Procedural vegetation

TNM084 Procedural images

Anderas Engberg - anden561

January 14, 2022

Contents

1 Introduction

Inspired by Minecraft world generation and biome-specific vegetation this project aims to implement procedural generation and placement of trees. The project builds upon an earlier project that implemented real-time rendering of an infinite terrain. Therefore it is desirable that trees and other objects are placed in a deterministic but random manner. This allows the user to go back to a previously visited location where everything still looks the same as when they left.

Biomes are introduced to simulate real world changes in the terrain, with vegetation varying depending on its surrounding environment. For example, desert areas consist of smooth dunes and are filled with cacti while rainforest areas are filled with greenery and mountains. Project goals were specified as follows:

- Procedurally generated trees that vary with input parameters.
- Trees are placed procedurally in the terrain.
- Leaves are rendered with billboards using instancing

If there would be enough time an additional goal was set:

- Trees can be rendered with different level of detail (LOD) and billboards as the lowest level

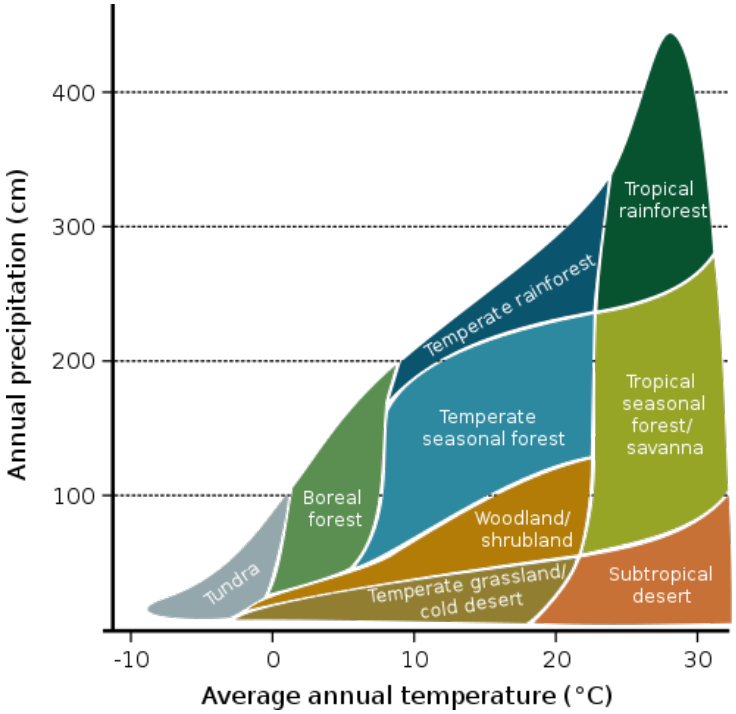


Figure 1: Whittaker diagram, by Navarras - `https://commons.wikimedia.org/w/index.php?curid=61120531`.

2 Implementation

2.1 Tools and libraries

The project was implemented with OpenGL 3.3 and the additional libraries GLM, GLAD and GLFW. GLM is a mathematical library which allows for vector and matrix operations. GLFW is used to setup the OpenGL environment and GLAD loads all OpenGL specific functions.

To create procedural vegetation a modified version of GLUGG was used, a lightweight library built by Ingemar Ragnemalm to replicate the old OpenGL immediate mode [9]. It was ported to work with GLM, GLAD, GLFW and updated to use the C++ std::stack.

Loading textures is done using the stb single file image loader by Sean Barrett [1] which support most popular image formats. Lastly Poisson point generation is done using the api provided by Sergey Kosarevsky [6].

2.2 Procedural biomes

Biomes are created by trying to simulate real-world processes. Here moisture and temperature is combined; from Whittaker diagram in Figure 1 we can determine what type of biome emerges from the given heat and moisture values.

Because we are going to use many different types of noise, both weather and environment. A noise map class was created that takes FBM parameters, i.e. number of octaves, amplitude, frequency and seed as input parameters. The noise map can then be evaluated using simplex noise or Perlin noise at a certain location (x,z) in space.

