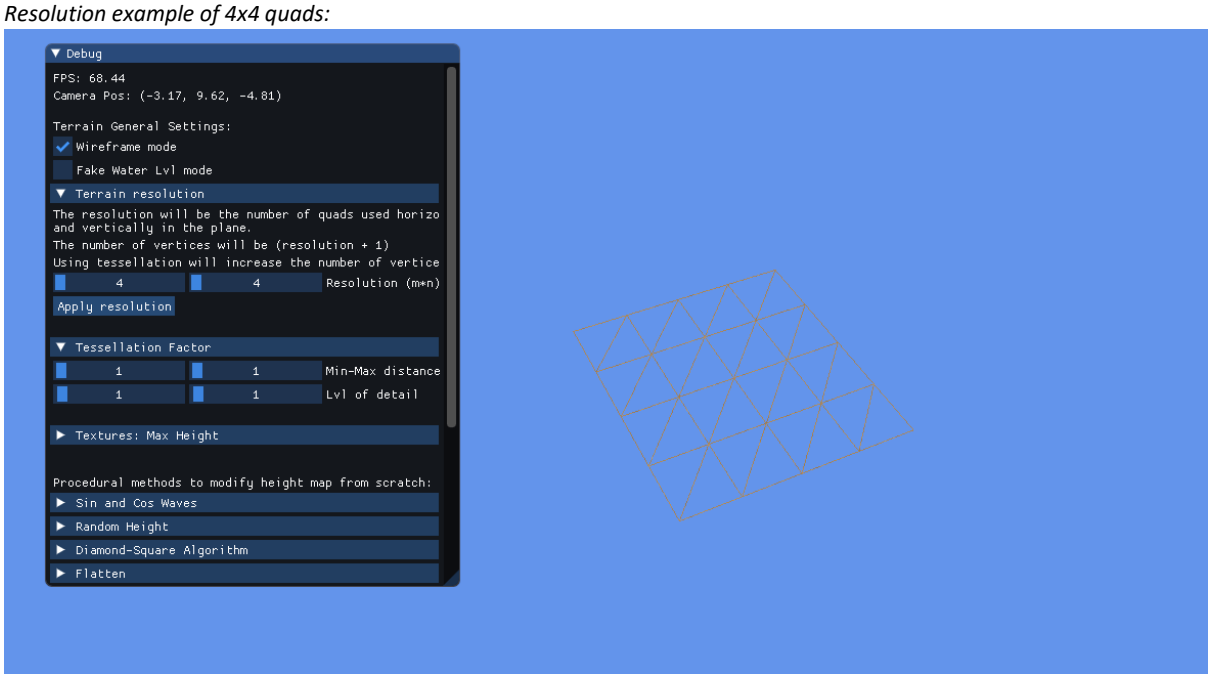
Faking water level activated:



2.1.3. Resolution

It is the number of quads initially used without considering the tessellation. The more quads used the more vertices the plane has, thus the higher is the resolution (higher detailed the plane is). Lineally speaking the number of vertices is resolution plus one.

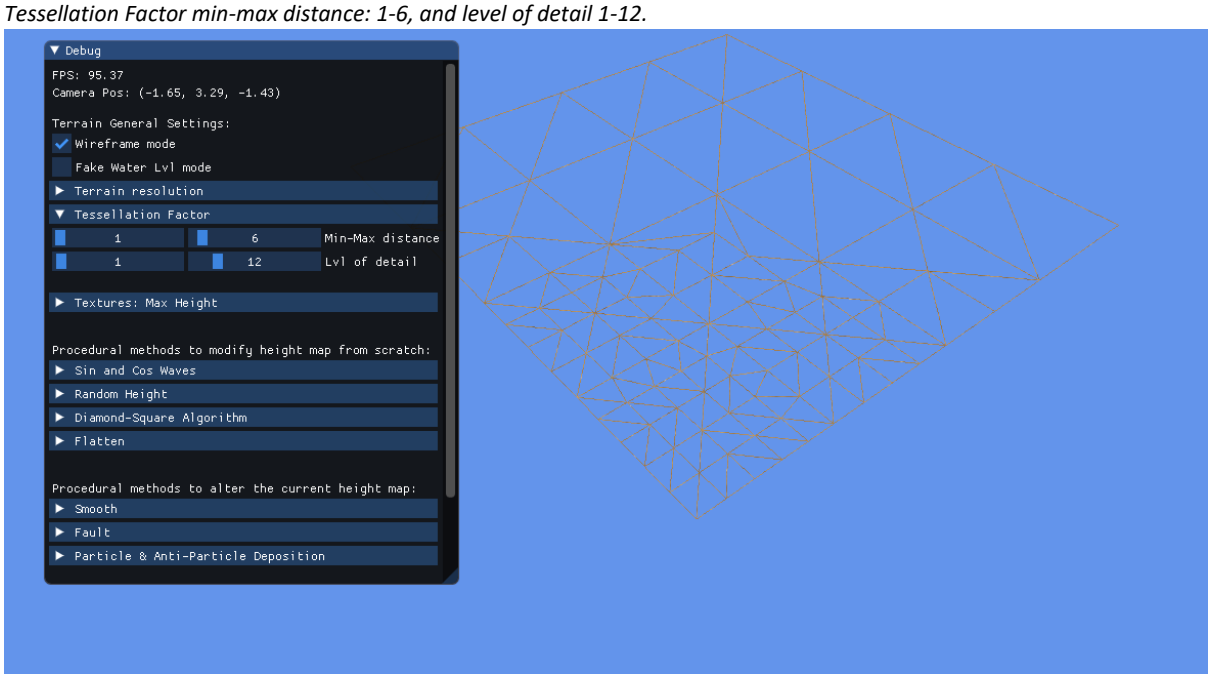
Resolution example of 4x4 quads:



2.1.4. Tessellation

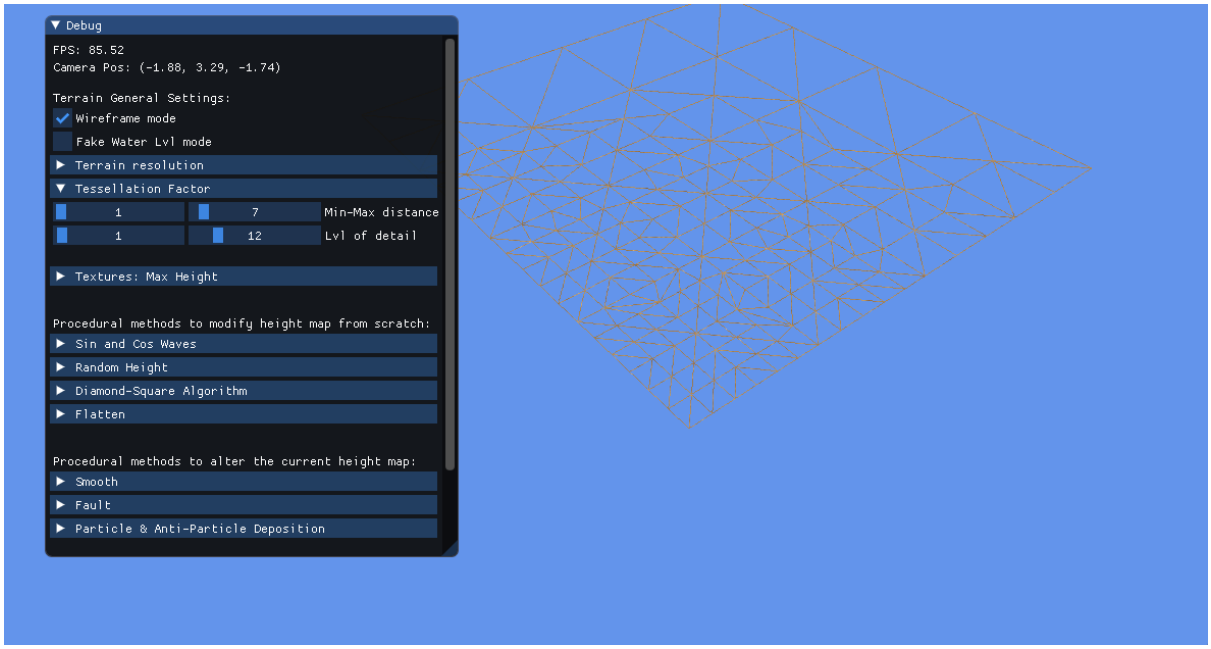
It adds details to the terrain by adding more vertices according to the camera position and the tessellation factor. This is done in the hull and domain shader. It uses 12 control points.

Tessellation Factor min-max distance: 1-6, and level of detail 1-12.

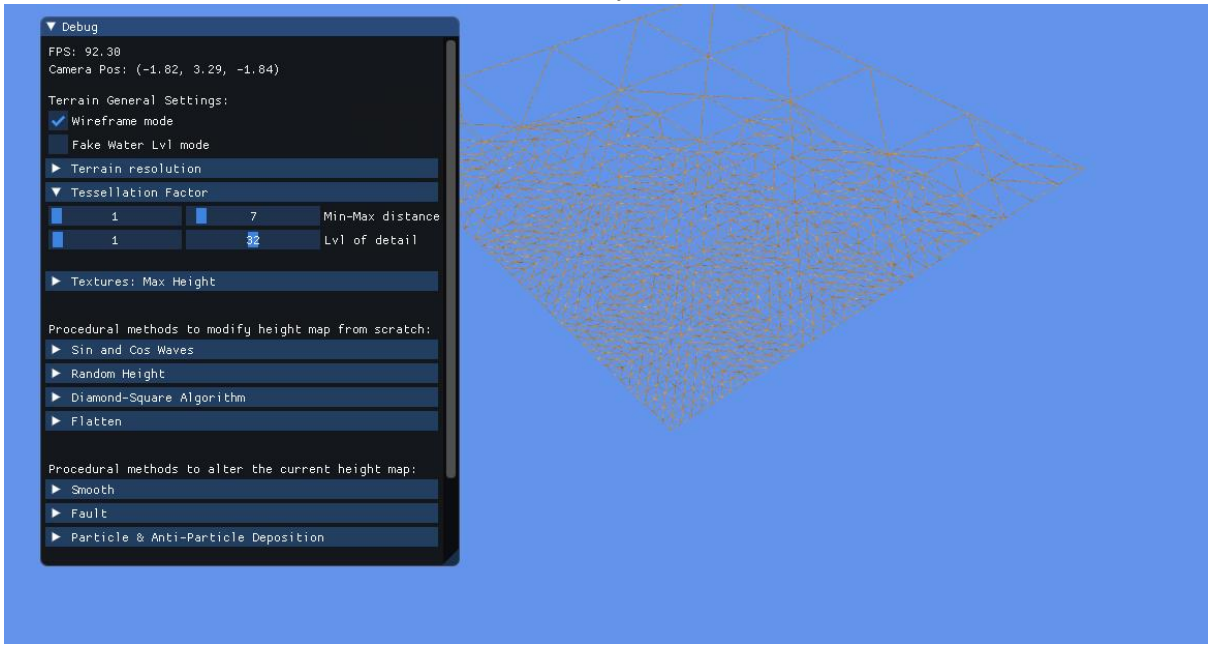


Incrementing the max distance will increase the distance limit to where the tessellation is done on the plane respect the camera position. Additionally increasing the level of detail will increase the number of vertices (triangles) from the camera position.

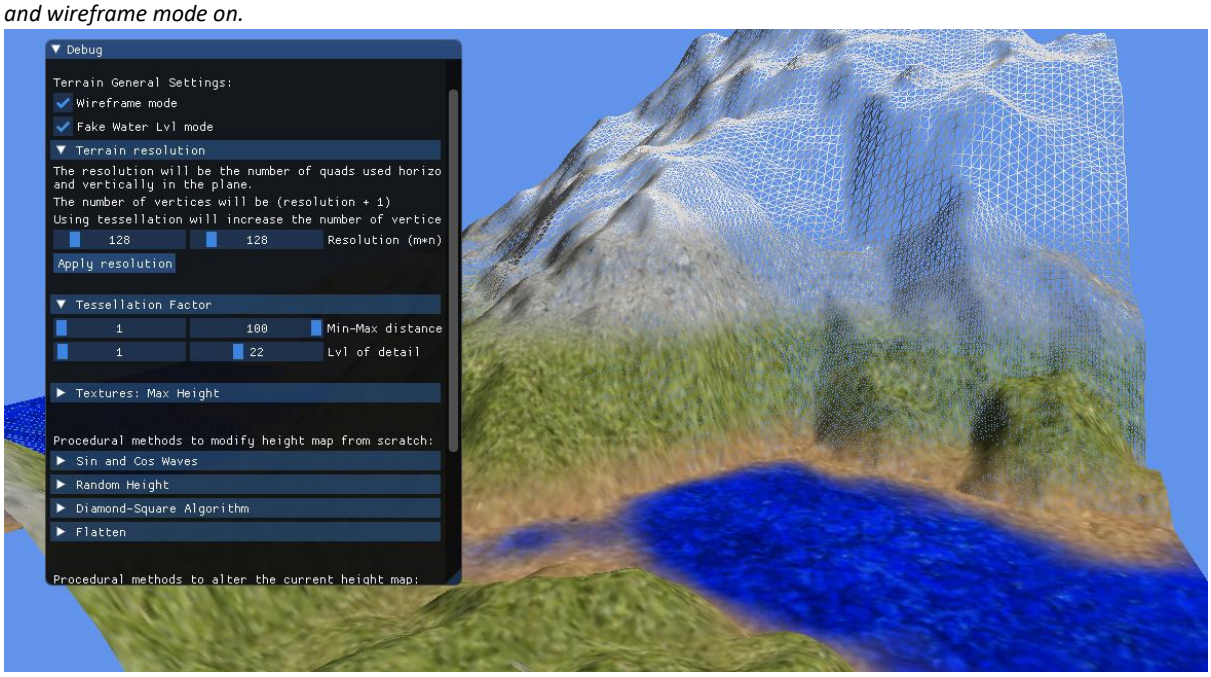
Tessellation Factor max distance increased to 7.



