ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Nahodil**	Jméno: **Petr**	Osobní číslo: **495574**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačové grafiky a interakce**		
Studijní program:	**Otevřená informatika**		
Specializace:	**Počítačové hry a grafika**		

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Procedurální generování scény pro vlakový simulátor

Název bakalářské práce anglicky:

Procedural Scene Generation for Train Simulator

Pokyny pro vypracování:

Zmapujte techniky modelování 3D scén vhodné pro vytváření prostředí pro simulátor jízdy vlakem. Soustřeďte se na využití procedurálních technik pro vytvoření okolní krajiny a instancí vegetace (stromy, keře, tráva). V herním enginu Unity implementujte procedurální generátor terénu, který bude schopen generovat okolní krajinu dynamicky během průjezdu vlaku definovanou trajektorií. Trajektorie průjezdu vlaku bude specifikována ve formě animační křivky a generování terénu jí musí být přizpůsobeno. Optimalizujte implementaci tak, aby generování a zobrazování běželo v reálném čase a efektivně využívalo možností GPU. Vyhodnoťte rychlost generování a zobrazování modelu na různě výkonném hardwaru.

Seznam doporučené literatury:

[1] Petr Brachaczek. Modely 3D scén pro jízdní simulátor. Bakalářská práce, ČVUT FEL 2017.
[2] Jana Kejvalová. Procedurální generování 3D modelu dle mapových podkladů. Diplomová práce, ČVUT FEL 2019.
[3] Alena Mikushina, Creation of modular 3D assets for videogames. Bakalářská práce, ČVUT FEL 2020.
[4] Jan Kutálek. Procedurální generování prostředí pro videohry. Bakalářská práce, ČVUT FEL 2021.
[5] Smelik, Ruben M., et al. 'A survey on procedural modelling for virtual worlds.' Computer Graphics Forum. Vol. 33. No. 6. 2014.
[6] Noor Shaker, Julian Togelius, Mark J. Nelson. Procedural Content Generation in Games. Springer International Publishing. 2016.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Jiří Bittner, Ph.D. **Katedra počítačové grafiky a interakce**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **17.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.

Datum převzetí zadání Podpis studenta

Bachelor Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Creating 3D scenes for a train simulator

Petr Nahodil

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
May 2023

Acknowledgements

I would like to express my gratitude to this project's supervisor doc. Ing. Jiří Bittner, Ph.D. for his guidance and support in completing this project.

Declaration

I declare that I have written this bachelor thesis independently and quoted all used sources.

Abstract

The scope of this bachelor thesis is procedural generation of terrain at run-time, the placement of vegetation in this terrain, and its adaptability to an animation curve defining a train track path. It should therefore be relatively simple to combine this work with a train simulator. The main focus was dynamic and fluid real-time generation of the infinite world around the camera, and on minimizing the application stuttering during the generation itself, i.e. on generating terrain asynchronously.

Keywords: Terrain Generation, Procedural Terrain, Height Map, LOD, Tessellation, Compute Shaders

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Praha 2, Karlovo náměstí 13, E-421

Abstrakt

Náplní této bakalářské práce je procedurální generování terénu za běhu aplikace, rozmisťování vegetace v tomto terénu, a jeho přizpůsobivost animační křivce definující trasu kolejí. Mělo by tedy být relativně jednoduché tuto práci spojit se simulátorem vlaku. Hlavní důraz byl kladen na dynamičnost a plynulost generování terénu nekonečného světa okolo kamery v reálném čase, a na minimalizaci zasekávání aplikace při samotném generování, tedy na jeho asynchronicitu.

Klíčová slova: Generování Terénu, Procedurální Terén, Výšková Mapa, LOD, Teselace, Compute Shadery

Překlad názvu: Vytváření 3D scén pro vlakový simulátor

Contents

Figures

Tables

Chapter 1
Introduction

This bachelor project's goal was creating a Unity application which can procedurally generate terrain for a train simulator, the train simulator itself was a different project assigned to another student.

The key objectives were to implement procedurally generated terrain, to distribute vegetation objects (such as trees and bushes) in the generated world, to generate the world dynamically around a camera, to make the terrain adapt to the trajectory of the train tracks defined by an animation curve, to generate the world in real time without undue loading or freezing of the application, to utilize the capabilities of the GPU for generating the terrain, and lastly to compare performance of the final product on hardware of varying performance.

I've accepted this assignment because procedural generation is something I've been curious about for a long time. Just like most people my age, I grew up playing video games. In video games, procedural generation is seen very often, it can be the game's defining feature to generate entire worlds from scratch (Minecraft, Terraria, Elite:Dangerous, No Man's Sky..), or it can be used in slightly more subtle ways, such as connecting predefined corridors and assets to create sprawling dungeons and place both enemies and rewards within them (Darkest Dungeon, Crypt of the NecroDancer, Spelunky..)

Having been curious about the concept of procedural generation I did have a vague idea of how it worked - I knew noise functions were used and that splitting the world into chunks was a good idea, but not much further than that. After the project I'm much more confident in this area and I'll be able to make better choices when dealing with it in the future.

Chapter 2
Theoretical Background

2.1 Procedural Generation

3D world applications that do not utilize run-time procedural generation usually use a world of static size and appearance created by 3D artists (though they might use procedural generation to create these worlds).

Run-Time dynamic Procedural generation in context of terrain [16] is a process in which just a small part of a potentially infinite world is generated seamlessly in background. This world is purely defined by parameters which influence the logic by which it is generated, not by shaping it directly.

While this process is based on pseudo-random noise functions, it does have the property that for the same input (world coordinates) it will generate the same output (terrain height). This is useful for making an illusion of a very large world. When a user moves far away from a part of the world (thus it is removed from memory) and later comes back to it, it is recalculated with the same output values. So from the user's perspective it is indistinguishable from there being one large world at all times.

2.2 Terrain Chunk

One of the most common concepts when dealing with procedurally generated worlds is a 'chunk'. A chunk is a part of the terrain of predetermined size, it is common to have chunks with static width and depth and infinite height. By splitting the world into these chunks, it becomes possible to implement terrain generation for a single chunk and then connect them by placing them in the correct positions in the 3D scene.

With this approach, having a configurable chunk render distance (i.e. the maximum distance from the camera in which chunks appear) becomes rather simple and it translates to the users of the application being able to set their preferred ratio between performance and render distance.

In applications which support changing the terrain, such as games which include mining, having the world split into chunks is also quite handy because it makes it possible to save the generated world (and any user-made changes to it) in a file per each chunk for example.

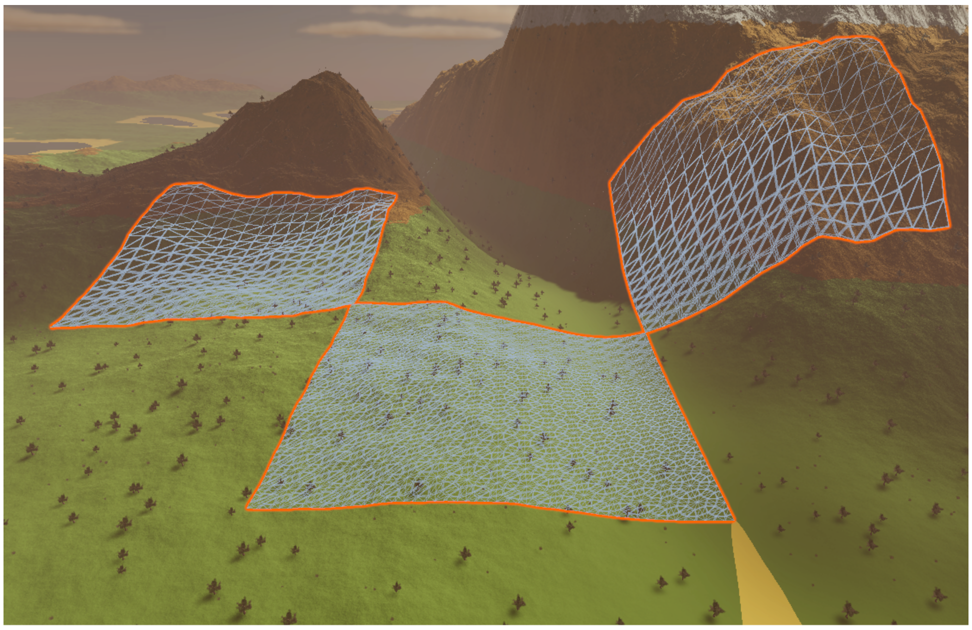


Figure 2.1: Three outlined chunks in the Unity editor

2.3 Mesh vs Height-map Generation

Mesh based terrain generation consists of creating the geometric representation of terrain directly. A mesh most often consists of an array of vertices which are 3D points consisting of three 32-bit floating point coordinates, and an array of (16 or 32 bit) integers which are used to index into the vertices array. Then, every three indices represent a single triangle of geometry. This approach is (in vast majority of meshes) more memory efficient than duplicating vertex data any time two or more triangles need to share a vertex.

A height-map [15] is a gray-scale 2D texture which, instead of representing color at any given point, represents geometry displacement. In the context of terrain, a height-map represents displacement in whichever axis is the "up/down" axis (Y in Unity).

Mesh generation is less memory efficient than height-mapping because each point in space has to be represented by three 32-bit floating point values and one or more (16 or 32 bit) integer index values. Height-mapping a point in space, on the other hand, requires only one 16 or 32 bit value.

An advantage of mesh generation is that it can (more easily) represent terrain with caves for example. A 2D height-map, being a 2D array of height values from top-down perspective of the terrain, can't represent more than one height value per point on the imaginary map. With mesh generation the chunk of terrain can have many overhanging ledges, caves etc.

In this bachelor thesis I first implemented mesh generation before I switched to height-maps and tessellation.

2.4 Level of Detail

Level of Detail [13], commonly referred to as LOD, is used to alleviate the GPU from having to render detailed geometry in situations when it is indistinguishable from rendering a version of that geometry with reduced **level of detail**, at the cost of some CPU resources.

LODs are typically numbered in such a way that LOD0 (LOD zero) is/has the highest quality geometry and LOD1, LOD2 and so on keep decreasing in quality. In other words, higher LOD number should mean lower quality geometry.

There are a few ways to choose which level of detail should be rendered in a given frame. The simplest is to choose an LOD based on distance to camera, while more complicated solutions (like what Unity uses as a built-in LOD solution) try to estimate the amount of pixels the object is going to take up on the screen.

Naturally, the further away from the camera the object is, and the less pixels it is estimated to take up, the lower resolution LOD should be used.

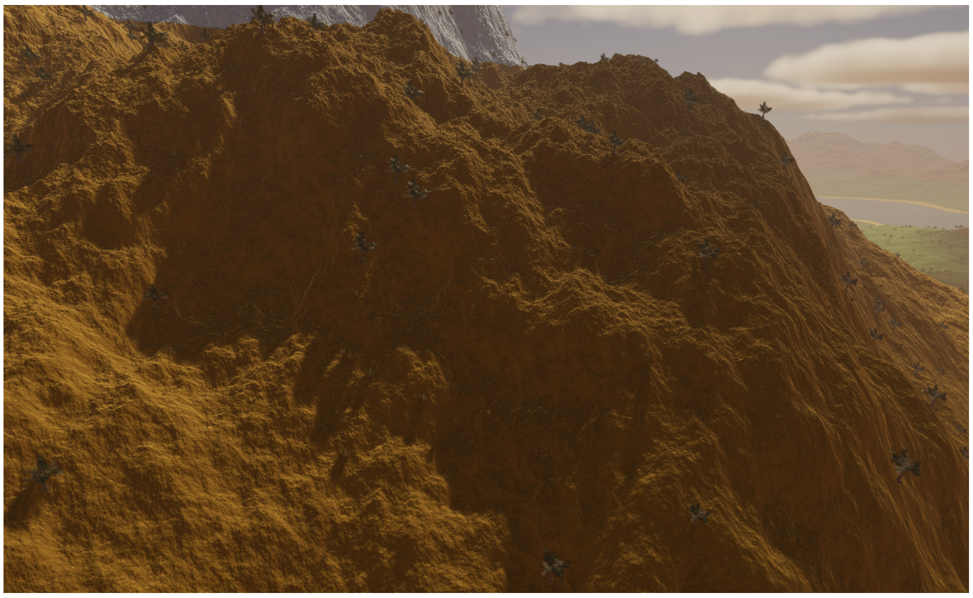


Figure 2.2: Chunk close to the camera (LOD0)