We generate a temperature map and moisture map from low-frequency simplex noise maps. The noise

values are remapped from the normal range [-1,1] to the range [0,1] to simplify later usage. Furthermore, the weather maps are visualized as seen in Figure 2 where red represent hotter areas and blue are colder areas, brown indicate dry areas and blue wet areas for the heat map and moisture map respectively. A simplified version of Whittaker diagram was used with only four different biomes shown in figure 3a. Similar to Jon Gallant solution the biome table is stored as a two-dimensional array allowing for easy and fast lookup using enums [4]. For every vertex point in the grid a moisture and temperature value is calculated which then determines the biome at that location shown in Figure 3b.

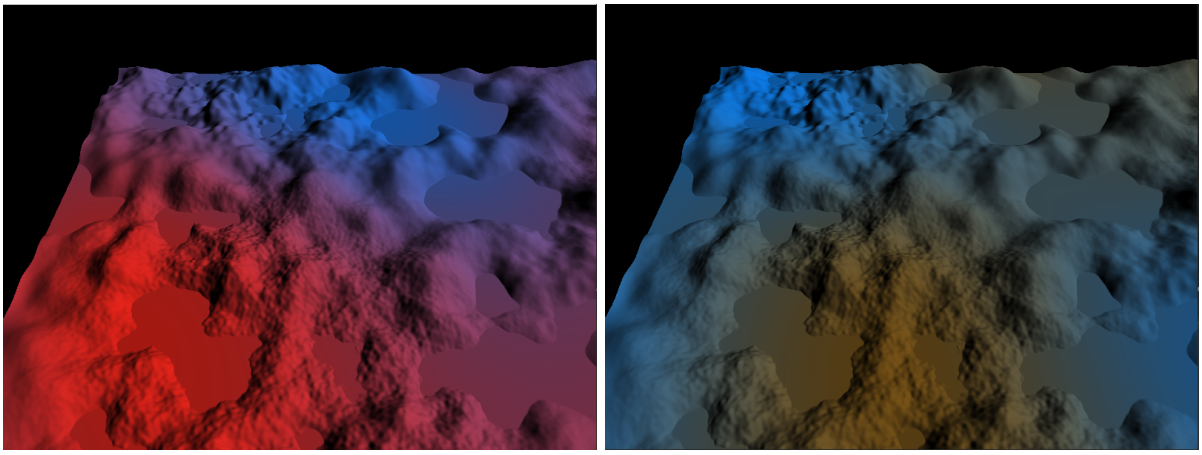
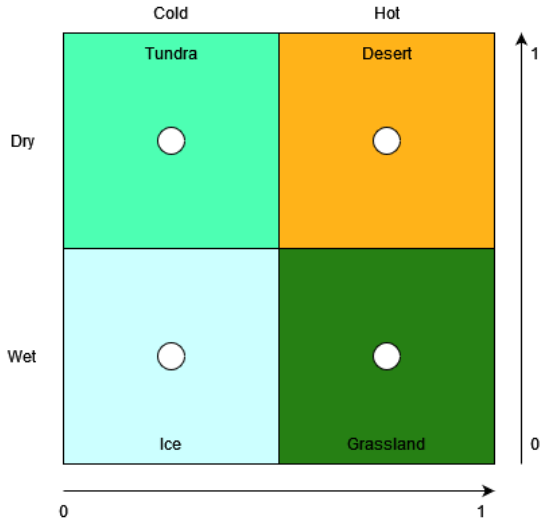
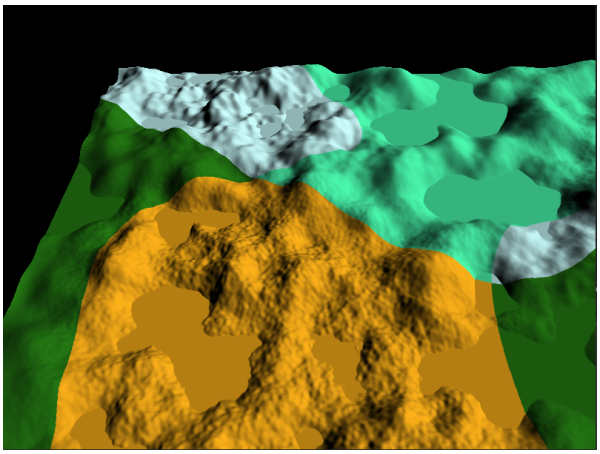


Figure 2: Heat map and moisture map visualized in the terrain.



(a)



(b)

Figure 3: Simplified Whittaker biome diagram (a) and result of biomes determined by heat and moisture map (b).