Tessellation Factor max distance increased to 7 and max level of detail increased to 32.



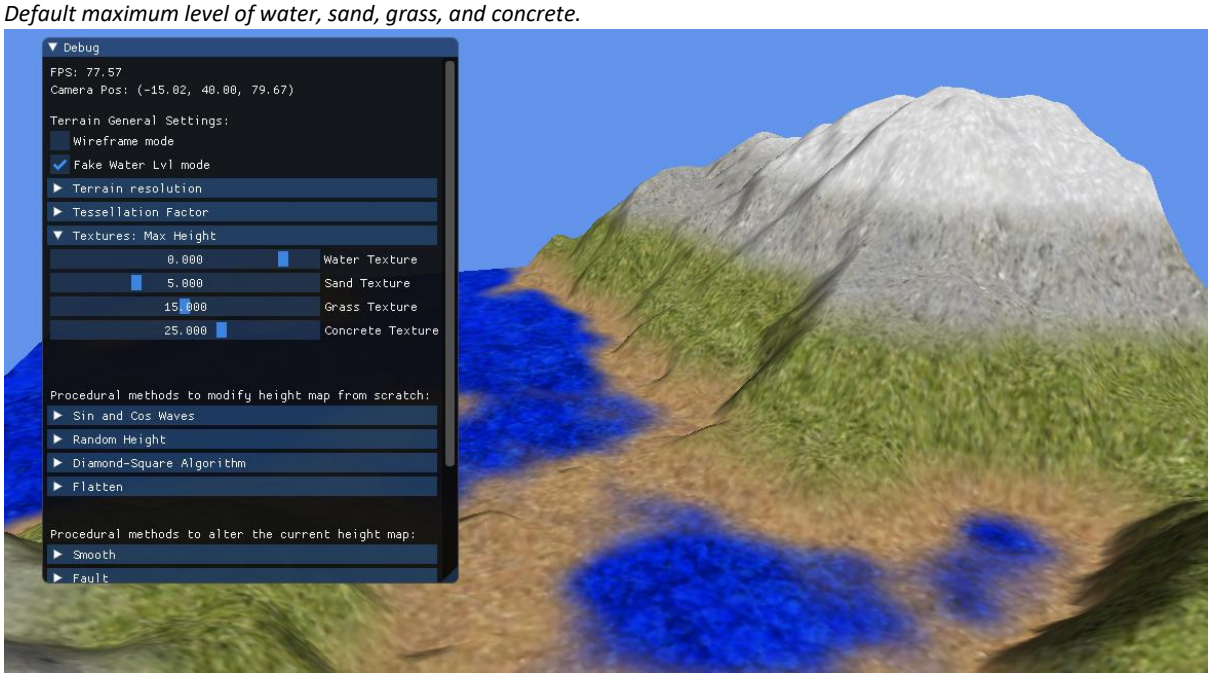
Tessellation Factor min-max distance 1-100, and min-max level of detail 1-22 in a terrain with procedural method executed and wireframe mode on.



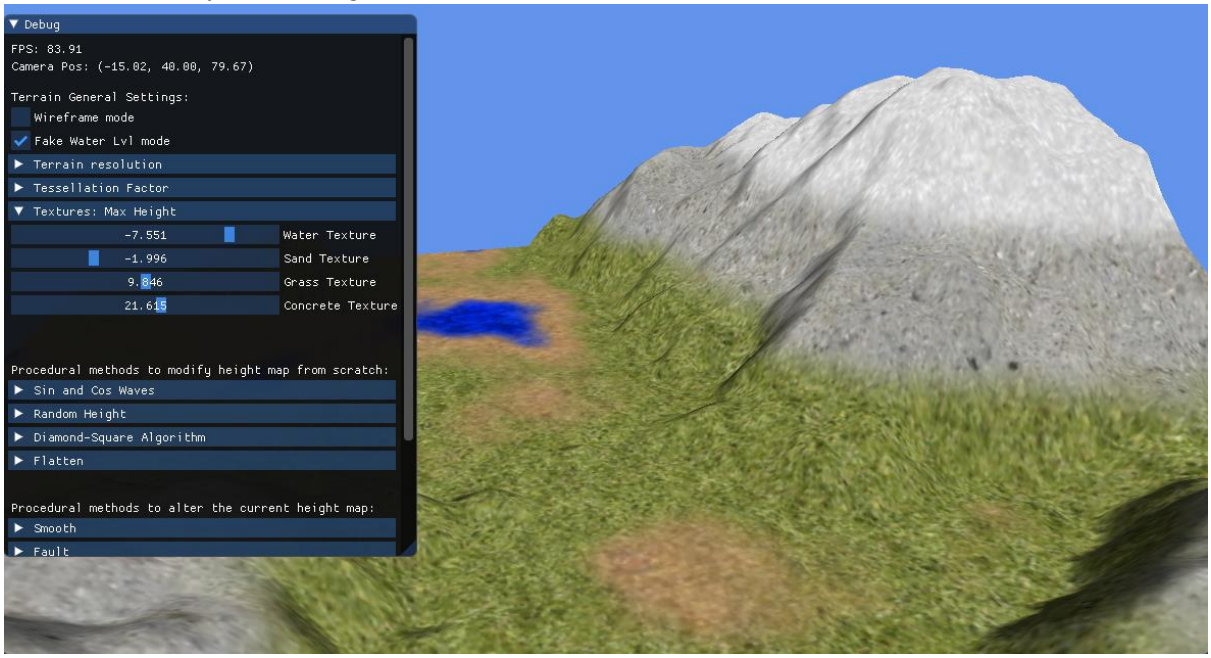
2.1.5. Changing texture levels

The application currently has four textures: water, sand, grass, concrete and snow. The maximum height the three first textures achieve can be changed using the UI, moving the textures in real time. This is done by passing the texture maximum heights to the pixel shader as a constant buffer. In this shader there has been created a function to use the maximum height to determinate what texture to use or if use a blend of two textures. The textures height limits in the UI are set dynamically depending on the neighbour textures so they do not overlap.

Default maximum level of water, sand, grass, and concrete.



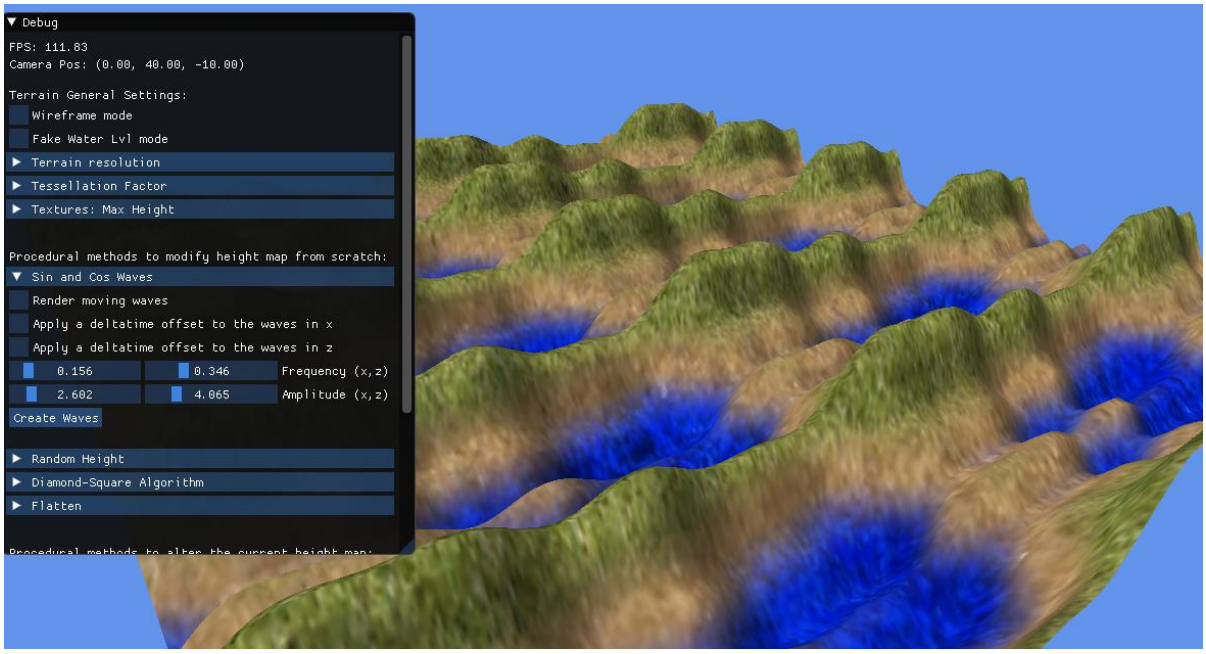
The maximum level of water, sand, grass, and concrete have been decreased.



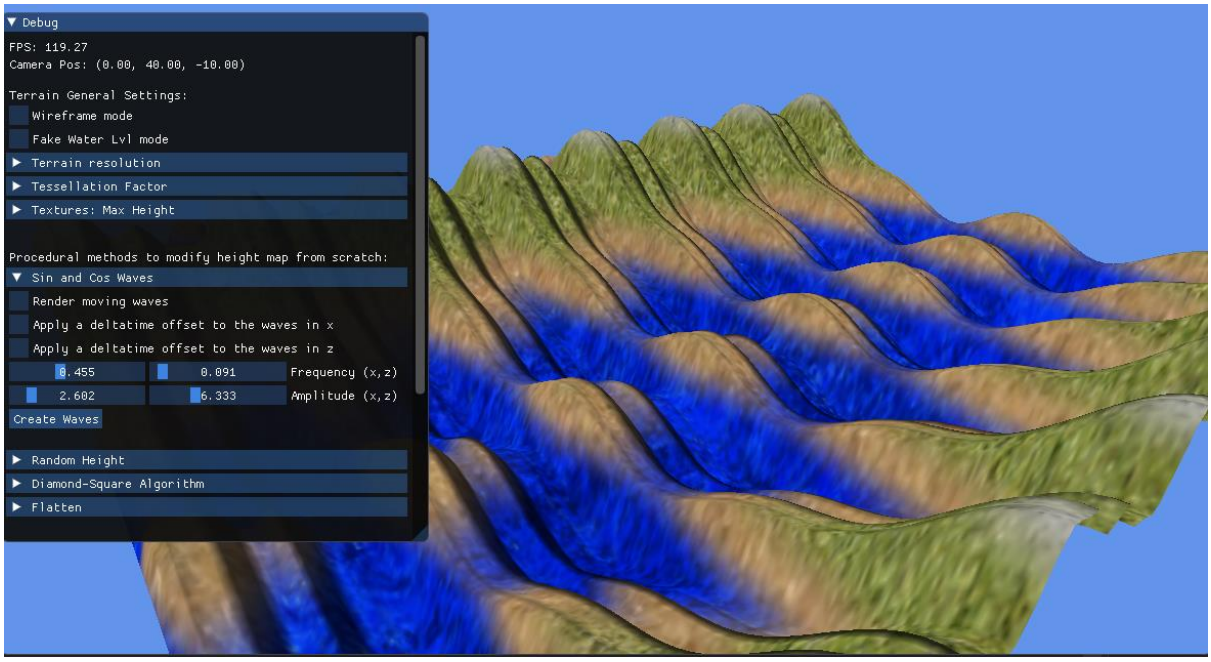
2.2. Procedural methods to modify height map from scratch.

2.2.1. Sin and Cos waves

It creates three sine waves effect along x-axis and three cosine waves effect along z axis.
Using the UI can modify the frequency and amplitude of sine and cosine.