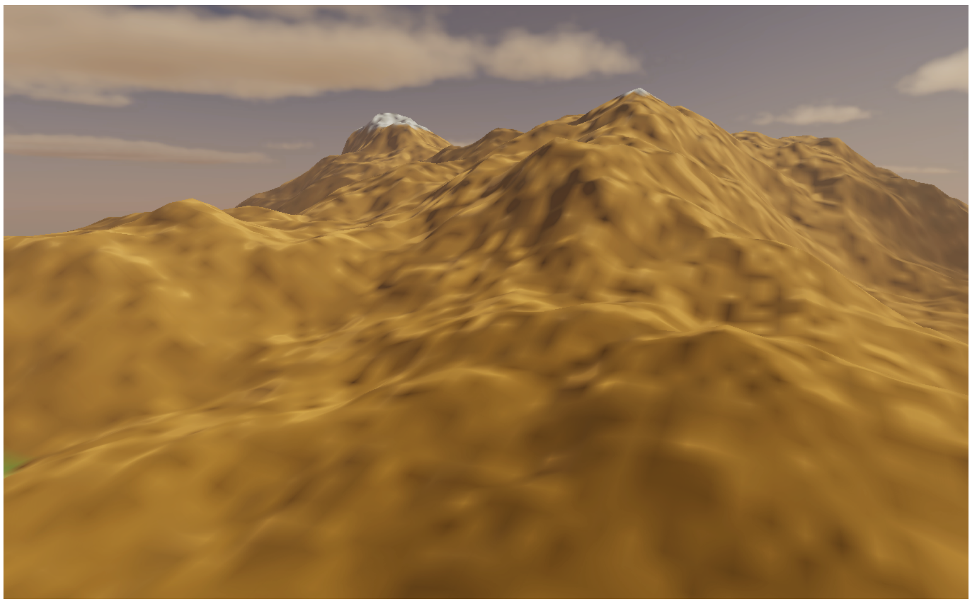


Figure 2.3: Chunk far from the camera (LOD3)

2.5 Tessellation

In computer graphics, tessellation [17] [18] is a process on the GPU in which triangles can be recursively split (subdivided) into more triangles in real time. This is of course slower than subdividing a mesh beforehand and then rendering it, but it's power lies in being able to dynamically change which parts of the mesh get tessellated, and how much, via shaders.

Tessellation is almost exclusively used with height-maps. If a mesh is tessellated but the newly created triangles aren't displaced, and thus multiple triangles are now used in place where one would look the same, then tessellation adds overhead and no visual fidelity. If the newly created triangles can access a height-map to sample displacement, however, it becomes possible to dynamically change the level of detail of meshes.

The typical use-case is to increase the tessellation factor when distance to camera decreases so that more triangles are only rendered if they're going to be seen. This can be used alongside (or in place of) more traditional Level Of Detail.

Figure 2.4 shows how this bachelor thesis uses tessellation. Tessellation factor, which controls how many times triangles get subdivided, decreases with increasing distance to camera.

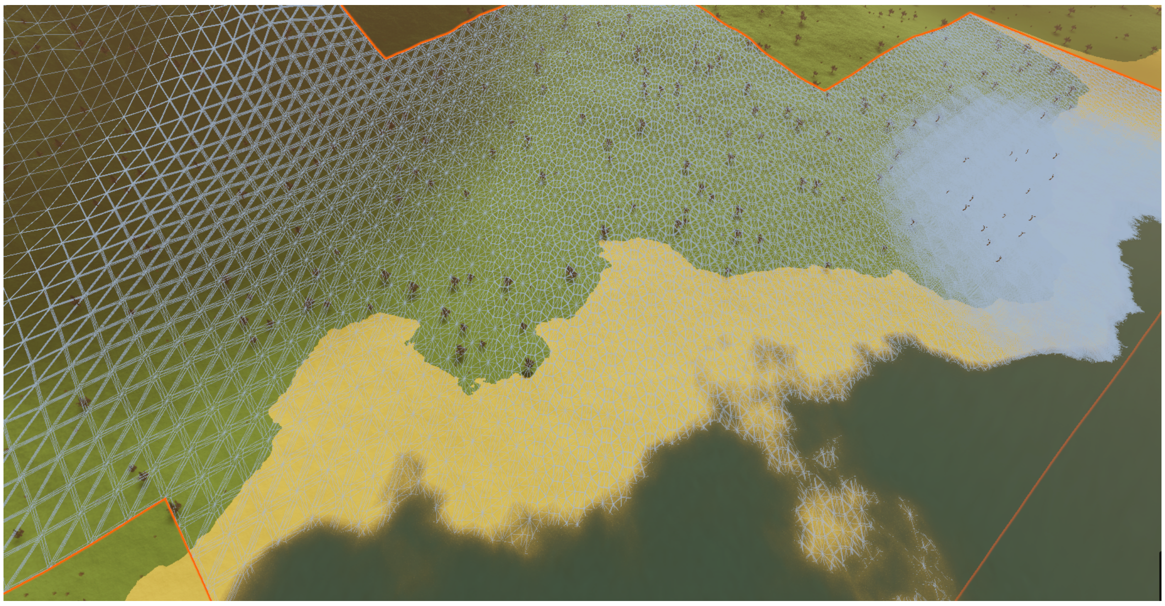


Figure 2.4: Decreasing tessellation factor based on distance to camera on the right

2.6 GPU Parallelism

Cores of graphics cards are slower than processor cores, and are capable of much less instructions. The GPU has many more cores though, for example a modern consumer grade CPU has anywhere from 4 to 24 cores, but a modern GPUs have thousands of cores even up to about 16000. Another difference is that GPUs handle conditional statements much less efficiently than CPUs, it must be noted that GPUs have been getting better and better at them, but they're still nowhere near as good at them as CPUs.

Therefore, algorithms which don't require (a lot of) conditional statements and can be split into many independent parts can potentially see massive performance gains if implemented on graphics cards instead of processors. The obvious example is most image processing algorithms, where the color processing of each pixel can be run in parallel.

Figures 2.5 and 2.5 show how CPUs and GPUs respectively handle a problem which can be split into many independent parallel parts. It doesn't matter which specific problem is being represented, but if we think of these figures as showing image processing, then the red squares represent pixels which haven't been processed yet, blue squares represent pixels which are currently being processed and green squares represent pixels which have already been processed. Note that the CPU (2.5) can only work on a small subset of the problem at a time, while the GPU (2.6) can work on a very large subset of it. Even though the processing speed of each individual pixel is likely faster on the CPU, the GPU solves the problem much faster overall because of its immense potential for parallelism.

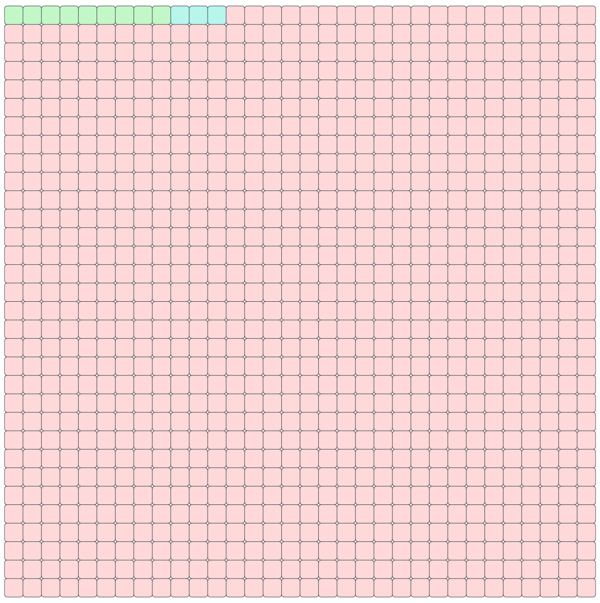


Figure 2.5: CPU parallelism

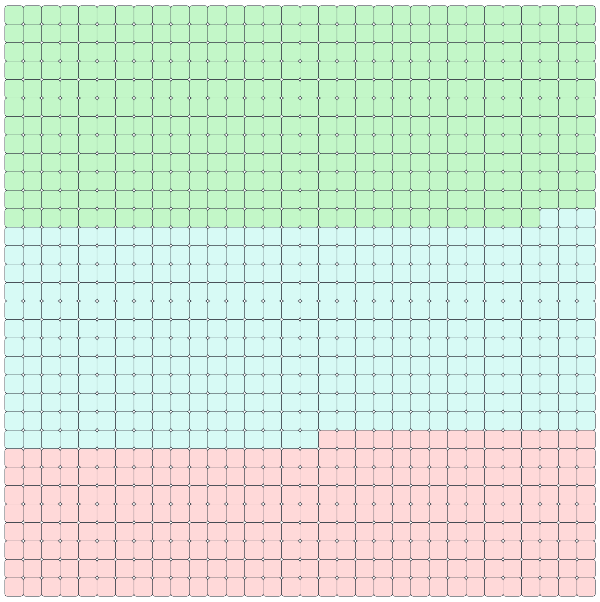


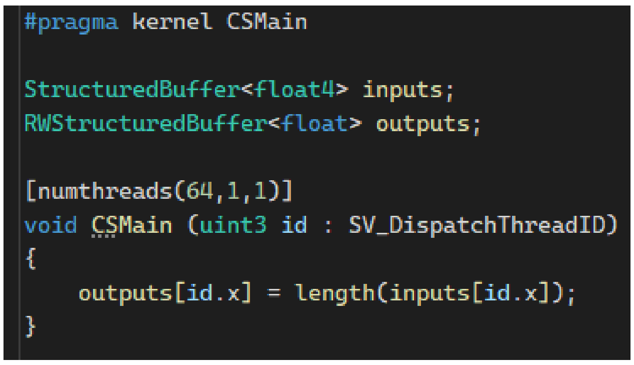
Figure 2.6: GPU parallelism

2.7 Compute Shaders

Following on from the previous section, a shader is a program that executes on a graphics card. There are many kinds of shaders. To name a few, vertex shaders are run on each vertex of a mesh that is being rendered and fragment shaders are run to calculate colors for pixels.

Unlike vertex and fragment shaders, **Compute** shaders [14] don't imply usage in rendering. Their usage is heavily parallel computing of independent data. In my bachelor thesis, I make use of compute shaders to calculate noise functions from which believable terrain can be constructed.

Figure 2.7 shows a very simple compute shader which computes lengths of 4D vectors in parallel. Realistically, compute shaders should be employed for more complicated tasks as time spent sending data to and from the GPU might be greater than the speed improvements of GPU parallelism.



```
#pragma kernel CSMain

StructuredBuffer<float4> inputs;
RWStructuredBuffer<float> outputs;

[numthreads(64,1,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    outputs[id.x] = length(inputs[id.x]);
}
```

Figure 2.7: Simple compute shader

2.8 Noise Functions

Noise functions are very commonly used as a primitive for procedurally generated textures, meaning their samples are usually combined in some way. Noise functions can have many dimensions, but 2D and 3D functions are most common in graphics.

Value noise functions assign pseudo random numbers to lattice points and interpolate between the surrounding lattice points, while Gradient noise functions assign gradients to the lattice points and interpolate between gradient dot products.

To generate terrain in the Unity application, I use Simplex noise (2.8), it is a gradient noise which is simpler to calculate and generates less directional artifacts than classic Perlin noise.

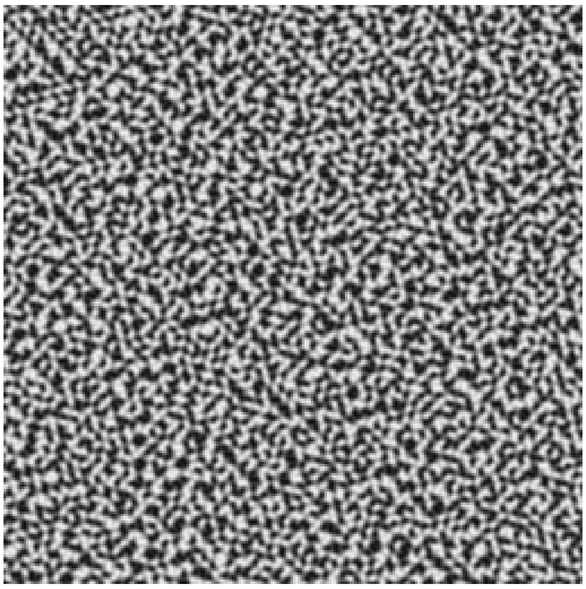


Figure 2.8: Simplex noise

2.9 Other Student's Work

2.9.1 Models of 3D Scenes for a Driving Simulator by Petr Brachaczek

In his bachelor thesis [7], Petr Brachaczek was tasked with creating a large static 3D environment around a road. The main differences between his approach to world creation and mine is the way we use procedural generation and how we adapt terrain to the road/tracks.

He used a software called World Machine which uses procedural generation to create terrain, so while his scene is static, mine is generated dynamically at run-time in a radius around a camera object. His approach has better potential for visual fidelity (because of the option of manual tweaking of terrain), while mine makes it so that there is functionally an infinite world.