Each biome type has its own rules for how the terrain should look like. For example desert is mostly smooth with low varying hills and valleys while woodland can reach higher altitudes and vary more rapidly. The terrain elevation at a specific point is given by the biomes terrain map which is a Perlin noise map where the FBM parameters are defined by the rules. However, computing areas of the terrain differently depending on the type of biome it belongs to causes some issues and artifacts that must be resolved. There can be large differences for the final height value between two nearby points that belong to different biomes. This is caused by the fact that we evaluate two completely different noise maps at similar locations. One such artifact is shown in Figure 4 where we can see large height differences at the border between the two biomes. To avoid such artifacts biomes must blend where they meet. This proved to be a really difficult problem to solve in a way that gives good looking results. Different

blending solutions have been proposed in similar projects such as computing a weight from the barycentric coordinates in a Voroni diagram, measuring the distance in world space or distances in temperature and moisture space [2][5][7].

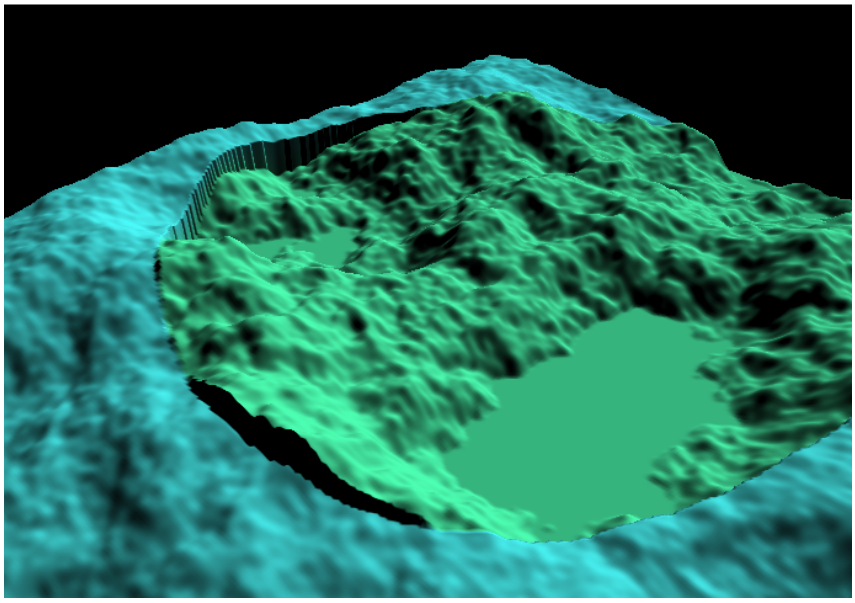


Figure 4: Biomes create artifacts at neighboring biome borders.

The height value is a combination of a baseline height added together with the biome specific height. The baseline is a low frequency noise map and can be thought of as the elevation coming from underlying bedrock or tectonic plates. The baseline gives a rough elevation detail and the biome adds smaller details such as mountains and valleys. The idea is to compute a weighted average for the final position of a vertex on the terrain. The biome table is extended with a midpoint c_i for each biome shown as a white dot in Figure 3a. For a given point p_0 in Figure 5 the distance $d_i = dist(p_0, c_i)$ to all biome midpoints is calculated. A weight is given to each biome as $w_i = 1/d_i^3$, lastly a position p_i is calculated using the biome terrain map and the final position P is given by

$$P = \frac{\sum_{i=1}^{n} w_i * p_i}{\sum_{i=1}^{n} w_i} + p_b \tag{1}$$

where p_b is the baseline position evaluated from the noise map and n being the number of biomes.

2.3 Procedural tree generation

Procedural trees can be generated using a L system with a set of replacement rules and a starting condition [8]. Another approach and the one taken in this project is to use recursive algorithms that define some form of rules. It was important that the trees look and feel somewhat realistic, however, modeling in OpenGL may not be the best environment suitable for this type of work.

2.3.1 Generation

We don't want all trees to look the same, therefore we introduce randomness and other data into the generation. We let the moisture map and temperature map control the behaviour of the tree, where the temperature affects the size of the tree and the moisture influence the number of branches a tree has.