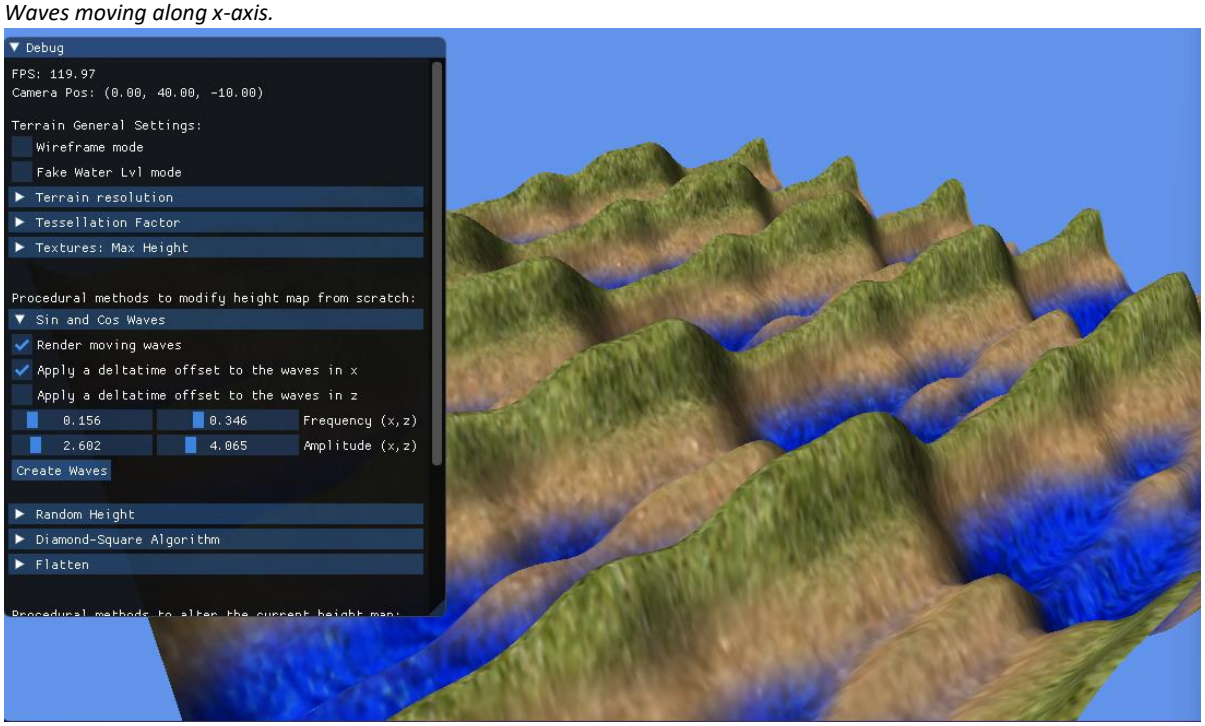


The frequency in x and z have been increased and decreased respectively. Furthermore, the amplitude in z has been increased.

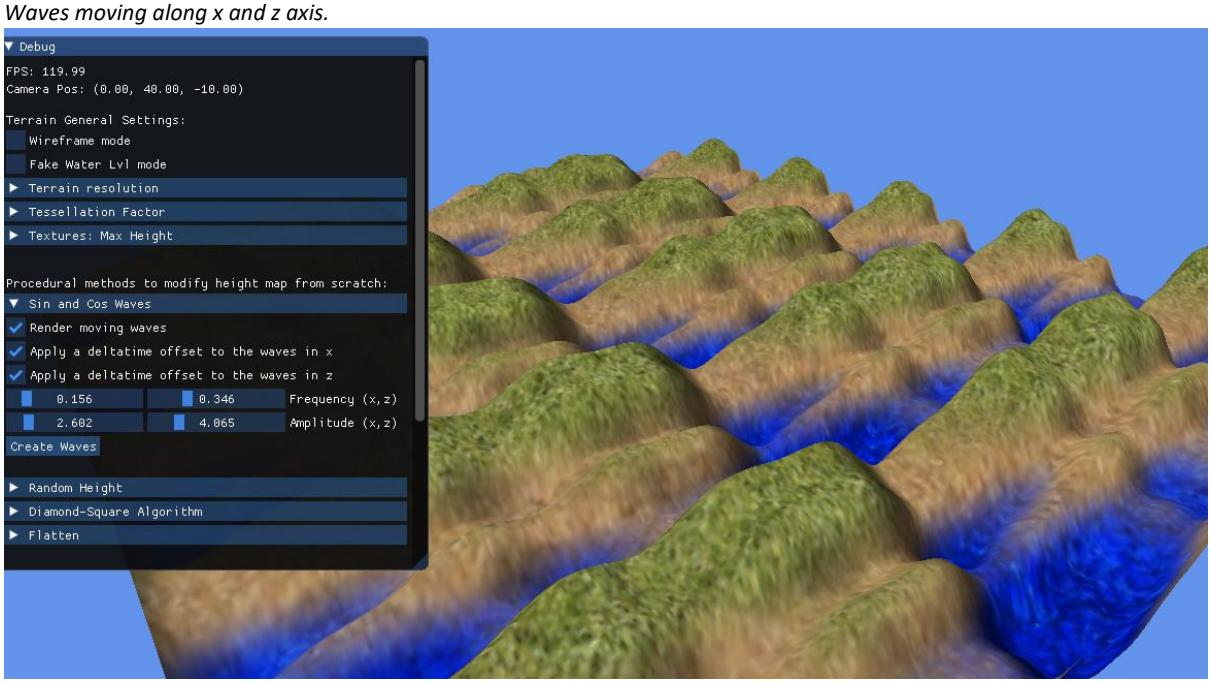


The UI includes the option of applying an offset to the waves in running time making them move. The offset applied is the delta time value. It can be applied to x and/or z axis.

Waves moving along x-axis.



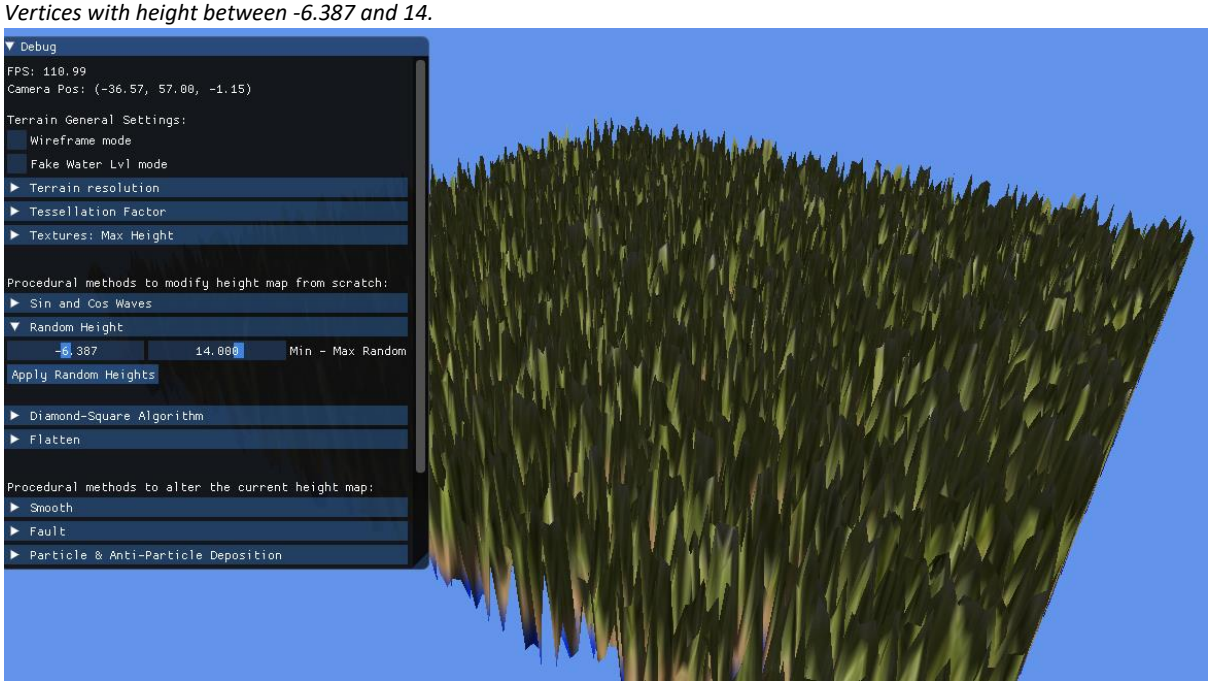
Waves moving along x and z axis.



2.2.2. Random Height

This method assigns a random height to each vertex in the plane. The height is obtained using the C++ <random> library. This is a value between the range (min-max value) passed to the method. The minimum and maximum values can be changed at using the UI. This method creates a very sharp terrain, it is usually used with the method smooth which will be shown later.

Vertices with height between -6.387 and 14.

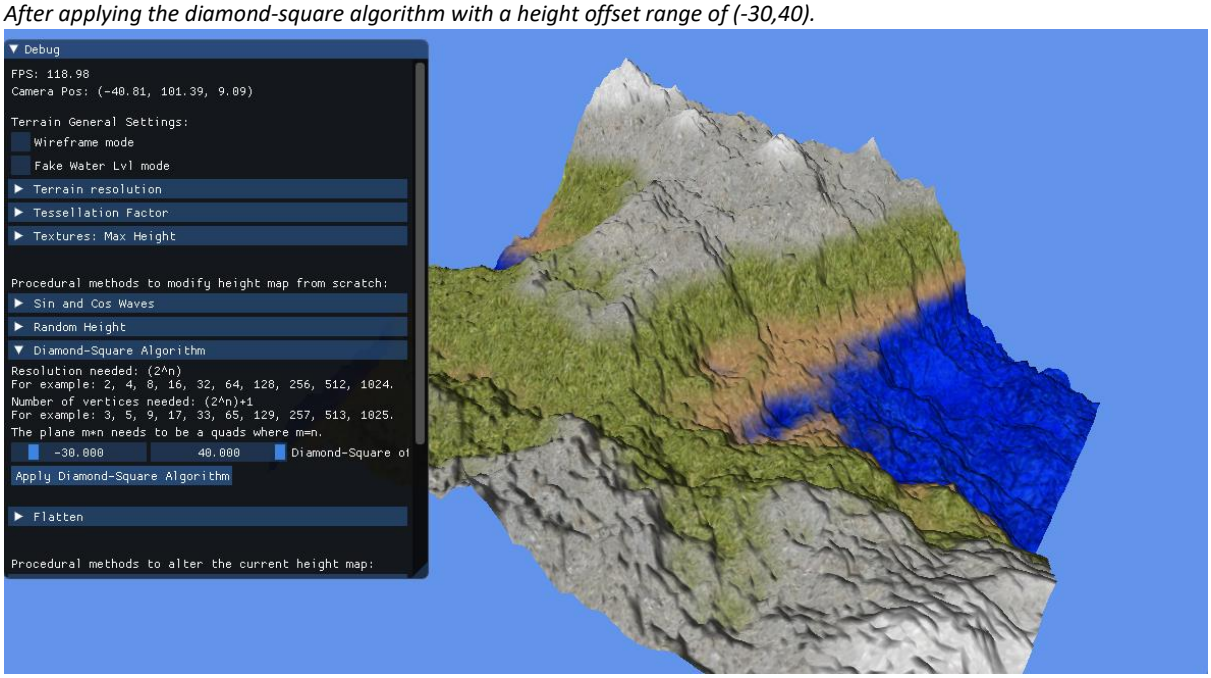


After applied three times the smooth method and increased the height for sand. It has been tried to create a dessert effect with sand dunes.

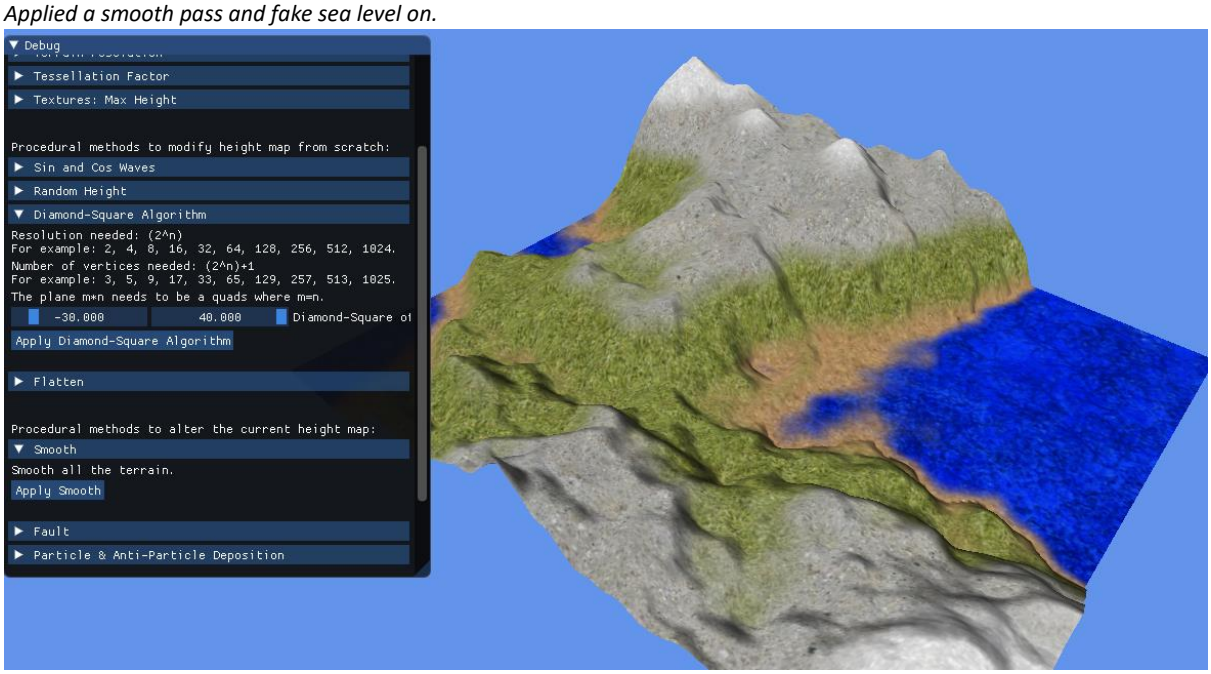
2.2.3. Diamond-Square Algorithm

Examples of terrain made using diamond-square algorithm, these examples use an offset range of (-30, 40) and a terrain with resolution of 128*128. Using the same height offset range can produce thousands of different terrains. If the range is short, the terrain tends to be flattener. On the other hand, if there are large height offset the terrain tends to be with more mountains, more noticeable.