To make the terrain adapt to the road, he used a top down gray-scale map where black signified the road and white signified terrain. The path of train tracks in my bachelor thesis is given by an animation curve, so I calculate the distance from the curve for every generated point of terrain and use a sine based weight function to determine how much should the terrain adapt.

2.9.2 Procedural model generation from real maps by Jana Kejvalová

In her master thesis [8], Jana Kejvalová was tasked with using procedural generation to create a realistic model from a 5x5km map. Just like Petr Brachaczek's (2.9.1), her assignment also tasked her with creating a static scene, and she also used World Machine (as well as other, open source, software).

2.9.3 Procedural generation of videogame environments by Jan Kutálek

In his bachelor thesis [9], Jak Kutálek used Houdini Engine to create software which can utilize procedural generation to make mazes for video games, the mazes can then be used in other game engines.

2.9.4 Creation of modular 3D assets for videogames by Alena Mikushina

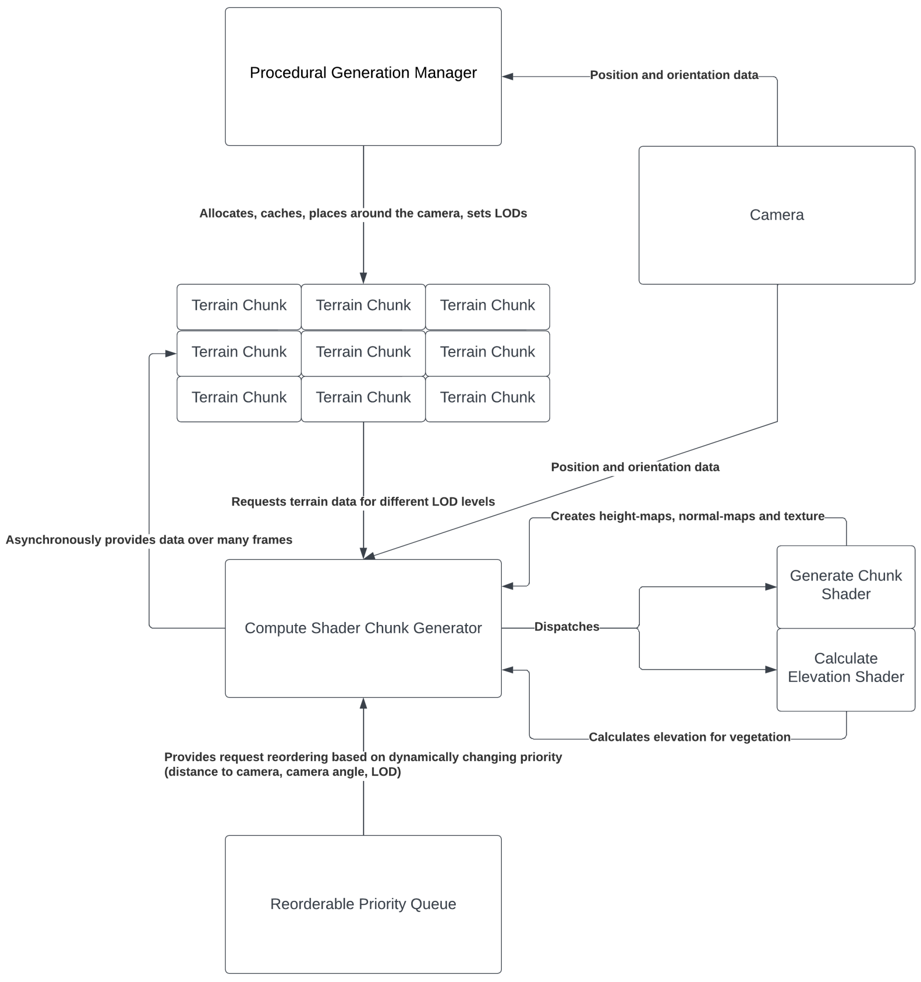
In her bachelor thesis [10], Alena Mikushina examined modelling techniques and modular design. In the practical part, by combining procedural generation and modular design, she achieved quick creation of modular assets.

Chapter 3

Implementation

3.1 Overview and Data Flow

This section serves as a brief summary of the application implementation.



Procedural Generation Manager
Position and orientation data
Camera
Allocates, caches, places around the camera, sets LODs
Terrain Chunk
Terrain Chunk
Terrain Chunk
Terrain Chunk
Terrain Chunk
Terrain Chunk
Terrain Chunk
Terrain Chunk
Terrain Chunk
Position and orientation data
Requests terrain data for different LOD levels
Creates height-maps, normal-maps and texture
Asynchronously provides data over many frames
Generate Chunk Shader
Compute Shader Chunk Generator
Dispatches
Calculate Elevation Shader
Calculates elevation for vegetation
Provides request reordering based on dynamically changing priority (distance to camera, camera angle, LOD)
Reorderable Priority Queue

Figure 3.1: Data Flow

Figure 3.1 explanation:

Procedural Generation Manager reads Camera's position and decides where to place Terrain Chunks, which it also allocates. When Terrain Chunks are placed, the manager also sets LOD for each of the chunks based on it's distance from the Camera.

When the chunk's LOD is set to a higher quality level than it currently is set to, it will ask **Compute Shader Chunk Generator** to asynchronously generate all terrain data for that LOD (height-map, normal-map, texture, vegetation placement). A cancellation token is included in the request so that the generator can tell if the request is no longer needed/valid.

The generator dispatches compute shaders to generate the chunks. It is limited how many shader instances can be dispatched each frame so as to not cause freezing/stuttering of the application (all compute shaders have to finish before the GPU starts rendering the scene). The order in which the chunks have their terrain generated depends on their distance to camera, angle to camera (which both change in time) and the LOD level. When the shaders finish, the corresponding chunk's callbacks are used to transfer the terrain data into it.

When a chunk gets too far away from the camera, either it's level of detail is decreased, in which case the textures it received from the generator are returned to a texture cache and vegetation data is de-allocated, or it is disabled and the manager puts it back into the chunk cache.

3.2 Implementation Details

3.2.1 External Code

There are only two instances of external code in the software project. One is a Vector Swizzle Extension class [3] to add swizzling to Vectors. The other is the srdnoise [4] function in srdnoise.hlsl, which has been copied from Unity's Mathematics package and edited to run in a shader.

3.2.2 Procedural Generation Manager

This class has three main responsibilities:

Dynamically instantiating and caching and placing chunks around a Camera, setting appropriate LOD levels of the chunks, and creating a single quad mesh which every chunk uses to render itself.

Chunk placement and LOD

In order to determine which chunks should be added and which should be removed, the manager keeps two sets of data: a set which contains positions of chunks that have been placed in the scene, and a set which contains positions of chunks that should currently be in the scene. The latter set doesn't have to be kept in-between updates, but doing so allows reusing already allocated heap memory.

Generating the positions of chunks which should **currently** exist around the camera position is done by preparing an empty set, getting the camera's current position, getting the maximum distance for the lowest-quality LOD (the assumption being that the lowest-quality LOD will have the greatest maximum distance), and finally looping through positions of chunks which are close enough in X and Z axis separately (i.e. close enough by Manhattan distance) and adding the positions which are close enough in Euclidean distance to the aforementioned set.

In figure 3.2, the circle represents the camera's position, the squares (both red and green) represent all positions within camera's Manhattan distance and the green squares represent the positions within camera's Euclidean distance (which are also the positions of chunks that should currently exist in the scene)

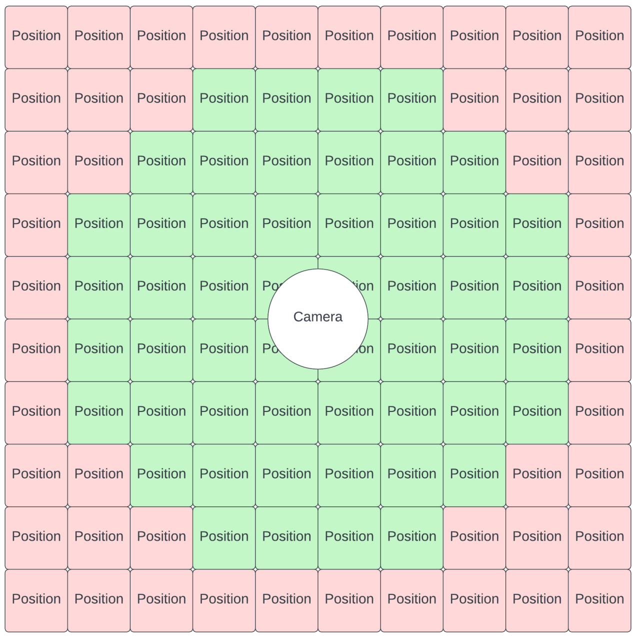
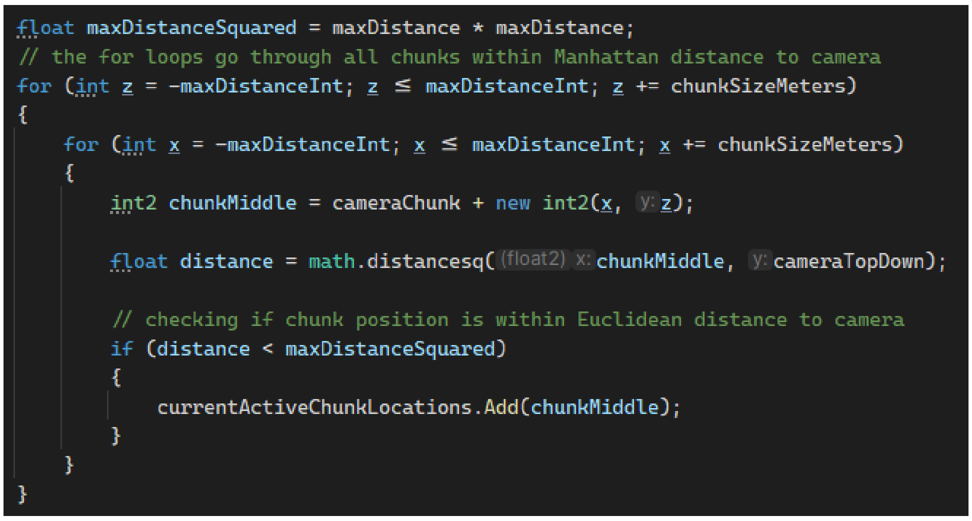


Figure 3.2: Chunks around camera diagram



```
float maxDistanceSquared = maxDistance * maxDistance;
// the for loops go through all chunks within Manhattan distance to camera
for (int z = -maxDistanceInt; z ≤ maxDistanceInt; z += chunkSizeMeters)
{
    for (int x = -maxDistanceInt; x ≤ maxDistanceInt; x += chunkSizeMeters)
    {
        int2 chunkMiddle = cameraChunk + new int2(x,  y: z);

        float distance = math.distancesq( (float2) x: chunkMiddle,  y: cameraTopDown);

        // checking if chunk position is within Euclidean distance to camera
        if (distance < maxDistanceSquared)
        {
            currentActiveChunkLocations.Add(chunkMiddle);
        }
    }
}
```

Figure 3.3: Chunks around camera code

When the current positions of chunks (set C) have been calculated and the previous (old) positions of chunks (set P) are still saved in memory, the positions of chunks which should be added to the scene (blue set A in figure 3.4) as well as the positions of chunks which should be removed from the scene (red set R in figure 3.4) can be determined using the set difference operation such that $A = C - P$, $R = P - C$.

The positions of chunks that should be kept in the scene, that is not removed nor added, (green set K in figure 3.4) can be calculated as $K = C - A$, but this doesn't need to be calculated in the application.