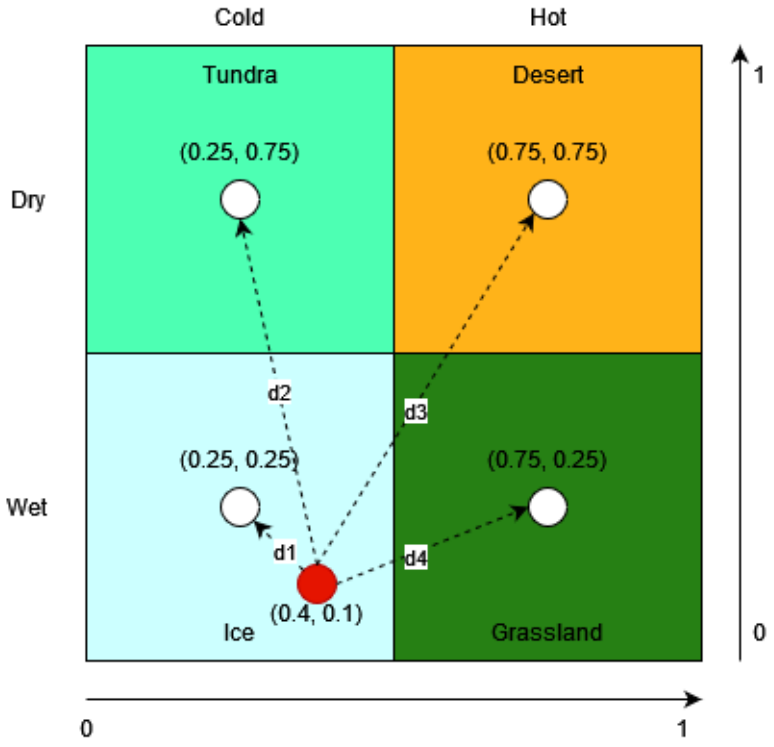


Figure 5: A given point in heat/moisture space computing the distance to each biome

Furthermore the biome type can determine the type of vegetation that grows for example we have cacti in the dessert, birch trees in the forest and "naked" trees without leafs in winter biomes.

The different trees and their generators are implemented in a hierarchical class system derived from a super class called tree. The factory pattern together with a tree generator is used to determine at run time what type of tree to spawn. A simplified pseudo code example can be seen in listing 1, full details can be viewed in the Tree.h on the GitHub repository [3].

```
Create body of tree;
Add a random number of branches;

for each branch
    rotate arbitrary around y and z axis
    create cylinder
    add new random number of branches
end

at max depth of tree create leafs
```

Listing 1: Pseudo code generating spruce trees

2.3.2 Procedural placement

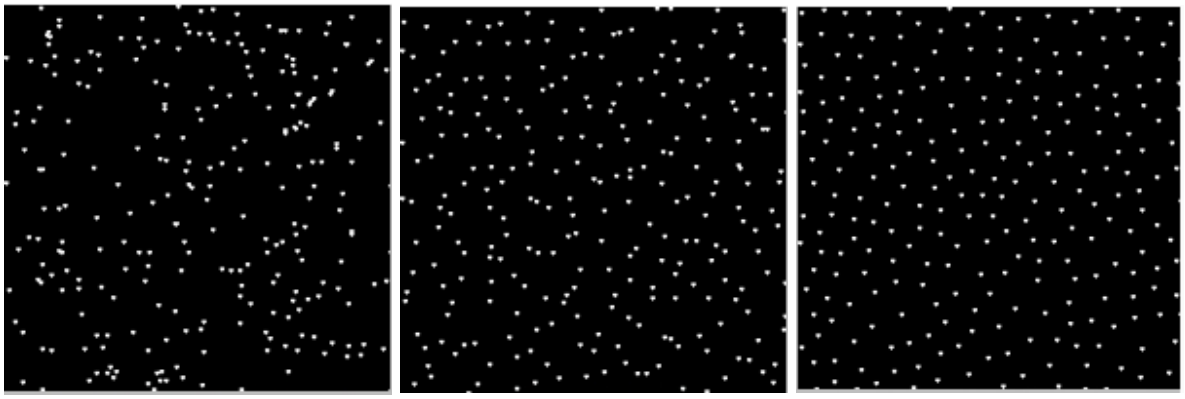
It was important that the trees are placed randomly however it needed to be deterministic so when the player walks back to a place they have previously visited the trees are in the same position, although they can look different. To achieve this effect random spawn points were generated at initialization and then mapped over the terrain grid. Three methods were evaluated, uniform random points, square jitter grid and Poisson disk sampling.