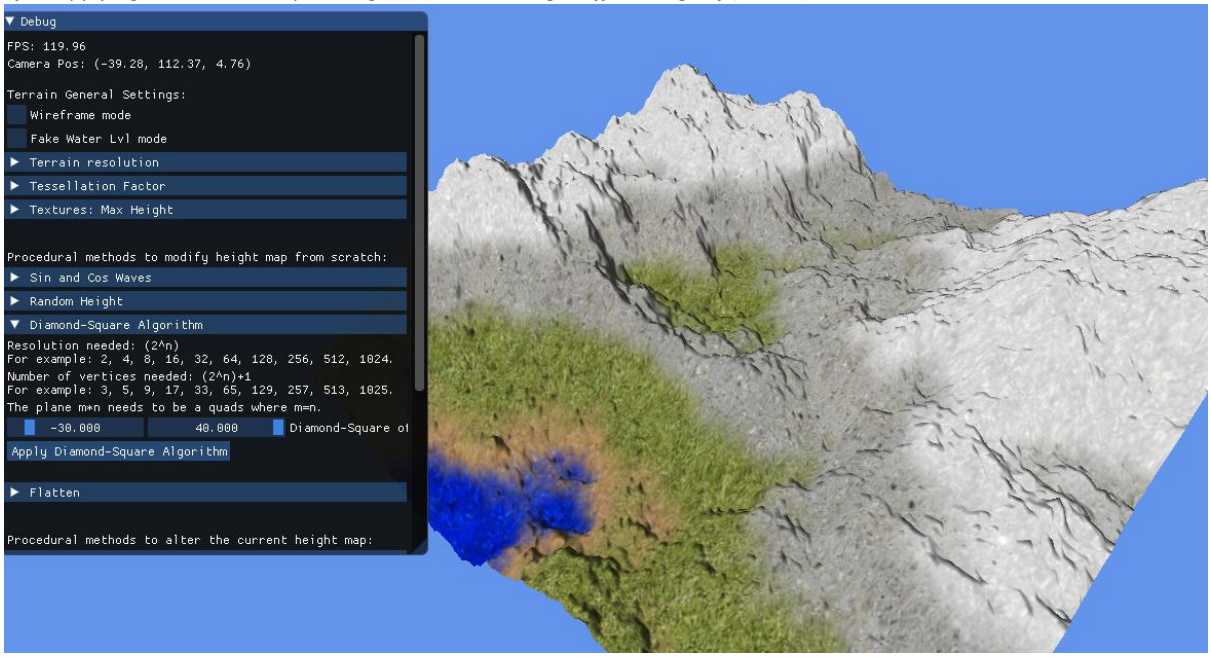
After applying the diamond-square algorithm with a height offset range of (-30,40).



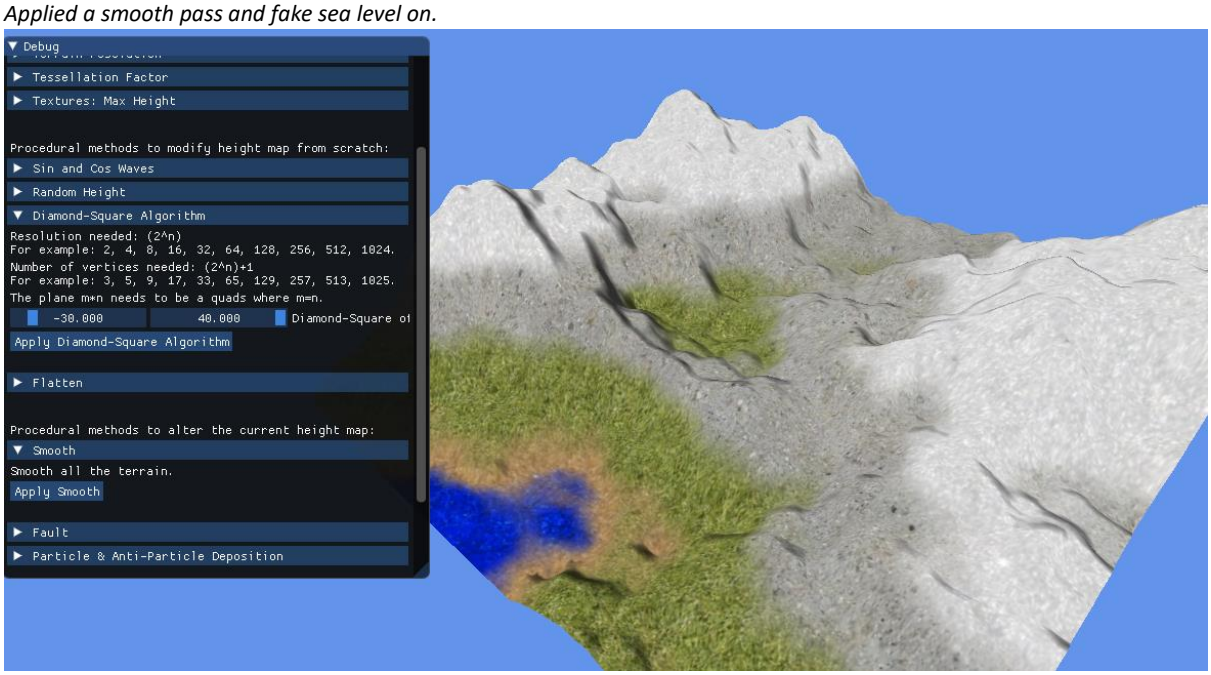
Applied a smooth pass and fake sea level on.



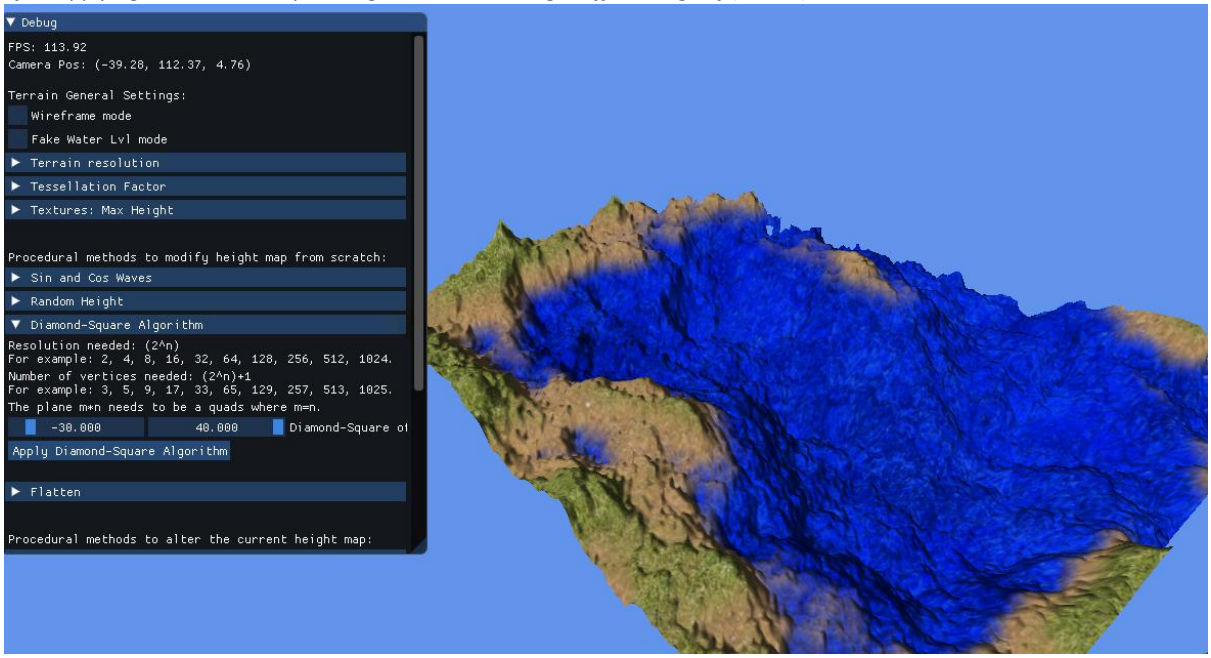
After applying the diamond-square algorithm with a height offset range of (-30,40).



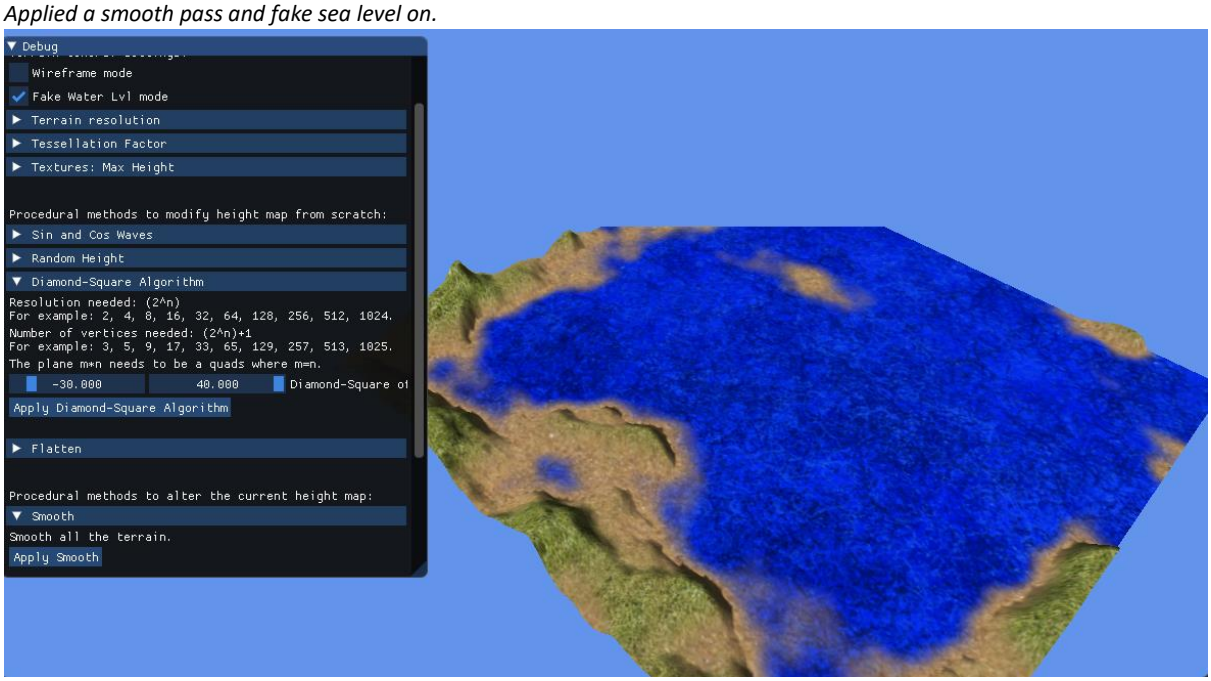
Applied a smooth pass and fake sea level on.



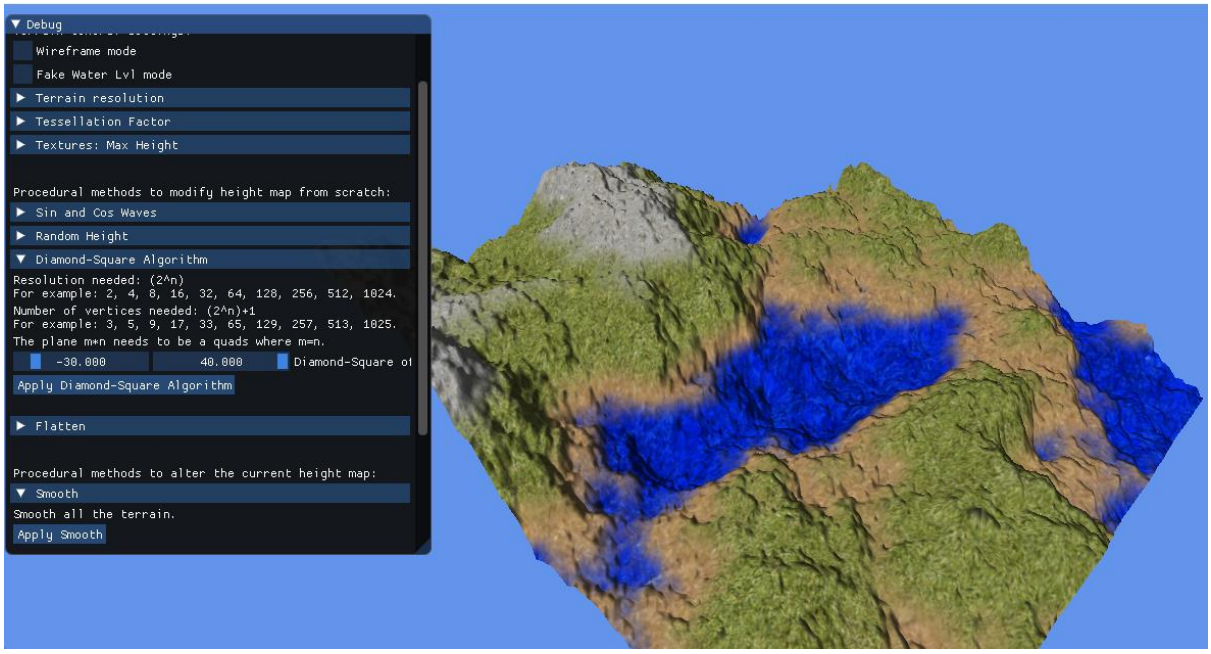
After applying the diamond-square algorithm with a height offset range of (-30,40).