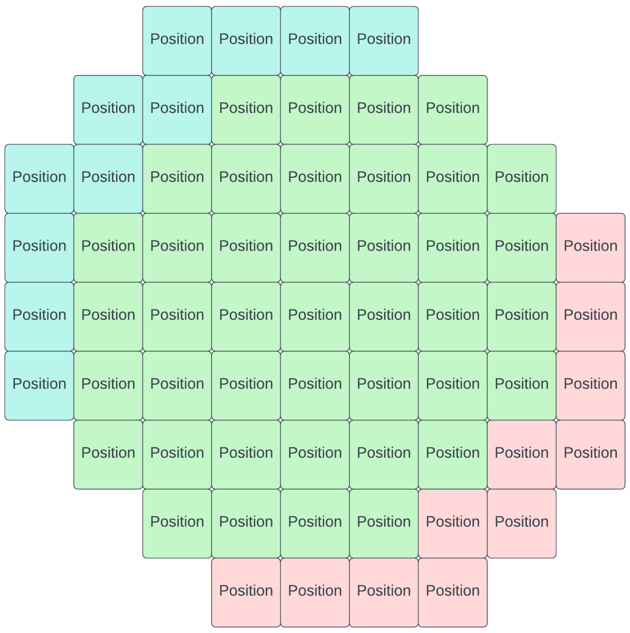
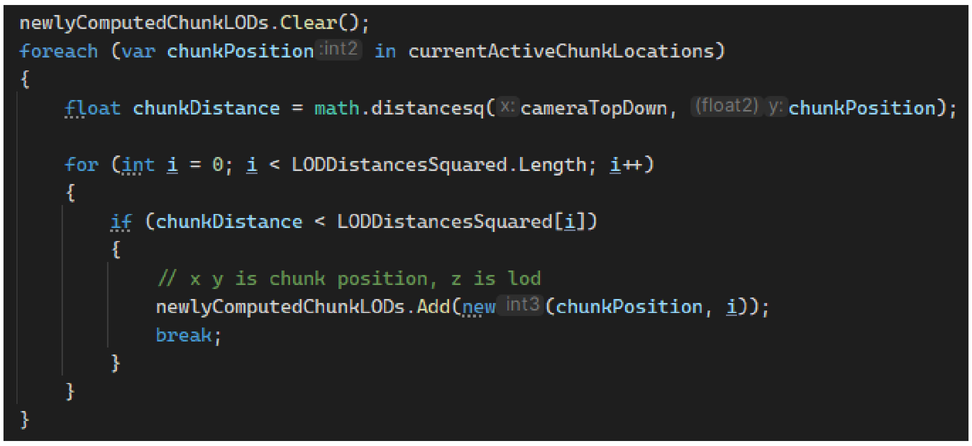

Position Position Position Position
Position Position Position Position Position Position
Position Position Position Position Position Position Position Position
Position Position Position Position Position Position Position Position Position
Position Position Position Position Position Position Position Position Position
Position Position Position Position Position Position Position Position Position
Position Position Position Position Position Position Position Position
Position Position Position Position Position Position
Position Position Position Position

Figure 3.4: Set operations on chunks

The LOD level of each chunk is determined using it's distance to the camera and known max distances each LOD has. The index of the first LOD the maximum distance of which is greater than the chunk's distance to camera is set as the chunk's LOD level (see code snippet in figure 3.5).



```
newlyComputedChunkLODs.Clear();
foreach (var chunkPosition:int2 in currentActiveChunkLocations)
{
    float chunkDistance = math.distancesq(x: cameraTopDown, (float2) y: chunkPosition);

    for (int i = 0; i < LODDistancesSquared.Length; i++)
    {
        if (chunkDistance < LODDistancesSquared[i])
        {
            // x y is chunk position, z is lod
            newlyComputedChunkLODs.Add(new int3(chunkPosition, i));
            break;
        }
    }
}
```

Figure 3.5: Code: Determining LODs for chunks

To optimize performance and keep the application from stuttering, everything mentioned in this section happens only when the camera actually leaves the chunk it is in (which is comparatively much simpler to determine each frame) and it is all calculated on a background thread asynchronously (potentially during multiple frames).

Quad Geometry Creation

When the application is starting, the manager creates a square quad mesh [12] which is made up of smaller squares. The size of the mesh as well as the amount of smaller squares within is adjustable. The final version of the application uses 1600 by 1600 meter quads which are made up by 256 (16 by 16) smaller quads.

Using height-maps and tessellation on this quad mesh, the chunks can appear highly detailed in close proximity to the camera, while taking up little GPU resources at a distance.

3.2.3 Compute Shader Chunk Generator

This Unity Monobehaviour is tightly coupled with the shaders it needs to do its job, therefore I'm putting them both in this section.

The Unity Monobehaviour

The chunk generator's main responsibility is to dispatch (run) compute shaders on the GPU which calculate terrain height-maps, normal maps, textures, and vegetation positions for chunks in the scene. The reason it isn't simply called "chunk generator" is because there is also an unused **Thread Pool Chunk Generator**, which uses CPU cores for these operations instead.

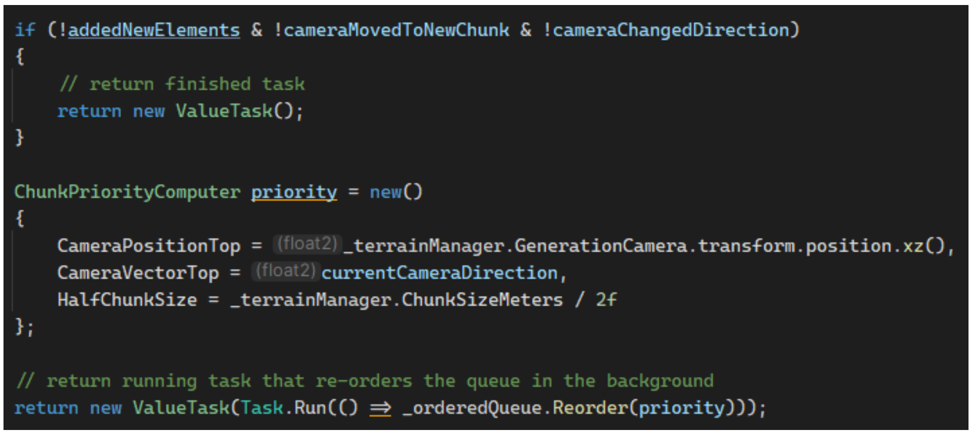
It also contains the points of a Catmull-Rom spline which indicates where the train tracks of a train simulator would be. During startup, it calculates samples of positions on the curve using the Catmull-Rom base matrix, which it then sends to the GPU so that the compute shaders can adapt the generated terrain to it.

Once a request for generating the maps is made, the generator puts it on a dynamic priority queue (see 3.2.4). The generator uses a Unity co-routine in order to spread it's operations across many frames so that it doesn't freeze the application until all requests are handled. In this asynchronous manner, it:

- Optionally reorders the request queue. It does so if the camera has moved into a new chunk, the direction it was looking at changed significantly, or new requests have been added. This is to ensure the priority is dynamically given to chunks depending on their distance and angle to camera and LOD level. See figure 3.6

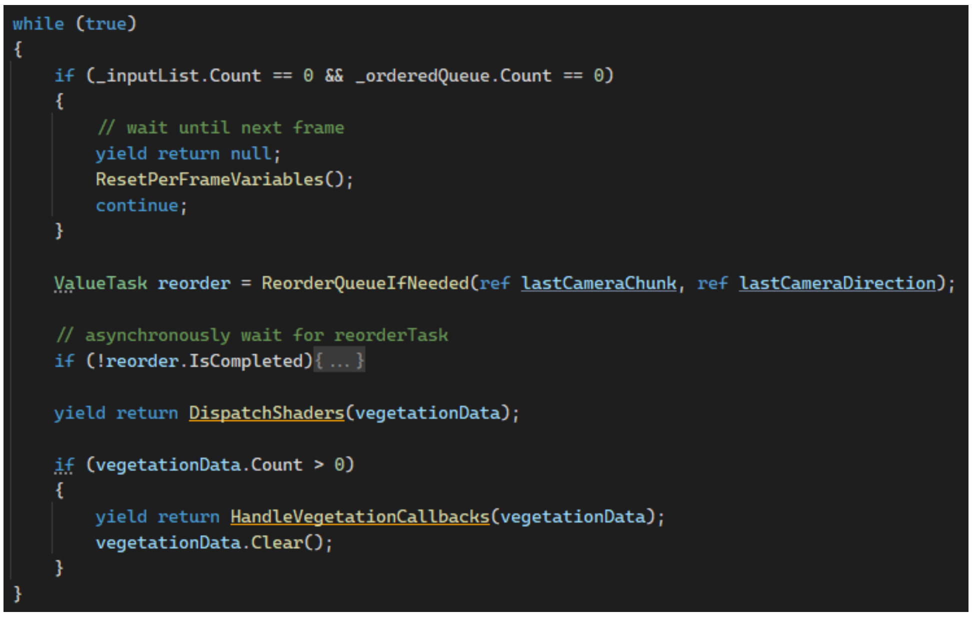
- Dispatches shaders to create maps for a limited number of chunks and also limited by the total amount of map pixels that can be generated each frame. This is because rendering cannot begin before compute shaders are finished processing, so 4k maps for example would make the application stutter/freeze. The final version of the application has this limit set to 1024 pixel resolution, therefore a 4k map computing will be split across at least 16 frames.

- Handles assigning vegetation positions, rotations and scale factors to chunks. This step requires possibly waiting multiple frames, because getting data back from the GPU is rather slow and happens asynchronously.



```
if (!addedNewElements & !cameraMovedToNewChunk & !cameraChangedDirection)
{
    // return finished task
    return new ValueTask();
}

ChunkPriorityComputer priority = new()
{
    CameraPositionTop = (float2)_terrainManager.GenerationCamera.transform.position.xz(),
    CameraVectorTop = (float2)currentCameraDirection,
    HalfChunkSize = _terrainManager.ChunkSizeMeters / 2f
};

// return running task that re-orders the queue in the background
return new ValueTask(Task.Run(() => _orderedQueue.Reorder(priority)));
```

Figure 3.6: Code: Reordering request queue if needed



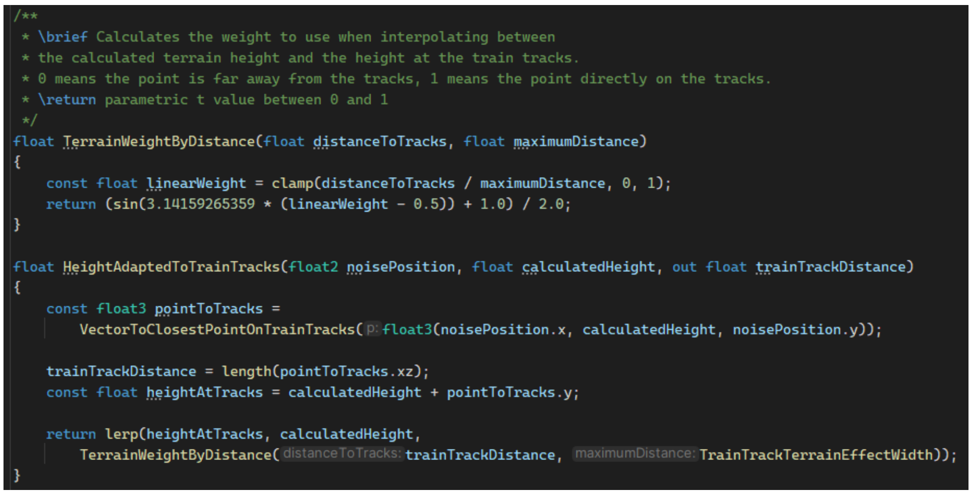
```
while (true)
{
    if (_inputList.Count == 0 && _orderedQueue.Count == 0)
    {
        // wait until next frame
        yield return null;
        ResetPerFrameVariables();
        continue;
    }

    ValueTask reorder = ReorderQueueIfNeeded(ref lastCameraChunk, ref lastCameraDirection);

    // asynchronously wait for reorderTask
    if (!reorder.IsCompleted){ ... }

    yield return DispatchShaders(vegetationData);

    if (vegetationData.Count > 0)
    {
        yield return HandleVegetationCallbacks(vegetationData);
        vegetationData.Clear();
    }
}
```

Figure 3.7: Code: Chunk generator's coroutine's update loop

The Compute Shaders

Generate Chunk Shader and **Calculate Elevation Shader** are used by the **Compute Shader Chunk Generator** to generate the needed maps and vegetation elevations respectively.