The first method randomly generated points in the range [0 - 1] for both x and z axis using the standard library random function. The problem with this method is that it can create very uneven placements for

a small number of trees, some values will be very sparsely placed while other will be too tightly packed which can be seen in Figure 6a. Moreover, there is really no way to control how compact the points are placed and is therefore not suited for vegetation placement.

The second method uses a grid of fixed sizes which controls how far apart the points are placed. One can either take a random position in each cell or add noise to the grid points which breaks the ugly grid structure. This method looks much better than uniform random values, however, jittered grids can still have gaps and clumps as seen in Figure 6b.

Lastly Poisson disk sampling works by generating random points near existing ones, all points must fulfill a minimum distance requirement to be placed [10]. The advantages with this method is that we can control how densely the points are placed. Furthermore, as we can see in Figure 6c there is no visible grid structure of the points. These properties makes the Poisson sampling very suitable for tree placement.



(a) Random points generated (b) Jittered grid (c) Poisson disk sampling

Figure 6: Images taken from: `http://devmag.org.za/2009/05/03/poisson-disk-sampling/` [Online; accesed 10-01-2022]

The spawn points generated from the Poisson sampling are mapped to world coordinates over each terrain chunk. The world position is then validated using a set of rules, if any of the rules are false the position is discarded and no tree is placed. Three validation rules are defined:

- Trees cannot be created in water, ie. if the position is below a certain threshold.

- Trees cannot grow at any inclination therefore the slope at the position is computed. Because it would be very expensive to find the nearest triangle at a position (x,y,z) to access the normal. Instead the normal is estimated using the cross method, ie. take a small offset step in the north, south, west and east direction and compute the normal as the cross product between north - south and west - east. If the angle between the normal and the up vector is less than 40 degrees a tree can be placed.

- Lastly we want to avoid the pattern that could arise from using the same points in every terrain chunk. This is done by evaluating a high frequency simplex noise at the spawn position (x,y,z), all values below zero are discarded. Since simplex noise has an average value of zero this method will remove roughly 50% of all points which must be taken into consideration when populating the terrain. On a positive note the same trees will exist every time a specific terrain chunk is generated.

The validation is visualized in Figure 7 where green areas are possible spawn locations and red invalid ones.

2.3.3 Rendering leaves

Leaves at the end of the tree can be created using GLUGG and another set of rules or by loading a new model made for example in Blender. The problem with this approach is the number of leaves in the

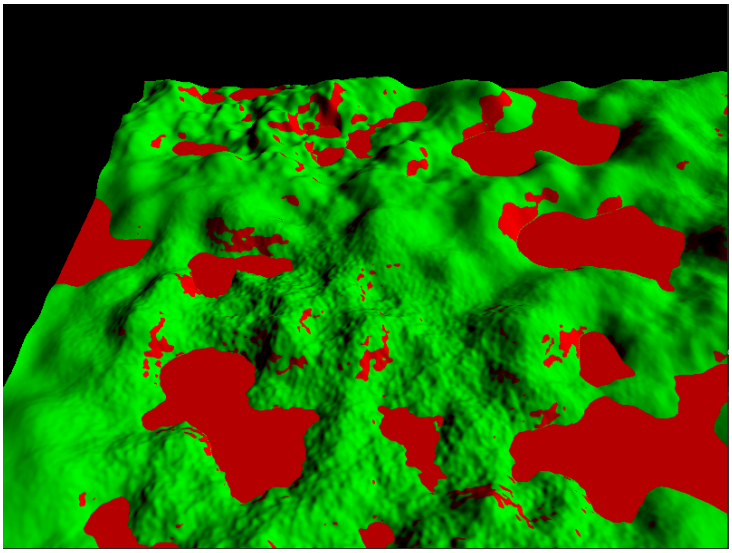


Figure 7: Tree validation of the terrain

scene. A tree may require hundreds of leaves and depending on the level of detail of the leaf model, this can result in many millions of extra polygons and can become quite expensive. A faster and cheaper method is to use textured billboards this result in one texture plus four vertices to render. However, as we will see making thousands of draw calls for each separate leaf is very expensive. The solution is to use instancing where we only load one model, make one draw call but render all the leaves at once each with its own model matrix.