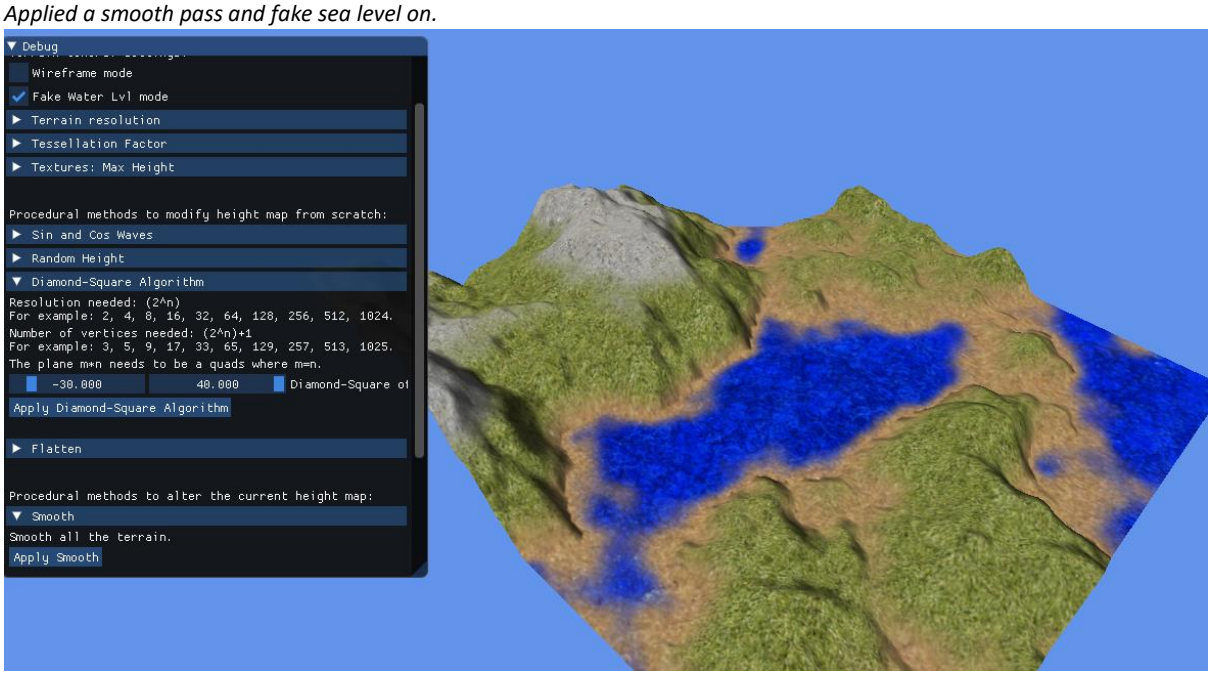
Applied a smooth pass and fake sea level on.



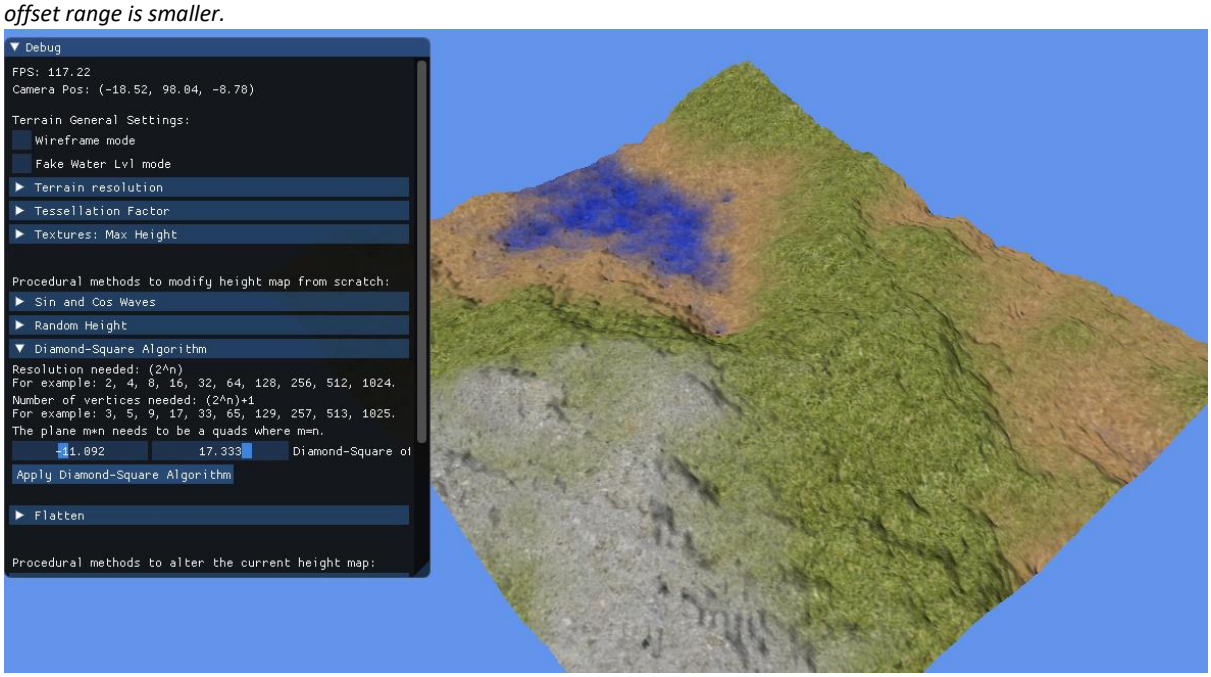
After applying the diamond-square algorithm with a height offset range of (-30,40).



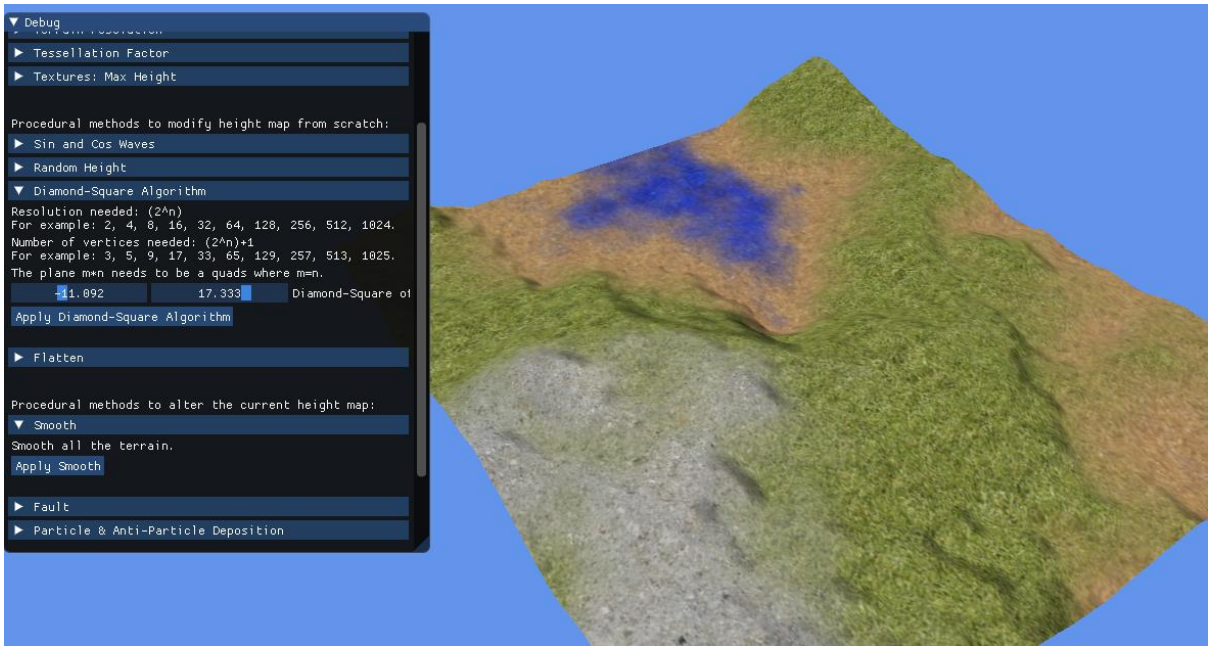
Applied a smooth pass and fake sea level on.



After applying the diamond-square algorithm with a height offset range of (-11.092,17.33) -> flattener terrain as the height offset range is smaller.



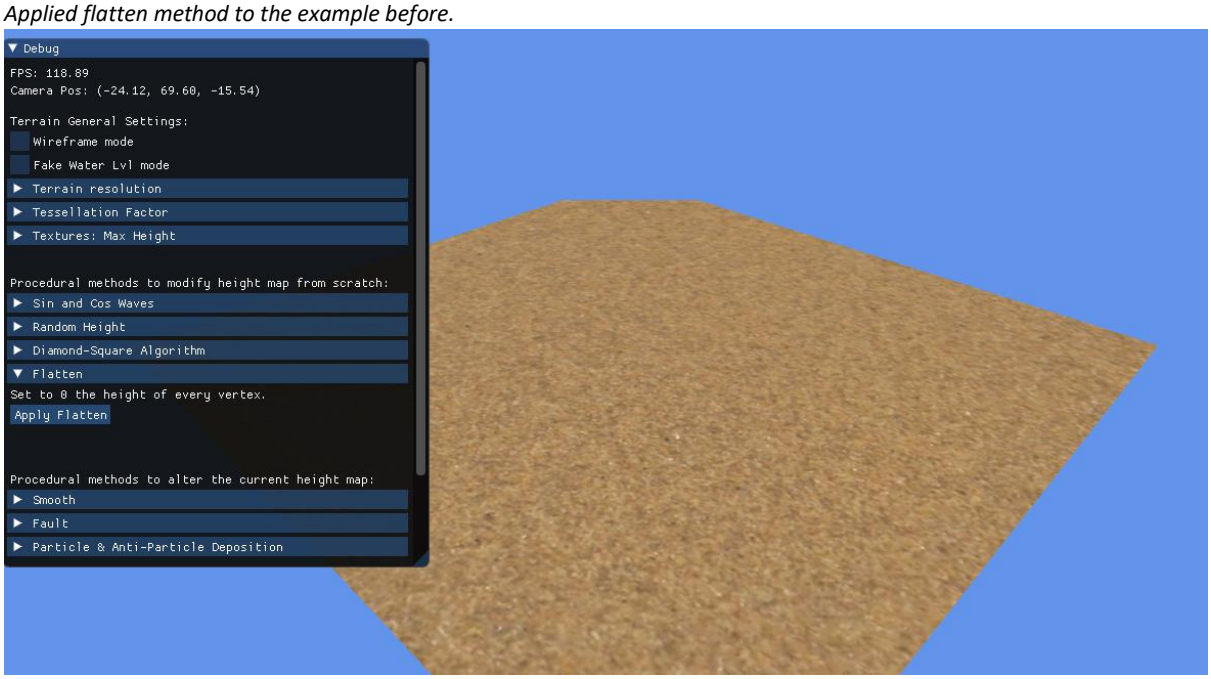
Applied a smooth pass and fake sea level on.



2.2.4. Flatten

It sets to zero every height in the height map collection, flatting the terrain.

Applied flatten method to the example before.