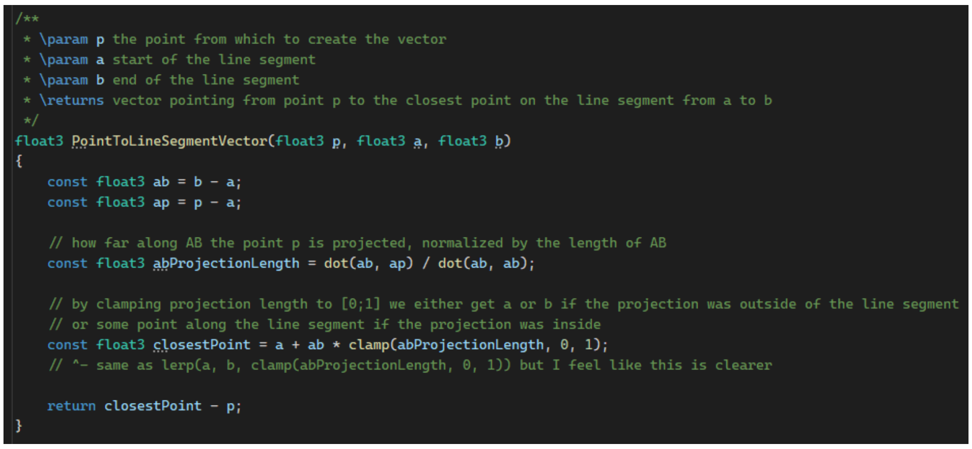
The most important part of both of these shaders is the function for calculating terrain height at a specific point. It's argument is a 2D world position at which to calculate the height, and its outputs are the calculated height adapted to the train tracks (see figure 3.8) and the distance to them (the tracks are defined by samples stored on the GPU that have been calculated by the generator).



```
/**
 * \brief Calculates the weight to use when interpolating between
 * the calculated terrain height and the height at the train tracks.
 * 0 means the point is far away from the tracks, 1 means the point directly on the tracks.
 * \return parametric t value between 0 and 1
 */
float TerrainWeightByDistance(float distanceToTracks, float maximumDistance)
{
    const float linearWeight = clamp(distanceToTracks / maximumDistance, 0, 1);
    return (sin(3.14159265359 * (linearWeight - 0.5)) + 1.0) / 2.0;
}

float HeightAdaptedToTrainTracks(float2 noisePosition, float calculatedHeight, out float trainTrackDistance)
{
    const float3 pointToTracks =
        VectorToClosestPointOnTrainTracks(p: float3(noisePosition.x, calculatedHeight, noisePosition.y));

    trainTrackDistance = length(pointToTracks.xz);
    const float heightAtTracks = calculatedHeight + pointToTracks.y;

    return lerp(heightAtTracks, calculatedHeight,
        TerrainWeightByDistance(distanceToTracks: trainTrackDistance, maximumDistance: TrainTrackTerrainEffectWidth));
}
```

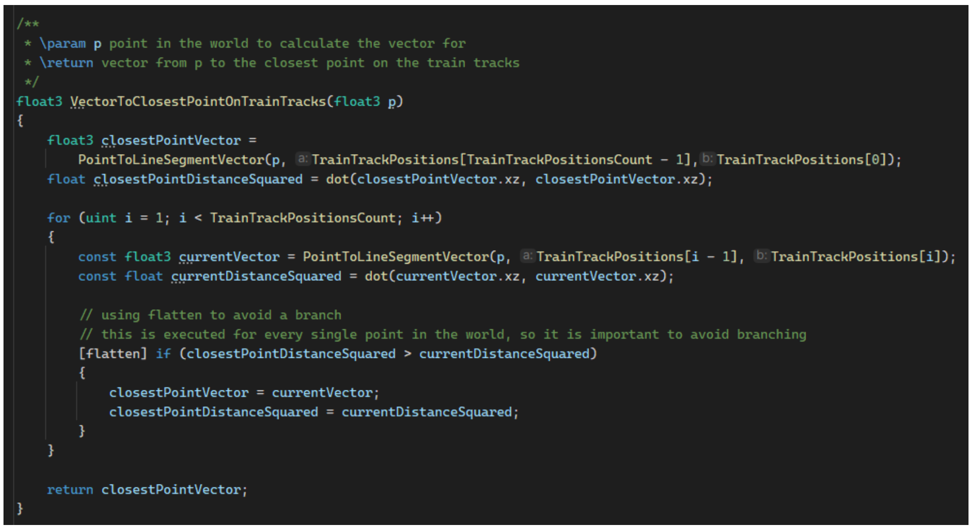
Figure 3.8: Adapting the calculated height to the tracks

The distance to train tracks is calculated by connecting each two adjacent spline samples by a line segment, then calculating the distances from the point in the world to each of the line segments (see figure 3.9), and finally returning the minimum of these distances (see figure 3.10).



```
/**
 * \param p the point from which to create the vector
 * \param a start of the line segment
 * \param b end of the line segment
 * \returns vector pointing from point p to the closest point on the line segment from a to b
 */
float3 PointToLineSegmentVector(float3 p, float3 a, float3 b)
{
    const float3 ab = b - a;
    const float3 ap = p - a;

    // how far along AB the point p is projected, normalized by the length of AB
    const float3 abProjectionLength = dot(ab, ap) / dot(ab, ab);

    // by clamping projection length to [0;1] we either get a or b if the projection was outside of the line segment
    // or some point along the line segment if the projection was inside
    const float3 closestPoint = a + ab * clamp(abProjectionLength, 0, 1);
    // ^- same as lerp(a, b, clamp(abProjectionLength, 0, 1)) but I feel like this is clearer

    return closestPoint - p;
}
```

Figure 3.9: Vector to the closest point on a line segment

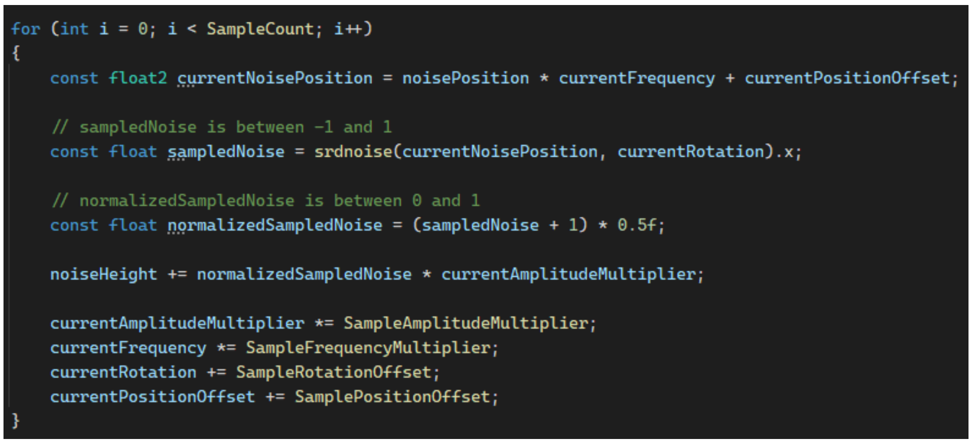


```
/**
 * \param p point in the world to calculate the vector for
 * \return vector from p to the closest point on the train tracks
 */
float3 VectorToClosestPointOnTrainTracks(float3 p)
{
    float3 closestPointVector =
        PointToLineSegmentVector(p, a: TrainTrackPositions[TrainTrackPositionsCount - 1], b: TrainTrackPositions[0]);
    float closestPointDistanceSquared = dot(closestPointVector.xz, closestPointVector.xz);

    for (uint i = 1; i < TrainTrackPositionsCount; i++)
    {
        const float3 currentVector = PointToLineSegmentVector(p, a: TrainTrackPositions[i - 1], b: TrainTrackPositions[i]);
        const float currentDistanceSquared = dot(currentVector.xz, currentVector.xz);

        // using flatten to avoid a branch
        // this is executed for every single point in the world, so it is important to avoid branching
        [flatten] if (closestPointDistanceSquared > currentDistanceSquared)
        {
            closestPointVector = currentVector;
            closestPointDistanceSquared = currentDistanceSquared;
        }
    }

    return closestPointVector;
}
```

Figure 3.10: Vector to the closest point on the tracks

The shader generates what looks to be believable terrain by sampling [1] simplex noise many times. If it only sampled the noise a single time (or a just a few times), the resulting terrain would look extremely simplistic and unnaturally smooth (see figure 3.12). To add variation and roughness to the terrain, the noise function is sampled many times (25 in the final build) with ascending frequency and descending amplitude as well as changing offset and rotation (see figure 3.11). This way, the low frequency high amplitude samples will shape the terrain's mountains, while the higher frequency lower amplitude samples will shape smaller hills, mountain details and ground irregularities. The screenshots after this text show the difference between 1 (figure 3.12), 5 (figure 3.13), 10 (figure 3.14) and 25 (figure 3.15) samples, as well as what can happen (because of floating point number rounding errors) when sample count is set too high (figure 3.16).

The normal map is calculated by sampling 3 height values per pixel (where one of these values goes into the Height-Map) and then using cross products to get the normal vector. At one point I've implemented the normal map calculation by using analytical derivatives, but I've found that the sampling approach looks better most of the time, especially on distant chunks which have very big gaps between samples.



```
for (int i = 0; i < SampleCount; i++)
{
    const float2 currentNoisePosition = noisePosition * currentFrequency + currentPositionOffset;

    // sampledNoise is between -1 and 1
    const float sampledNoise = srdnoise(currentNoisePosition, currentRotation).x;

    // normalizedSampledNoise is between 0 and 1
    const float normalizedSampledNoise = (sampledNoise + 1) * 0.5f;

    noiseHeight += normalizedSampledNoise * currentAmplitudeMultiplier;

    currentAmplitudeMultiplier *= SampleAmplitudeMultiplier;
    currentFrequency *= SampleFrequencyMultiplier;
    currentRotation += SampleRotationOffset;
    currentPositionOffset += SamplePositionOffset;
}
```

Figure 3.11: Sampling simplex noise

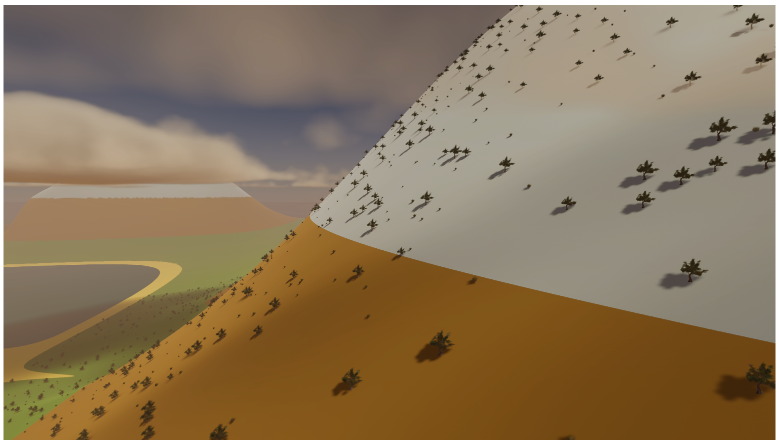


Figure 3.12: Terrain using 1 sample

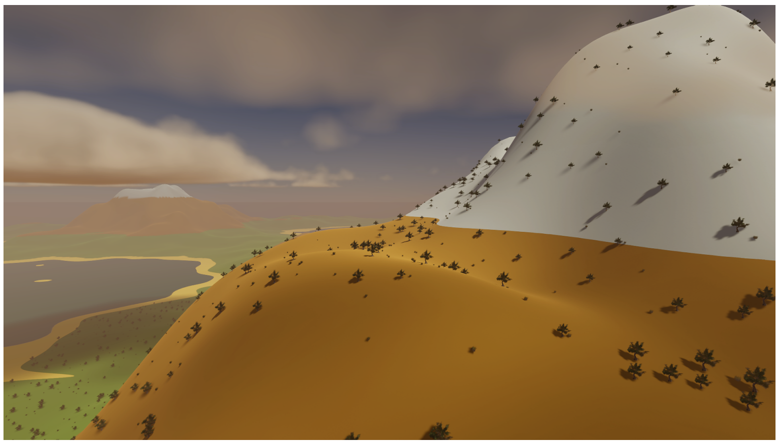


Figure 3.13: Terrain using 5 samples

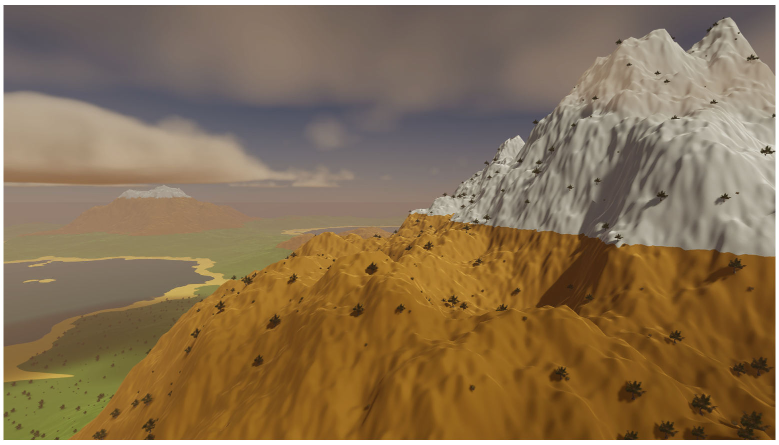


Figure 3.14: Terrain using 10 samples

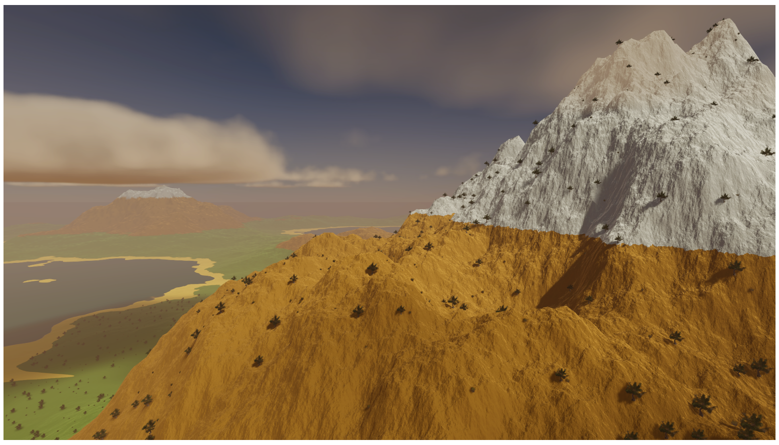


Figure 3.15: Terrain using 25 samples

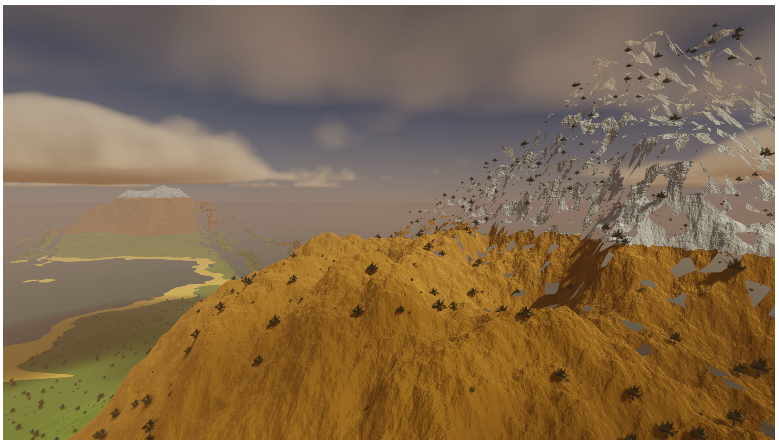


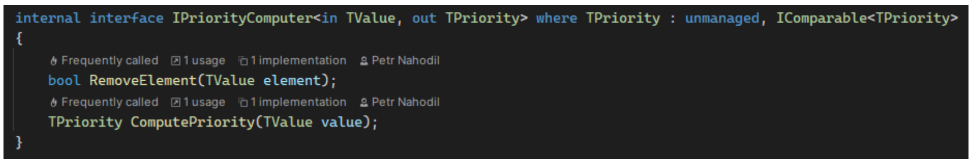
Figure 3.16: Terrain using 68 samples (broken because of floating point inaccuracies)

3.2.4 Reorderable Priority Queue

This class is mainly used by the generator to encapsulate the logic of reordering the chunk request queue. It has En-queue and De-queue methods to push and pop elements from it, and a Reorder method to sort the elements based on a newly computed priorities.

This method has to have a way to compute a priority for each element, and normally the way to express that in C# would be using Func<TValue,TPriority> (a reference to a function accepting an element from the queue and returning a priority value which can be compared to other priority values). The downside is that this approach implies every call to this function is dynamically dispatched at runtime, which could slow down the reordering.

Instead, I'm using an interface that acts as a substitute to a function reference. By using generic constraints to make sure the object being used as the de-facto function pointer is a struct, it is possible to inline the method computing the priority.



```
internal interface IPriorityComputer<in TValue, out TPriority> where TPriority : unmanaged, IComparable<TPriority>
{
    Frequently called  1 usage  1 implementation  Petr Nahodil
    bool RemoveElement(TValue element);
    Frequently called  1 usage  1 implementation  Petr Nahodil
    TPriority ComputePriority(TValue value);
}
```

Figure 3.17: Priority Computer Interface

3.2.5 Terrain Chunk

The terrain chunk's lifetime and placement is done by the **Procedural Generation Manager** (3.2.2) and it's terrain data (height-map, normal map, texture) is calculated by the **Compute Shader Chunk Generator** (3.2.3).

Responsibilities of the terrain chunk are:

- Holding it's terrain data
- Knowing which LOD level it is supposed to be using ("wanted" LOD level)
- Requesting for the generator (3.2.3) to calculate terrain data based on wanted LOD level
- Not holding onto terrain data of LOD level lower (better) than wanted LOD (when wanted LOD level is set to a lower quality LOD than it was previously, the chunk will free (or return to cache) the resources associated any LOD level lower (better) than wanted LOD)
- Using the highest quality LOD level available to it (it will use LOD levels of lower quality if it's wanted LOD level hasn't been calculated yet)

To render itself, the terrain chunk uses a tessellation shader created in Unity's shader graph [19]. This shader, meant to run on a quad mesh with 0 height, takes a height-map and uses it for tessellation vertex displacement. The shader reduces the tessellation factor (i.e. the amount of times a triangle is subdivided) based on the distance to camera, as can be seen in figure 3.18 (also 2.4).

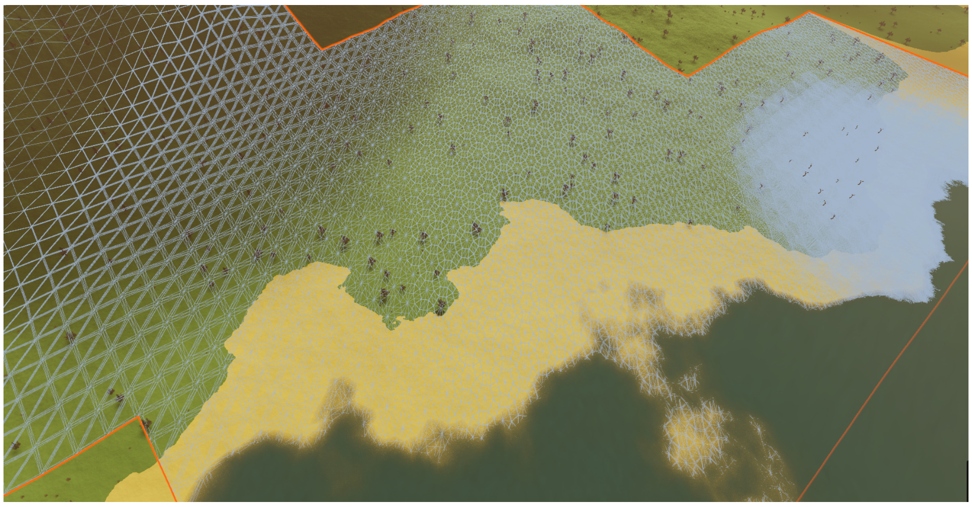


Figure 3.18: Tessellation factor decreasing with distance to camera

The terrain data each chunk holds also includes an optional (nullable) array of matrices which represent the translation, rotation and scale of the chunk's vegetation. I first tried keeping vegetation as Unity Game Objects, but that resulted in very slow rendering. Only keeping the TRS matrices and calling Unity's Graphics class to render the vegetation significantly helped performance. Only two vegetation models are used, a tree [6] and a bush [5].

3.2.6 Burst Friendly Cancellation Token Source

The cancellation token source is a simple class used for cancelling any operation. It can hand out cancellation tokens, each cancellation token is capable of checking if cancellation has been requested on the source. It is heavily inspired by **CancellationTokenSource** from the .Net standard library, though it is much simpler in its implementation.

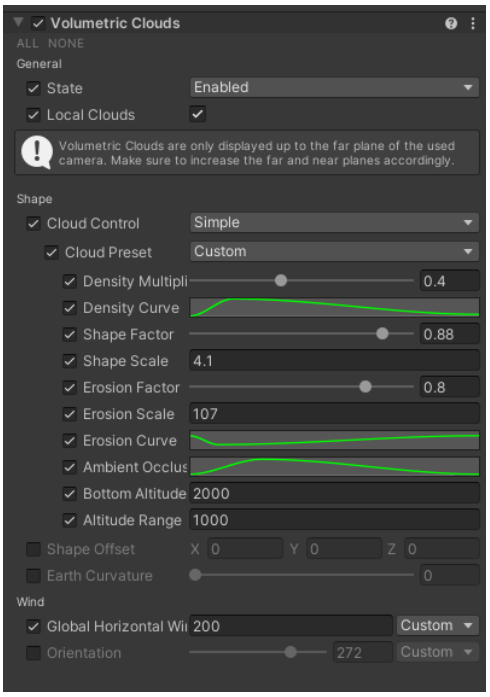
The main usage in the final application is to create a cancellation token source for every request a chunk makes to the chunk generator, and cancel the request if the chunk gets disabled (that is, if the manager removes it from the scene).

The "burst friendly" part of the class's name means that it can also hand out special cancellation tokens which can be used in native code outside of the context of the garbage collector. That is needed in order to check for cancellation in **burst compiled** functions. I didn't bring attention to what a "burst compiled" function is in the Theoretical Background (2) chapter because while I used them during development, I have switched to using GPU shaders instead.

Regardless, I've continued using this class for cancellations in the managed environment (where the garbage collector is handling object lifetimes) even though I don't have a use for the "burst friendly" aspect anymore.

3.2.7 Lighting, Clouds, Shadows, Water, Fog

I used Unity's HDRP (High Definition Render Pipeline) to add a sun, physically based sky, volumetric clouds, volumetric fog, dynamic shadows, and a water level with caustics into the project. The introduction of these aspects was therefore just a matter of setting up the HDRP profile. Figure 3.19 shows how clouds were set up as an example.



Volumetric Clouds

ALL NONE

General

✓ State	Enabled
✓ Local Clouds	✓

Volumetric Clouds are only displayed up to the far plane of the used camera. Make sure to increase the far and near planes accordingly.

Shape

✓ Cloud Control	Simple
✓ Cloud Preset	Custom
✓ Density Multipli	0.4
✓ Density Curve	
✓ Shape Factor	0.88
✓ Shape Scale	4.1
✓ Erosion Factor	0.8
✓ Erosion Scale	107
✓ Erosion Curve	
✓ Ambient Occlus	
✓ Bottom Altitude	2000
✓ Altitude Range	1000
Shape Offset	X 0 Y 0 Z 0
Earth Curvature	0

Wind

✓ Global Horizontal Wi	200 Custom
Orientation	272 Custom

Figure 3.19: Setting up clouds in Unity

3.3 Hurdles And Workarounds

3.3.1 Texture2D Allocation

Texture2D, which is a class that needs to be used to create a texture at run-time in Unity, normally exists in both the CPU memory (RAM) and GPU memory (VRAM). There's no way to tell Unity not to allocate the CPU side of this texture, therefore the workaround is to first create it, then call **Apply()** with a boolean parameter which uploads whatever is in the CPU side to the GPU side.

The upload operation unfortunately creates noticeable stutter for larger textures (4k), I implemented a texture pool so that this stutter only happens at the beginning of the application before stabilizing, but in the end even that initial stuttering was quite distracting, so I opted **not** to clear the CPU memory since there's no way to truly get rid of it's negative effects on frame-rate smoothness.

The one texture type that can be created just on the GPU side is **RenderTexture**, but that texture type is more volatile - it can be deallocated outside of my control on certain events, so I couldn't use it in place of **Texture2D**.

3.3.2 Vegetation Asymptotic Complexity

When I first implemented vegetation by storing only each instance's translation-rotation-scale matrix, it seemed to be very slow and I didn't know why. It turned out that when creating those matrices, I forgot to clear the buffer into which I was saving them before copying them into an array, therefore every chunk that had vegetation was rendering it's own vegetation AND all vegetation of previous chunks, resulting in $O(N^2)$ complexity.

3.4 Unused Classes

3.4.1 Thread Pool Chunk Generator

A chunk generator that had very similar function to the Compute Shader Chunk Generator (3.2.3), the main difference being it used a custom thread pool to generate all terrain data on the CPU. I've chosen this approach at first because I was more familiar with implementing these algorithms on the CPU than on the GPU (not to mention anything written to run on the GPU is harder to debug) and because I thought it'd be fast enough.

After implementing a non-trivial terrain function, I realized this approach was just too slow. 32x32 textures were being computed in a manner of milliseconds and 512x512 textures took about half a second to compute. That speed was acceptable, but 4K textures (4096x4096) took between 10 and 15 seconds each. 4K textures are used for the highest quality LOD, and it took away from visual fidelity of the application to have to wait this long for a newly entered chunk to become more detailed.

3.4.2 Blocking Priority Queue

This priority queue was similar to the Reorderable Priority Queue (3.2.4) with the main distinction being it was also thread safe (safe for concurrent usage by multiple CPU threads).

This was needed back when the chunk generator being used ran on a thread pool (3.4.1) as opposed to using compute shaders (3.2.3). After I stopped using the thread pool chunk generator, I also decided to rewrite this queue into a single-threaded version (3.2.4) to get rid of the now needless complexity of thread safety.

Chapter 4

Results

4.1 Limitations

4.1.1 Not Combined With Train Simulator

The bachelor thesis was meant to **ideally** be combined with a bachelor thesis of another student (who was working on a train simulator). The assignments of our individual theses were formulated so that they'd be defendable without combining them, but it would've been nice to have them combined. Neither of us had time to spend on combining them unfortunately.