2.3. Procedural methods to adjust the current height map.

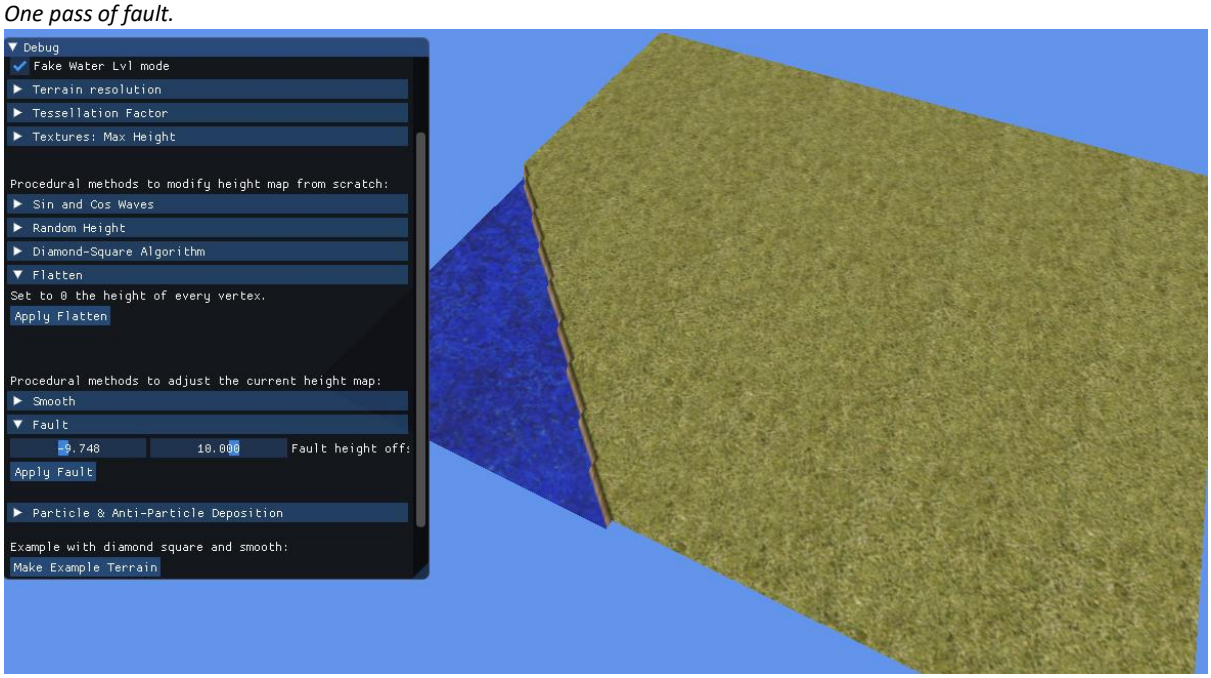
2.3.1. Smooth

As it has been seen in the previous examples the smooth method makes the plane a little bit flat, removing the sharp points. This is done by calculating the height average of nine neighbours (vertices) including the current height and setting that average to that vertex.

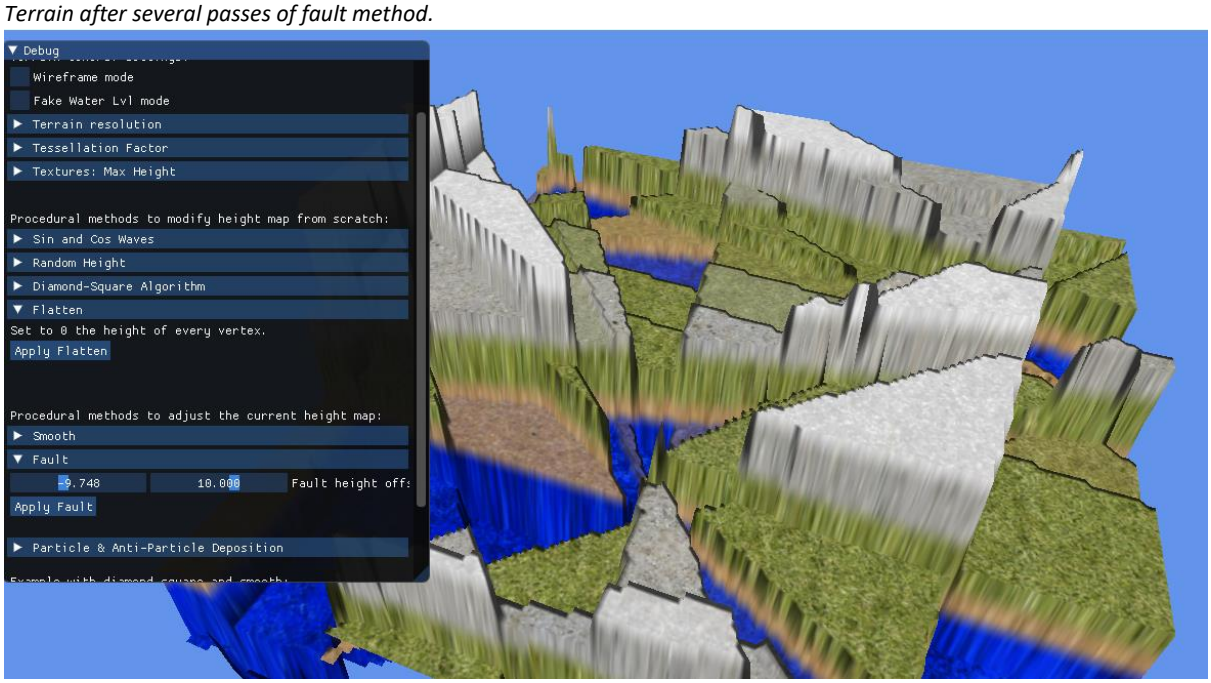
2.3.2. Fault

Fault methods create a line between two random points in the terrain. Every vertex which is on the left of that line is levelled up using a random height in the range passed, the rest of the vertices are levelled down by that random height.

One pass of fault.



Terrain after several passes of fault method.

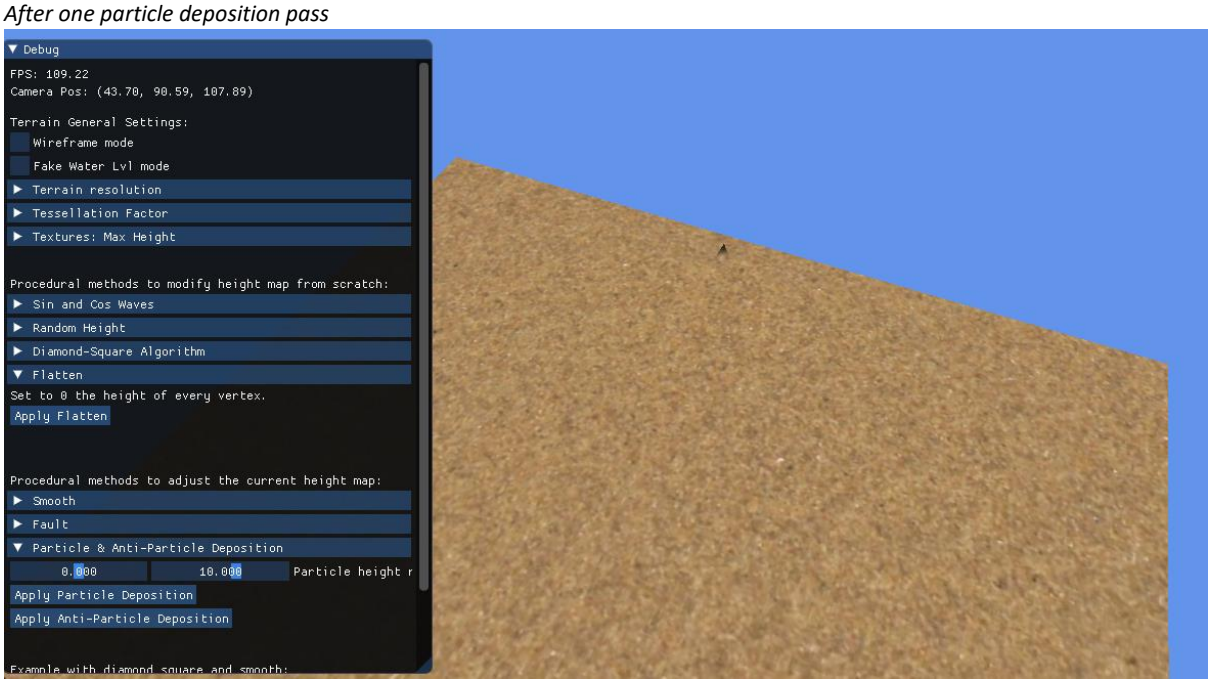


After four smooth passes and the sea fake level on.

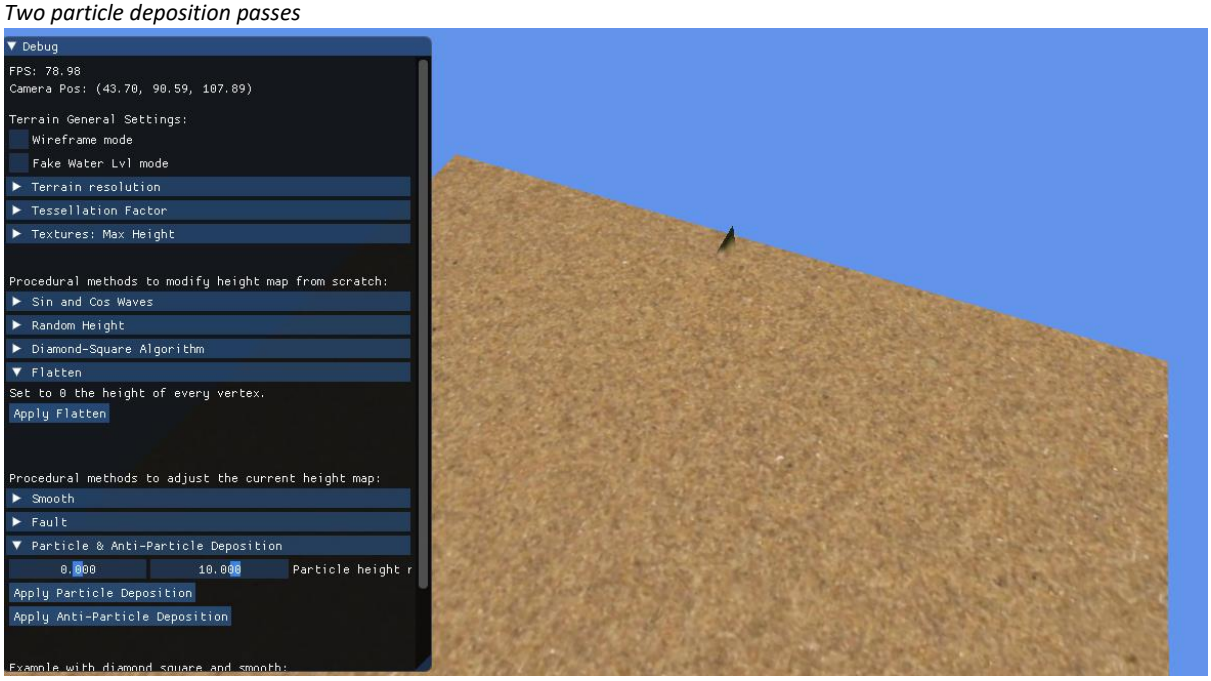
2.3.3. Particle & Anti-Particle Deposition

Particle deposition method add a particle to the terrain where the emitter is localised. This particle emitter is placed randomly in the map when it is initialised. It checks if the eight vertices' neighbours around the current point have lowest height than the current point. If so, then it drops that particle to that lowest neighbour, increasing the height. This height addition will be a random number between the range sent as a parameter. This range can be modified in the UI.
There is an anti-particle option which has the opposite effect. Where the anti-particle is deposited it removes height from that vertex, choosing the neighbour which is higher in this case.
To these deposits can be applied the smooth pass later.