4.1.2 No UI or Settings

I was planning on making a simple UI in which the user would be able to change some parameters of world generation, including the maximum LOD distances, texture resolutions corresponding to LOD levels, the amount of vegetation, and last but not least the train track spline.

These values aren't hard-coded though, they're set up as Unity's **SerializeField** private fields, from which I load values at scene startup. It wouldn't require any major changes to allow for UI settings, but I unfortunately ran out of time to implement this feature.

4.1.3 Lack of Textures and Simplistic Height-Based Coloring

I planned to add seamlessly tiled textures with normal maps for sand, grass, rocks and snow and use them on the terrain instead of the simple 4 color system. Also, I'd have liked to map the locations of these biomes using a separate noise function and make the divisions/borders between them smoother.

4.1.4 Smooth Normals Around the Track

Because of the way I'm handling terrain adaptation to the track (that is, by using a weight function to interpolate between the noise-generated elevation and the spline-defined track height, see 3.8) the normals around the tracks are somewhat too smooth. In other words, the terrain around the tracks maps the weight function too closely for my liking.

I'd have liked to keep some more terrain irregularities and/or add another layer of normal mapping via detailed sand/grass textures (as discussed in the previous subsection) but, while it'd add visual fidelity, I've had to prioritize other aspects of the application to finish this bachelor thesis in a timely manner.

4.1.5 Vegetation Rendering

The vegetation, as it currently stands, even after making it $O(N)$ instead of $O(N^2)$ (see 3.3.2), couldn't be set to appear more often or render very far. I unfortunately didn't have much time to implement proper vegetation LOD or other optimisations.

4.2 Benchmarks

4.2.1 Testing Methodology

For testing, a build of the application was made with a specific camera rotation and orientation. Camera wasn't moved or rotated in the benchmark. Frames per second were benchmarked on several systems after the initial world generation overhead when all of the generation was finished. See figure 4.1 for a screenshot of benchmarked scene view.

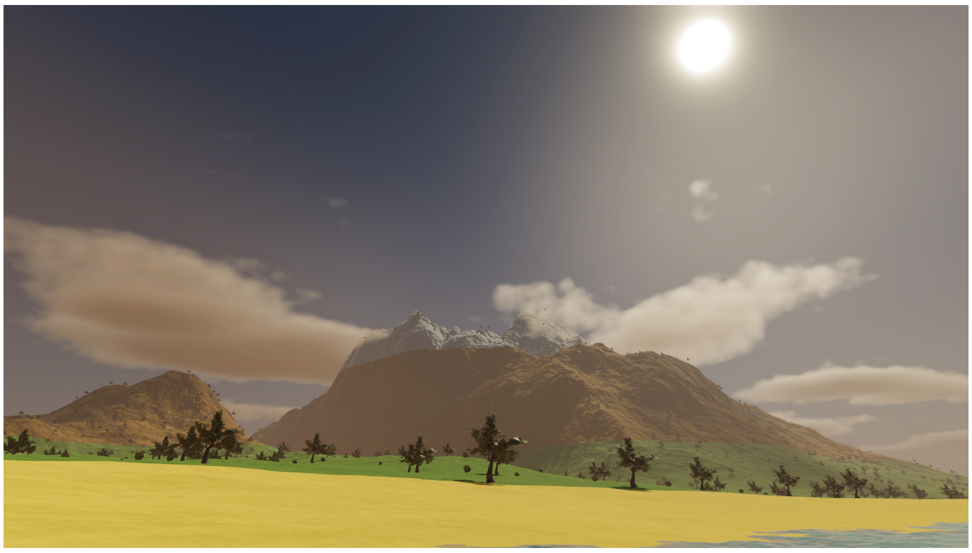


Figure 4.1: Benchmark scene view

4.2.2 Benchmark results

System Type	Processor	Graphics Card	Resolution	Average FPS
Laptop	Intel i9-11980HK	Nvidia RTX 3080	1920x1080	132
Desktop	AMD Ryzen 7 3800X	Nvidia RTX 3070	2560x1440	87
Desktop	AMD Ryzen 5 5600X	AMD RX 580	1920x1080	30

Table 4.1: Benchmark table

It should be noted that the application takes up about 7.4GB of VRAM.

In terms of performance of terrain generation itself (not just the per-frame rendering which is the main contributor to the FPS results) it was very fast on every system tested - after the initial 20 seconds to a minute when the entire radius around the camera generated for the first time there wasn't any noticeable generation overhead. Any time a camera entered a new chunk, all the chunks that were added or had their LOD updated finished doing so before the camera could move to another new chunk. Given the fairly high speed of the camera, I'd say the speed at which terrain was generated was very quick owing to the speed of compute shaders.

I had an opportunity to run the application on one more system (AMD Ryzen 7 3800X CPU, AMD RX Vega 64 GPU), but it would always crash when starting. Reading the crash dump files, I was able to determine the problem was that the graphics API (vulkan) didn't support video decoding to textures on that system. I think the application crashes with this error because I use Render Textures (textures which can be written to on the GPU) to represent the generated terrain.

When I tried using different graphics APIs, I've found that the application won't even start on my machine using DX12, and it won't show vegetation nor adapt terrain to train tracks on DX11, possibly because those rely on compute buffers.

4.2.3 Benchmarks Conclusion

Aside from the system on which the application wouldn't run, I think the performance, even though it could always be better, isn't bad. When iterating, I could only see performance on my machine (the laptop from the results table), which I aimed to get above 100 frames per second at least. On a similarly modern system which uses an RTX 3070 GPU, the performance isn't bad either, especially when considering the increased monitor resolution.

I was worried how the application would run on slightly older mid-range systems, in the end the results on the RX 580 GPU ended up being better than I expected. 30 frames per second, while not the smoothest, is still serviceable. To put the performance in perspective, the vast majority of games on the immensely popular Play-Station 4 gaming console (which was being manufactured and sold in the years 2013-2020) ran at 30 frames per second, too.

4.3 Screenshots

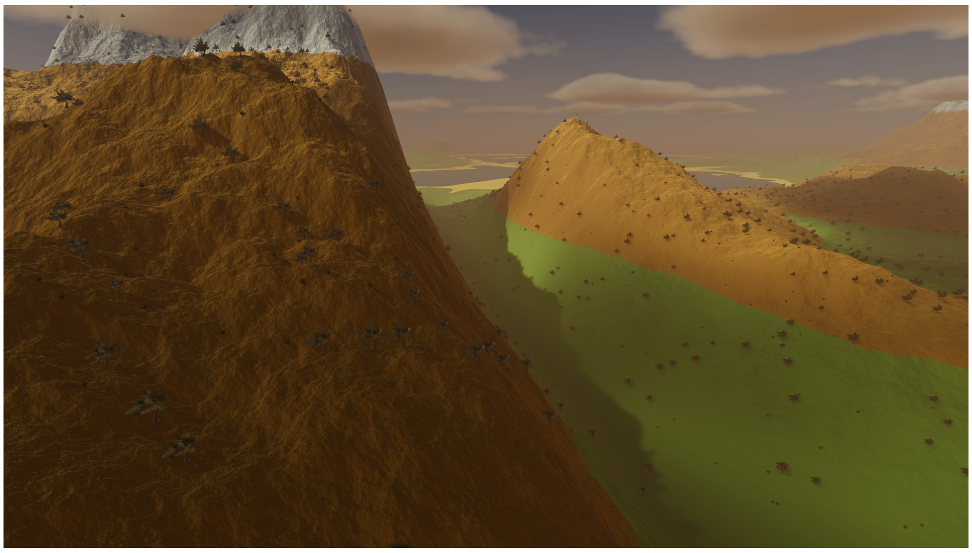


Figure 4.2: Terrain adapted to train track spline

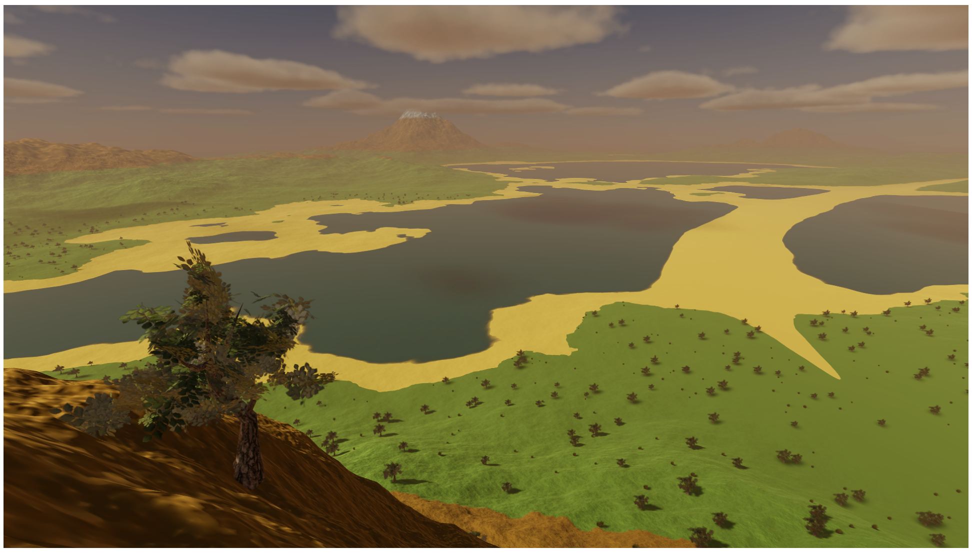


Figure 4.3: Lakes around train track spline

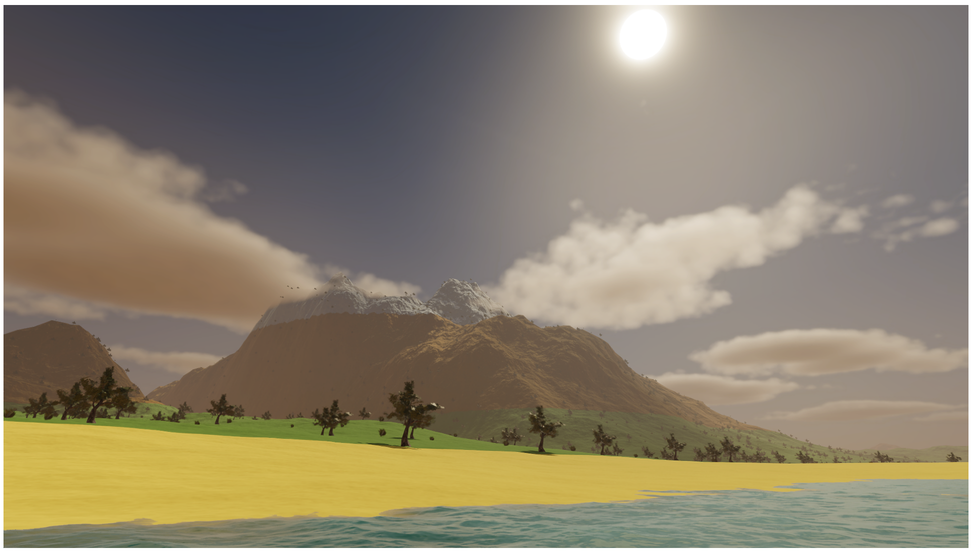


Figure 4.4: Water, sun and clouds

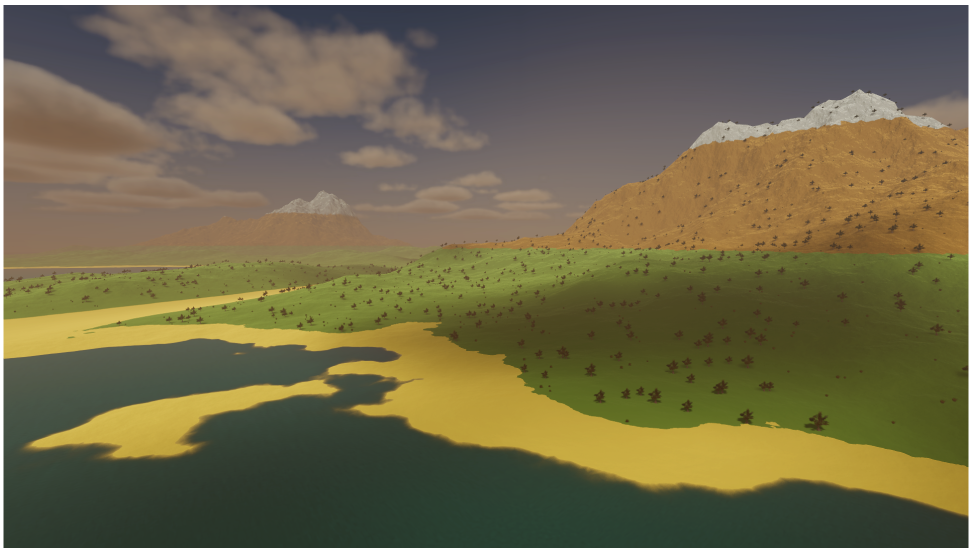


Figure 4.5: Cloud shadows

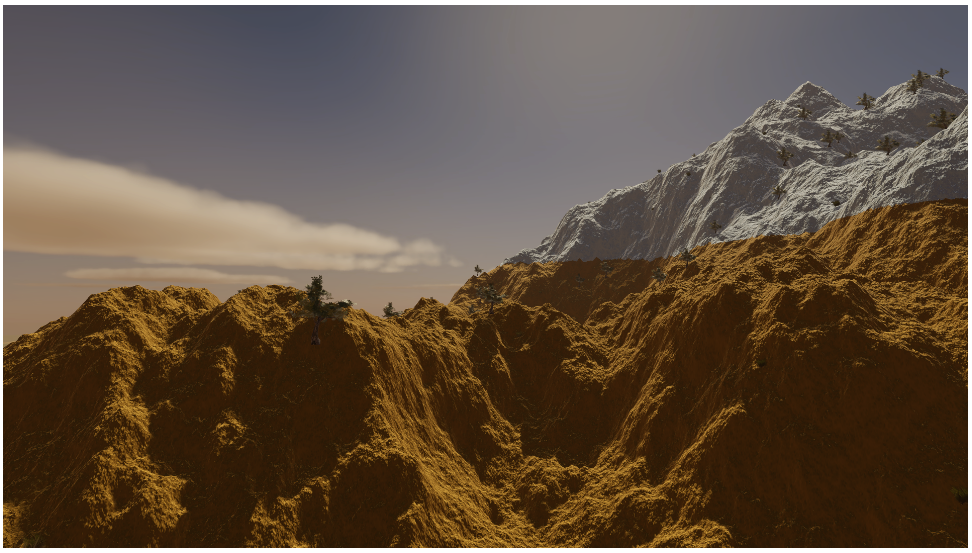


Figure 4.6: Shadows on rocky terrain

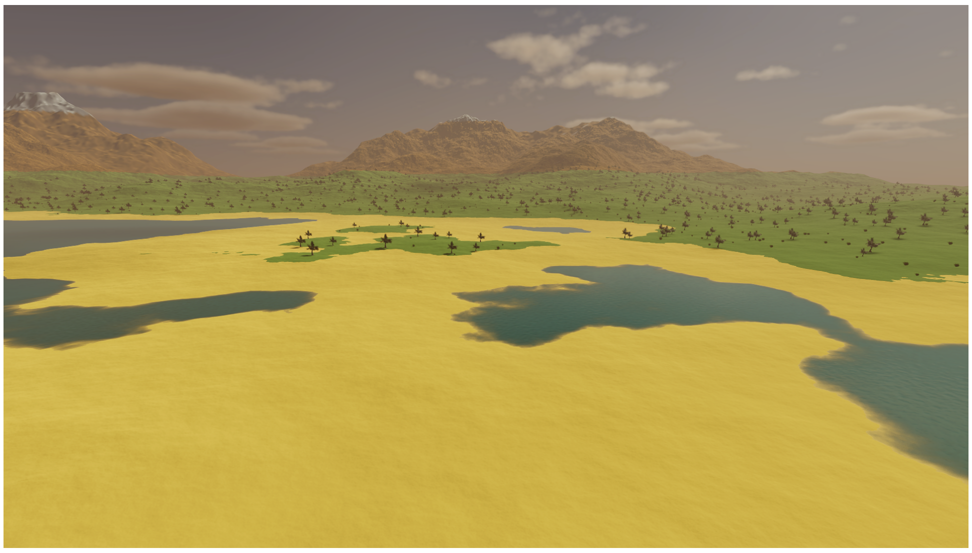


Figure 4.7: Lowlands

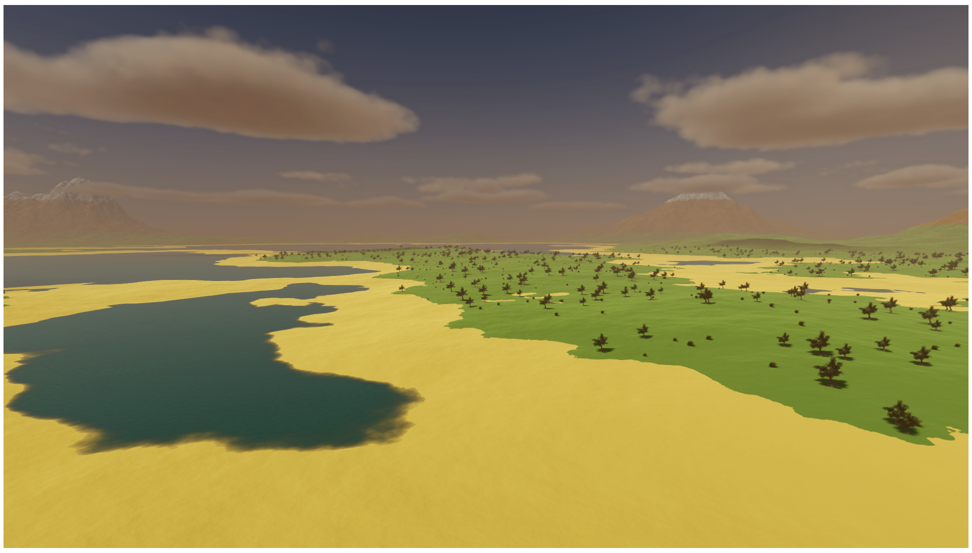


Figure 4.8: Lowlands

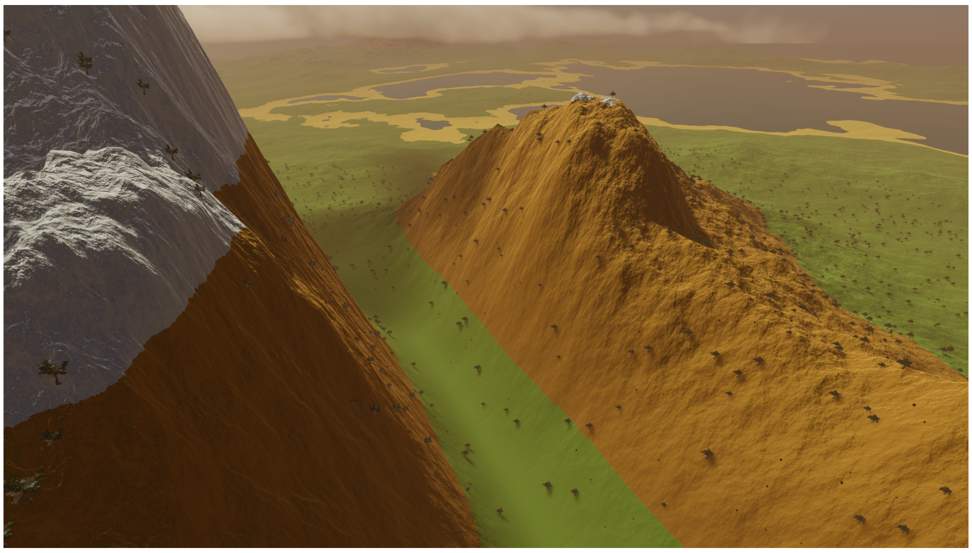


Figure 4.9: Terrain adapted to train track spline

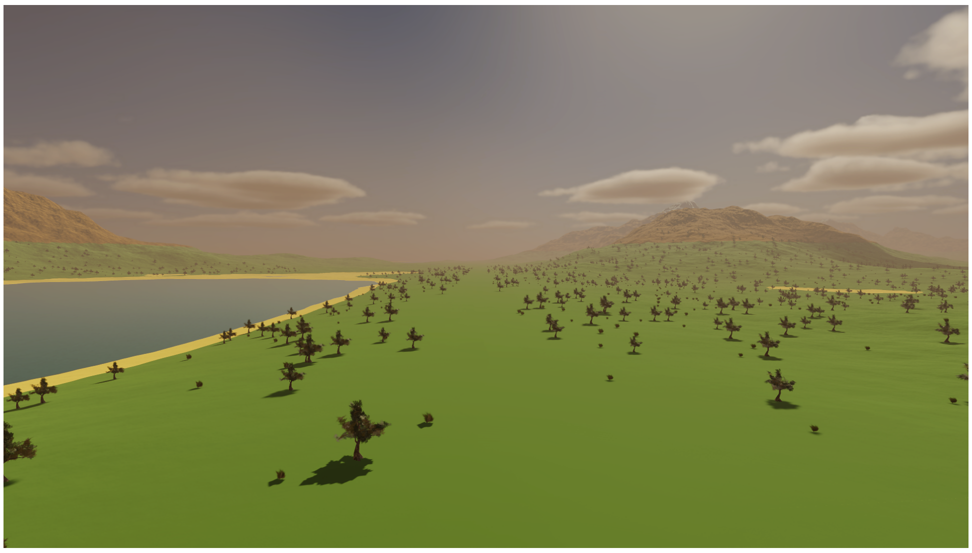


Figure 4.10: No vegetation on train track

Chapter 5
Conclusion

Even though I didn't manage to complete every goal I set for myself in this bachelor thesis, I'm still pleased with how it came out. The performance of the final product is acceptable in my eyes, I think the main goals the assignment set for the project have been fulfilled, and I've certainly learned a lot during development not just about procedural generation, but also about writing shaders and optimizing performance in Unity.

As I mentioned in the introduction, procedural generation is a widely used technique which I'll surely encounter very often in my professional life in the future. As such, this bachelor thesis was extremely valuable to me as the first (major) encounter with procedural generation, not to mention enjoyable to write, barring fighting some of Unity's peculiarities on the way.

I'd also like to take this opportunity to once again express my gratitude to the supervisor of this bachelor thesis, doc. Ing. Jiří Bittner, Ph.D. for his very helpful advice throughout the entire development process, for steering my focus on the important aspects of the project, and for giving me valuable feedback for writing this bachelor thesis after the software project was completed.

Bibliography

[1] Making maps with noise functions by Red Blob Games at https://www.redblobgames.com/maps/terrain-from-noise/

[2] Unity Documentation at https://docs.unity3d.com/Manual/index.html

[3] Unity Vector swizzle extension by Glaiel-Gamer at www.reddit.com/r/Unity3D/comments/2l5331/wrote_a_little_thing_that_adds_swizzling_to/

[4] Unity Mathematics library at https://docs.unity3d.com/Packages/com.unity.mathematics@1.2/manual/index.html

[5] Free bush model by Thunder at https://sketchfab.com/3d-models/bush-1e3321934c41424e9e2c02a24fd00aba

[6] Free lowpoly tree model by dionne at https://sketchfab.com/3d-models/lowpoly-tree-b562b2e9f029440c804b4b6d36ebe174

[7] Models of 3D Scenes for a Driving Simulator by Petr Brachaczek at https://dspace.cvut.cz/handle/10467/68427

[8] Procedural model generation from real maps by Jana Kejvalová at https://dspace.cvut.cz/handle/10467/80367

[9] Procedural generation of videogame environments by Jan Kutálek at https://dspace.cvut.cz/handle/10467/94735

[10] Creation of modular 3D assets for videogames by Alena Mikushina at https://dcgi.fel.cvut.cz/theses/2020/mikusale

[11] PERLIN NOISE in Unity - Procedural Generation Tutorial by Brackeys at https://www.youtube.com/watch?v=bG0uEXV6aHQ

[12] PROCEDURAL TERRAIN in Unity! - Mesh Generation by Brackeys at https://www.youtube.com/watch?v=64NblGkAabk

[13] MAKE YOUR GAME RUN SMOOTH - Unity LOD Tutorial by Brackeys at https://www.youtube.com/watch?v=ifNyVS2_6f8

[14] Getting Started with Compute Shaders in Unity by Game Dev Guide at https://www.youtube.com/watch?v=BrZ4pWwkpto

[15] 3D World Generation: Heightmap Tutorial by SimonDev at https://www.youtube.com/watch?v=hHGshzIXFWY

[16] 3D World Generation #2 (Perlin Noise) by SimonDev at https://www.youtube.com/watch?v=U9q-jM3-Phc

[17] What the Heck is Tessellation?! by Greg Salazar at https://www.youtube.com/watch?v=p_VpAMaxwpY

[18] Unity URP | Tessellation + Interactive Grass by MinionsArt at https://www.youtube.com/watch?v=aLzUJRQHLWc

[19] Tessellation and Displacement - Shader Graph Basics - Episode 41 by Ben Cloward at https://www.youtube.com/watch?v=ycJ434Lh21w