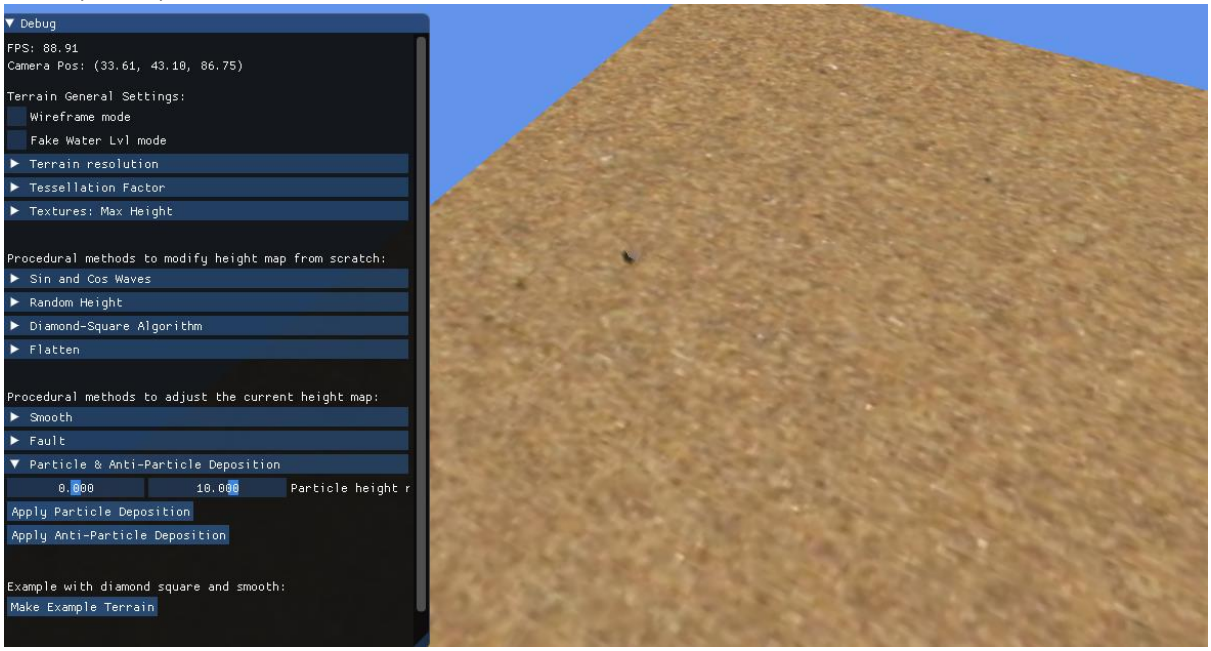
After one particle deposition pass



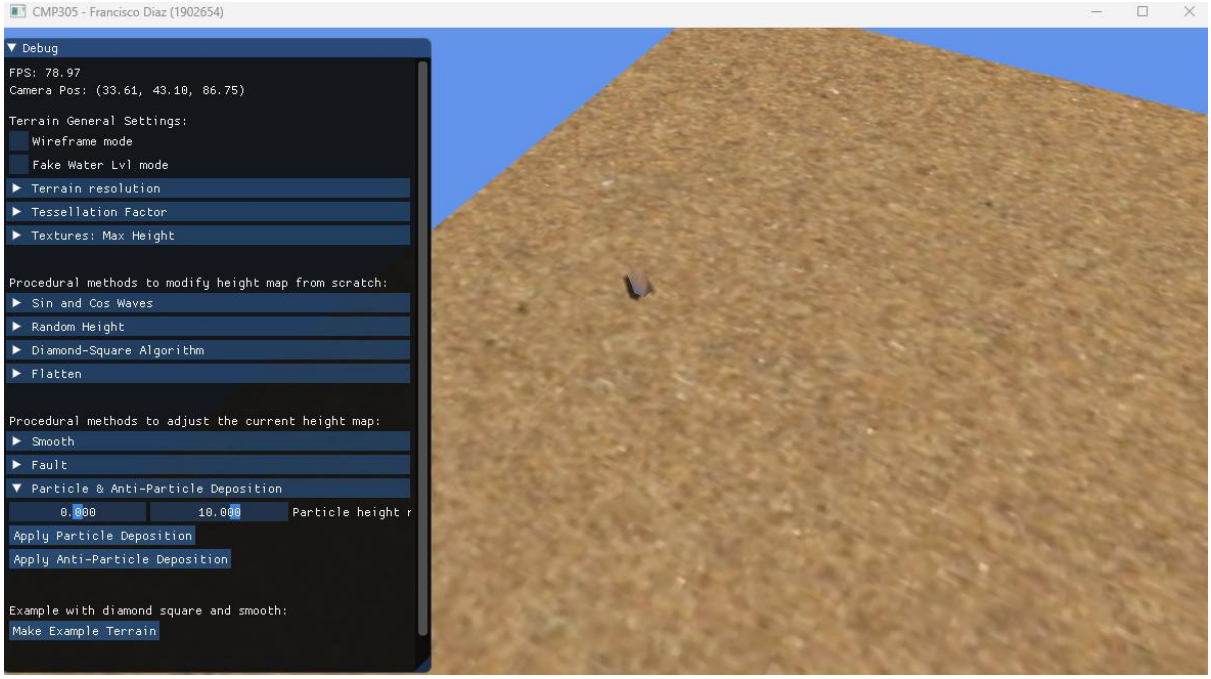
Two particle deposition passes



An anti-particle passes



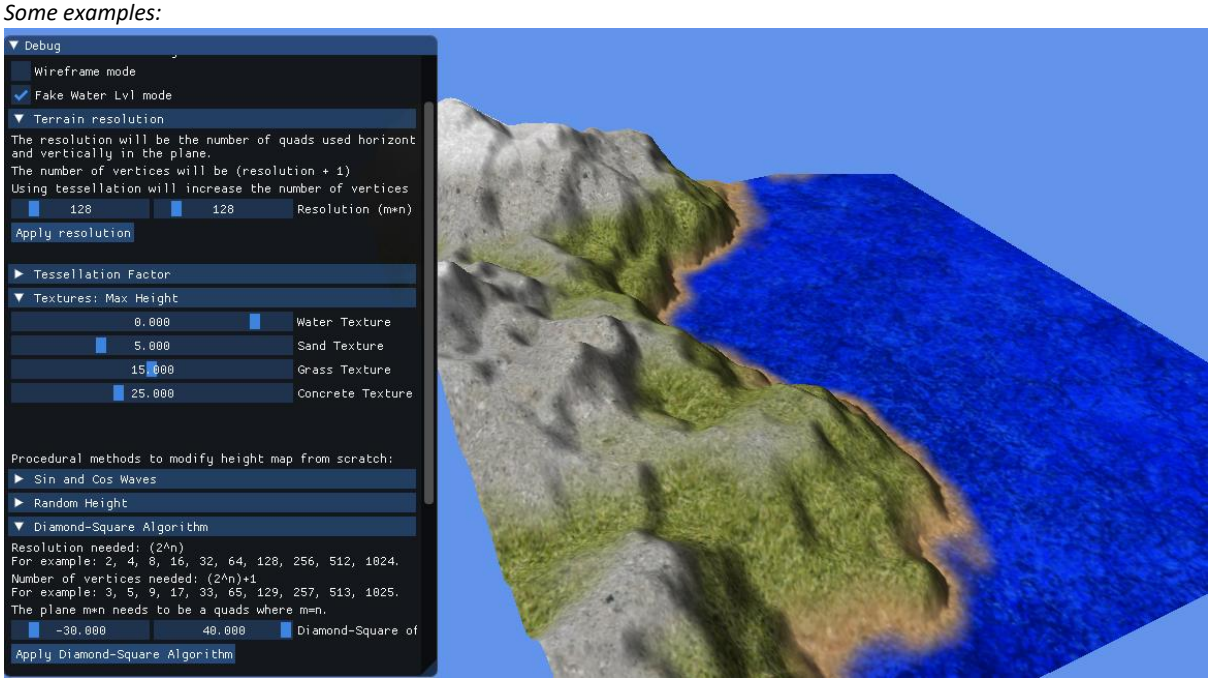
Two antiparticle passes

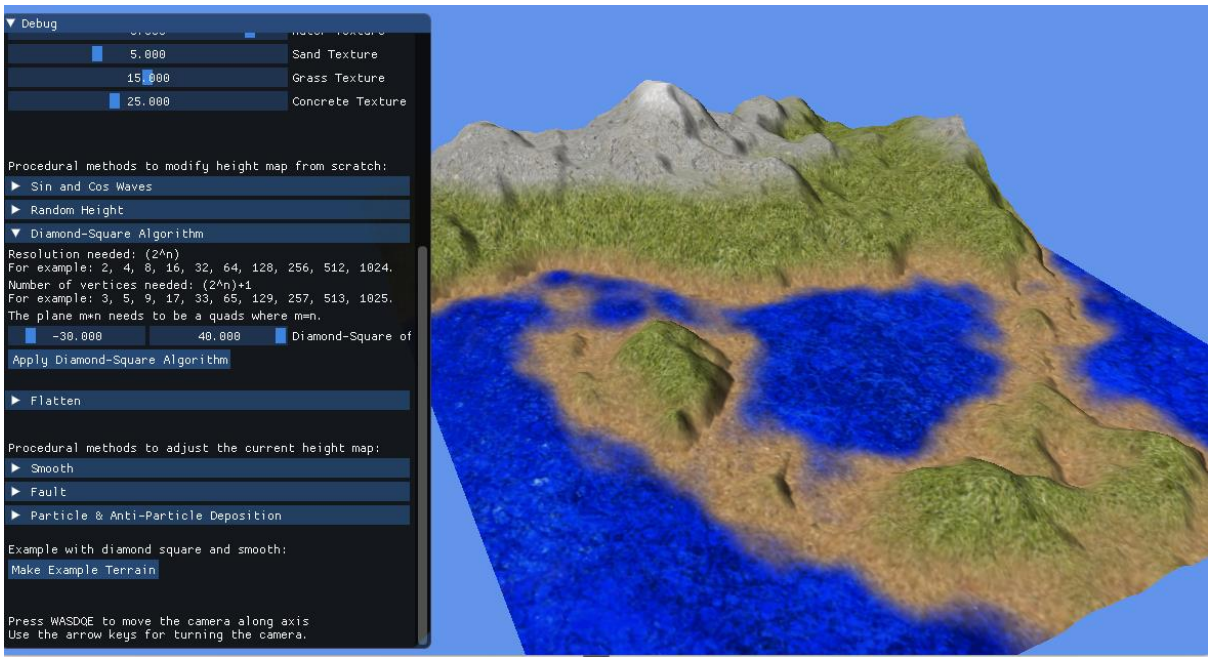


2.3.4. Example Terrain

It makes a terrain using the diamond-square algorithm followed by a smooth pass.
Predefined parameters which can be checked in the UI are used for these two algorithms. The parameters are: The fake water level on, a terrain with resolution 128*128, the texture maximum heights to 0, 5, 15, and 25 for water, sand, grass and concrete respectively, and finally a height offset range for the diamond-square of (-30, 40).
Each time the button is pressed, it will create a different terrain.

Some examples:

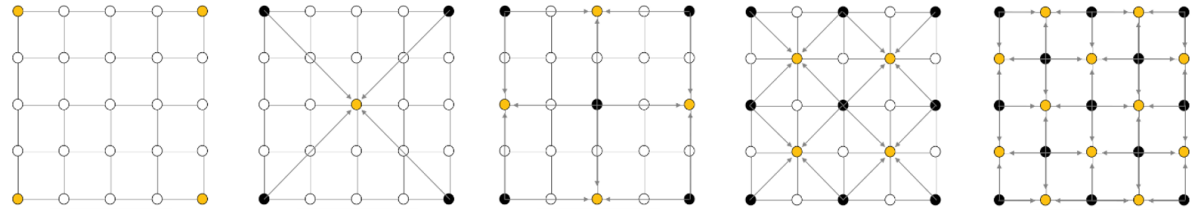




3. In-depth description of the key procedural features

3.1. Diamond-Square Algorithm

*Diamond square image on a 5*5 array*



3.1.1. Requirements

The diamond square algorithm it needs a plane of width and deep of 2^n + 1 vertices or as it is shown in the UI a 2^n of resolution (quads).

For this there is a first filter at the beginning of the algorithm, which makes uses of the functions ceil and log2 to find out if the 2^n + 1 is met, if it is not them the algorithm it is not run:

```
// Check if this algorithm can be applied to this terrain
// The vertices needs to be (2^n)+1 where n>0
// By taking log2 of N and then pass it to floor and ceil if both gives same result then N is power of 2
// as we are checking 2^n+1 then we neeed to substract 1 from the resolution to check this
// Removing the posibility of (2^0)+1 => 1+1 => 2
if ((ceil(log2(resolution.x)) != floor(log2(resolution.x)) || resolution.x == 2) ||
    (ceil(log2(resolution.y)) != floor(log2(resolution.y)) || resolution.y == 2))

{
    return; // exit this function as the terrain resolution is odd
}
```

3.1.2. Pre-Algorithm

First step: Initialise the four corners of the plane.

The top left, top right, and bottom left and bottom right corner heights are initialised to a random height between the range height parameter passed into the function, which can be modified in the UI diamond-square section.

TerrainMesh.cpp -> DiamondSquareAlgorithm function

```
// Initialise corners points [(m*n) localitation in the array heightMap] //
int topLeft, topRight, bottomLeft, bottomRight;
int center;

// set m and n values
int m_start = 0;
int m_end = resolution.x;
int n_start = 0;
int n_end = resolution.y;

// get the height map indices of the corners in the heightmap array
topLeft = GetHeightMapIndex(m_start, n_start);
topRight = GetHeightMapIndex(m_start, n_end);
bottomLeft = GetHeightMapIndex(m_end, n_start);
bottomRight = GetHeightMapIndex(m_end, n_end);

// set the height offset for the initial corner points
Range tmpHeightOffsetRange = heightOffsetRange;

// Asign a random height to each corner
heightMap[topLeft] = Utils::GetRandom(tmpHeightOffsetRange);
heightMap[topRight] = Utils::GetRandom(tmpHeightOffsetRange);
heightMap[bottomLeft] = Utils::GetRandom(tmpHeightOffsetRange);
heightMap[bottomRight] = Utils::GetRandom(tmpHeightOffsetRange);
```

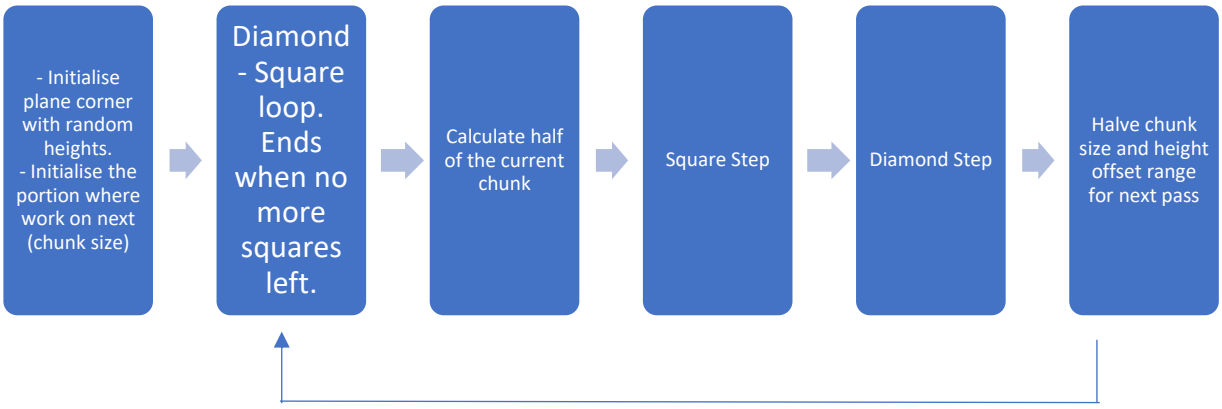
Second step: Initialise the proportion of the plane where the following diamond-square passes are going to be run. At the beginning this is initialise to the whole plane.

TerrainMesh.cpp -> DiamondSquareAlgorithm function

```
XMINT2 chunkSize =  XMINT2(resolution.x, resolution.y); // portion we are working on
```

The third step is where the actual diamond-square is going to take place. It is inside a while loop which stops when the portion where to work on is lower than one, in other words, it stops where there are not more squares left.

The first thing which needs to be done prior the call to square and diamond steps is calculating the half of the chunk size. This parameter will be the half of chunk size and it will be sent to square and diamond square to be used there. After that the square steps is called followed by the diamond step. Once the steps are done the portion where we are working on needs to be halved, same as the height range where the random height offset are obtained inside the square-diamond steps.



TerrainMesh.cpp -> DiamondSquareAlgorithm function

```
while (chunkSize.x > 1 && chunkSize.y > 1)
{
    // get the half of the portion we are working on
    XMINT2 half = XMINT2(chunkSize.x / 2, chunkSize.y / 2);

    // Apply Square Step //
    SquareStep(m_start, m_end, n_start, n_end, chunkSize, tmpHeightOffsetRange, half);

    // Apply Diamond Step //
    DiamondStep(m_start, m_end, n_start, n_end, chunkSize, tmpHeightOffsetRange, half);

    // halve the portion of the plane where to work next
    chunkSize = XMINT2(chunkSize.x/2.0f, chunkSize.y/2.0f);

    // halve the height offset
    tmpHeightOffsetRange.min /= 2.0f;
    tmpHeightOffsetRange.max /= 2.0f;
}
```

3.1.3. Square Step:

It is done inside a double for loop for obtaining the corners and the centre of the square we are working. The corners are obtained using the chunk size and the centre is obtained using the half of the chunk size. After obtaining the points, it is needed to calculate the height average of these four corners (top left, top right, bottom left and bottom right). This height average plus a random height offset will be the new height of the centre of the square.

For example, if the corners have a height of 5, 2, 8, 1 then their average will be 4, plus a random height of 2 the new height of the centre of the square will be 6.

TerrainMesh.cpp -> SquareStep function

```
void TerrainMesh::SquareStep(int& m_start, int& m_end, int& n_start, int& n_end, XMINT2& chunkSize, Range& tmpHeightOffsetRange, XMINT2& half)
{
    for (int m = m_start; m < m_end; m += chunkSize.x)
    {
        for (int n = n_start; n < n_end; n += chunkSize.y)
        {
            // get the height map indices of the corners of the square in the heightmap array
            int topLeft = GetHeightMapIndex(m, n);
            int topRight = GetHeightMapIndex(m, n + chunkSize.y);
            int bottomLeft = GetHeightMapIndex(m + chunkSize.x, n);
            int bottomRight = GetHeightMapIndex(m + chunkSize.x, n + chunkSize.y);

            // calculate the average of the four corners
            float cornersAvg = (heightMap[topLeft] + heightMap[topRight] + heightMap[bottomLeft] + heightMap[bottomRight]) / 4.0f;

            // square centre point
            int centre = GetHeightMapIndex(m + half.x, n + half.y);

            // set the height to the centre point
            float randomHeightOffset = Utils::GetRandom(tmpHeightOffsetRange);
            heightMap[centre] = cornersAvg + randomHeightOffset;
        }
    }
}
```

3.1.4. Diamond Step:

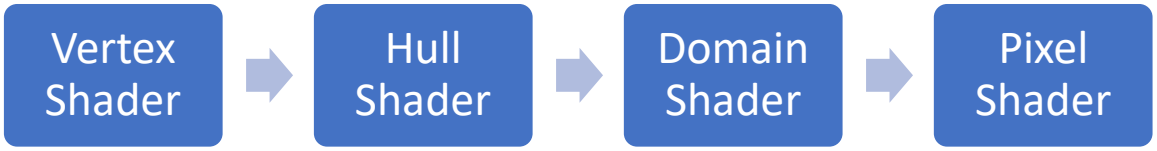
It is also done inside a double for loop, however in this case we want to obtain the corners of a diamond (top, left, right and bottom) and its centre point. Sometimes there will be a missing point in the diamond so instead of four corners will have three, this happens when the middle point of the diamond is in the limit of the chunk size, for this reason it is necessary to check if each corner point is in bounds, if so then count it for the average. After calculating the height average of the corners set this value plus a random height to the centre point of the diamond.

TerrainMesh.cpp -> DiamondStep function

```
void TerrainMesh::DiamondStep(int& m_start, int& m_end, int& n_start, int& n_end, XMINT2& chunkSize, Range& tmpHeightOffsetRange, XMINT2& half)
{
    for (int m = m_start; m <= m_end; m += half.x)
    {
        for (int n = (m + half.x) % chunkSize.y; n <= n_end; n += chunkSize.y)
        {
            int count = 0;
            float cornersSum = 0;

            // top corner
            if (InBounds(m - half.x, n))
            {
                cornersSum += heightMap[GetHeightMapIndex(m - half.x, n)];
                count++;
            }
            // left corner
            if (InBounds(m, n - half.y))
            {
                cornersSum += heightMap[GetHeightMapIndex(m, n - half.y)];
                count++;
            }
            // right corner
            if (InBounds(m, n + half.y))
            {
                cornersSum += heightMap[GetHeightMapIndex(m, n + half.y)];
                count++;
            }
            // bottom corner
            if (InBounds(m + half.x, n))
            {
                cornersSum += heightMap[GetHeightMapIndex(m + half.y, n)];
                count++;
            }

            // calculate average
            float cornersAvg = (float)cornersSum / (float)count;

            // diamond center point
            int center = GetHeightMapIndex(m, n);

            // set the value to the center point of the diamond
            float random = Utils::GetRandom(tmpHeightOffsetRange);
            heightMap[center] = cornersAvg + random;
        }
    }
}
```

3.2. Tessellation & Multiple textures

Brief of the process in the shaders used for this project:



It has been implemented a tessellation of 12 controls points (a quad and its neighbours, a total of five quads). For this has been redone the vertices generation for creating the plane in ccw.
Additionally, it has been added a hull and a domain shader.
The vertex shader is now only for passing the vertex position, texture coordinates and normal to the hull shader.
In the hull shader the tessellation factor, the distance from the camera, etc are calculated and the points are sent to the domain shader.
At this point the new vertex position, texture coordinate and normal position are calculated and sent to the pixel shader. These new parameters are calculated using bilinear interpolation (lerp).

tessellation_ds.hlsl, main function

```
// Determine the position of the new vertex (Bilinear Interpolation)
// Deal with y coords
// Linearly interpolate(lerp) between 0 and 1
float3 v1 = lerp(patch[0].position, patch[1].position, uvCoord.y);
// Linearly interpolate(lerp) between 3 and 2
float3 v2 = lerp(patch[3].position, patch[2].position, uvCoord.y);
// The final vertex position will be the lerp result between v1, v2 based on our x-coords
vertexPosition = lerp(v1, v2, uvCoord.x);

float2 t1 = lerp(patch[0].tex, patch[1].tex, uvCoord.y);
// Linearly interpolate(lerp) between 3 and 2
float2 t2 = lerp(patch[3].tex, patch[2].tex, uvCoord.y);
// The final vertex position will be the lerp result between v1, v2 based on our x-coords
texVal = lerp(t1, t2, uvCoord.x);

float3 n1 = lerp(patch[0].normal, patch[1].normal, uvCoord.y);
// Linearly interpolate(lerp) between 3 and 2
float3 n2 = lerp(patch[3].normal, patch[2].normal, uvCoord.y);
// The final vertex position will be the lerp result between v1, v2 based on our x-coords
normalVal= lerp(n1, n2, uvCoord.x);
```

Additionally, it is here where if the fake water level toggle is activated the vertices below zero are set to zero before sending applying the world, view, and projection matrices.

tessellation_ds.hlsl, main function

```
// fake the water level by setting the height to 0 where it is lower
if (fakeWaterLevel == 1.0f && vertexPosition.y < 0.0f)
{
    vertexPosition.y = 0.0f;
    normalVal = float3(0.0f, 1.0f, 0.0f);
}
```

In the pixel shader there has been added ambient light to the light calculation previously provided and instead of using one texture for the whole terrain now it uses four textures. The texture or textures which are used in a pixel depend on the height of its vertex. There is transition between two the textures, they are blended. The weight of each texture in this blend is obtained using the function smoothstep from DirectX11.

Tessellation_ps.hlsl calculateTextureColour function. Example of transtition between texture0 (water) and texture1(sand).

```
// Only texture 0
if (height < maxHeightTex0 - transitionDistance)
{
    textureColour = textures[0].Sample(sampler0, uv);
}
// smooth transition from texture 0 to texture 1
else if (height <= maxHeightTex0)
{
    float minGradingHeight1 = maxHeightTex0 - transitionDistance; // height where the transition to texture 1 starts
    float maxGradingHeight1 = maxHeightTex0; // height where the transition to texture 1 ends

    // calculate weight of each texture
    float weight1 = smoothstep(minGradingHeight1, maxGradingHeight1, height);
    float weight0 = 1.0f - weight1; // calculate how much weight is for texture 0, being one the max value and knwoing the weight of texture 1

    textureColour = blend(textures[0].Sample(sampler0, uv), textures[1].Sample(sampler0, uv), weight0, weight1);
}
// only texture 1
```

4. Discussion of the code architecture and organisation

App1 class contains the terrain mesh and all the data which is used in the GUI to be sent to the procedural methods in the terrain mesh. It has been added a function update which is called before the render function to update any parameter in the program such as camera and the waves in movement if they are activated. This will follow the common game flow. The GUI function has all the options to modify the parameters for the procedural methods or to run/stop them at any time.

TerrainMesh class creates the mesh buffer and contains the height map which is used in the procedural methods. Each time the height map is modified the buffer must be updated with the new height, normal, etc. It contains all the procedural methods which modify the height map.

Emitter class contains the Particle structure and Emitter Behaviour which for this project has been set only to default. This class has a function which is called from the terrain mesh particle/anti-particle deposition, which will drop a particle in the position of the emitter (later in the terrain mesh function is done if the particle falls exactly there or in any of the neighbours as previously in this documentation was said).

Utils class contains a range structure for heights and two static functions which return a random float. These are used to get a height from a height range in the procedural functions.

Tessellation Shaders contains all the structure necessary for the buffer. It is here where the shaders (vertex, hull, domain and pixel) are initialised with their parameters such as world, view and projection matrices, tessellation factor, camera position, fake water level toggle, textures, diffuse, ambient and direction light which are sent to the shaders in each render pass.

Shaders:
- tessellation_vs.hlsl: Vertex Shader, only for sending vertices, texture coordinates and normals to the hull shader.
- tessellation_hs.hlsl: Hull Shader, where the tessellation is factor and camera position is done patching the points to the quad.
- tessellation_ds.hlsl: Domain Shader, where the new vertices are calculated and applied against to the world, view and projection matrices.
- tessellation_ps.hlsl: Pixel Shader where is calculated what texture to use or what percentage of two textures plus the light calculation.

5. Critical appraisal of the solution to the brief

The solution brief has had as a goal to generate terrains as realistic as possible using different techniques of procedural methods. As a main feature the use of the diamond-square algorithm, multiple textures, and tessellation with 12 control points. The most efficient technique is the diamond-square algorithm followed by a smooth pass. The rest of the techniques even though they can be used in conjunction (particle deposition, fault, sine/cosine waves etc) they do not produce a realistic terrain.
A thing might take into consideration to improve is the addition of smoothing normals in the terrain mesh as it was done previously before applying the tessellation.
Additionally, It might be interested to separate the procedural methods from the terrain mesh class so they could be used in other meshes. However as in this project are only used in this mesh it has been decided to keep it there.

6. Reflection on what has been learned

It has been learned how to use different techniques such as multiple textures, tessellation, diamond-square, smooth, faking water level to achieve a goal, generate a terrain as much realistic as possible. Additionally, it has been improved how to apply other module material into this project (tessellation) in order to obtain a better procedural terrain as both modules share the use of directx11. It is interesting how much time can save generating terrains using procedural methods. This experience with procedural methods can benefit me in the future as this is hugely used in the gaming industry. It is of great help not only for getting a job where they use procedural methods if not also for personal projects or game jams where the time to create a game is short. It has to be taking into account that procedural methods not only generate terrains, can be generate thousands of different things such as animation, plants, textures, etc thus it is widely used in many areas.
To summarise, it has been acquired a great fundament in the game industry and very useful in a game developer career.

7. References

Klayton Kowalski (2021). Game Development Tutorial | Diamond Square and Procedural Map Generation. Youtube. Available at: https://www.youtube.com/watch?v=4GuAV1PnurU (Last Accessed: 10/04/2022)

CMP301 (2021-2022). Graphics Programming with Shaders. Tessellation with 3 control points (triangle) provided in the lab 08.

DelftStack (2020). Generate a Random Float Number in C++. Available at: https://www.delftstack.com/howto/cpp/how-to-generate-random-float-number-in-cpp/ (Last Accessed: 09/12/2021)