A quad class holds a texture and a list of unique transformation matrices that describes the world position of each leaf. The code is based on the works of Joey De Vries [11]. One could use a uniform array variable in the shader and use the instance_id to retrieve the correct matrix, there is however a limit to how much data that can be uploaded to the shader this way. Instead we will use instanced arrays which are defined as a vertex attribute, we will upload the matrices to the shader this way. We can circumvent the maximum allowed data for a vertex attribute which equals a vec4 by using four attributes, since a mat4 is basically 4 vec4s. The code is given in listing 2, where the last call glVertexAttribDivisor tells OpenGL when to update the content of the attribute, ie. one per instance.

Two quads are drawn at the end branches of a tree rotated 90 degrees apart. The rotation reduces the feeling of flat leaves, the result can be seen in Figure 8

```
...
glBindBuffer(GL_ARRAY_BUFFER, matrixVBO);
glBufferData(GL_ARRAY_BUFFER, modelMatrices.size() * sizeof(glm::mat4),
    &modelMatrices[0], GL_STATIC_DRAW);

std::size_t vec4Size = sizeof(glm::vec4);
glEnableVertexAttribArray(3);
glVertexAttribPointer(3, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)0);
glEnableVertexAttribArray(4);
glVertexAttribPointer(4, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(1 * vec4Size));
glEnableVertexAttribArray(5);
glVertexAttribPointer(5, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(2 * vec4Size));
glEnableVertexAttribArray(6);
glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, 4 * vec4Size, (void*)(3 * vec4Size));

glVertexAttribDivisor(3, 1);
glVertexAttribDivisor(4, 1);
glVertexAttribDivisor(5, 1);
glVertexAttribDivisor(6, 1);
...
```

Listing 2: Uploading mat4 to a shader using vertex attributes

3 Results

A rendering test was carried out to compare the effect of instance rendering. The test was performed on a windows computer with Nvidia GTX 980-ti graphics card, Intel i5 4690k processor and 16GB 1600MHz of ram. Without instancing one billboard leaf model was redrawn with a new matrix uploaded to the shader each time. With instance rendering one draw call was made for one billboard with all the data uploaded as explained in section 2.3.3. The result can be seen in table 1.

Number of billboards	fps without instancing	fps with instancing
0	144	144
25 000	90-100	144
60 000	39-42	144
100 000	25	144
200 000	12	140-144
500 000	4-6	115-130
1 000 000	1-2	65-70

Table 1: Rendering a scene without vs with instanced rendering

The result of procedural generated trees with instanced billboards as leaves can be seen in Figure 8. The variation of trees in different biomes is shown in Figure 9 and Figure 10 shows the result of placing trees using Poisson disk sampling.

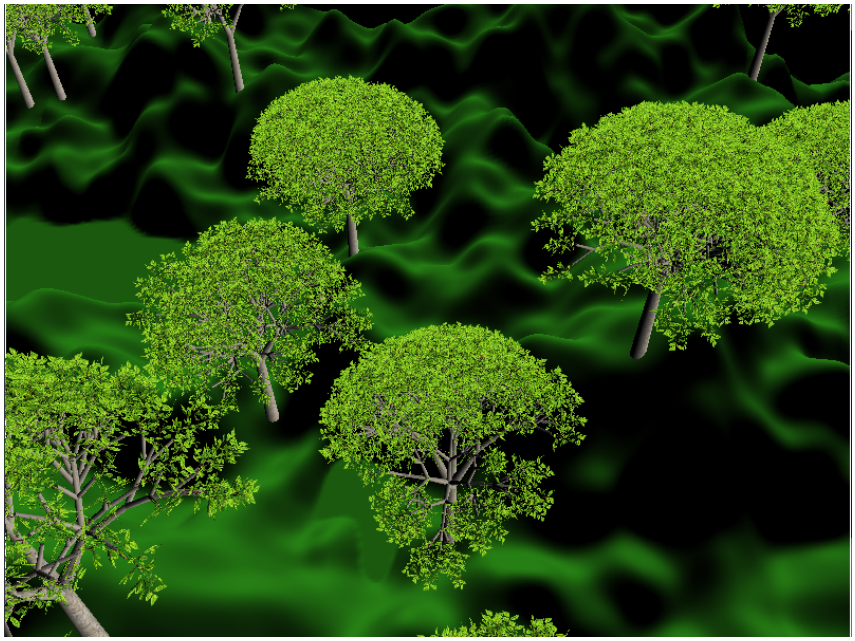


Figure 8: Resulting tree with billboard leaves

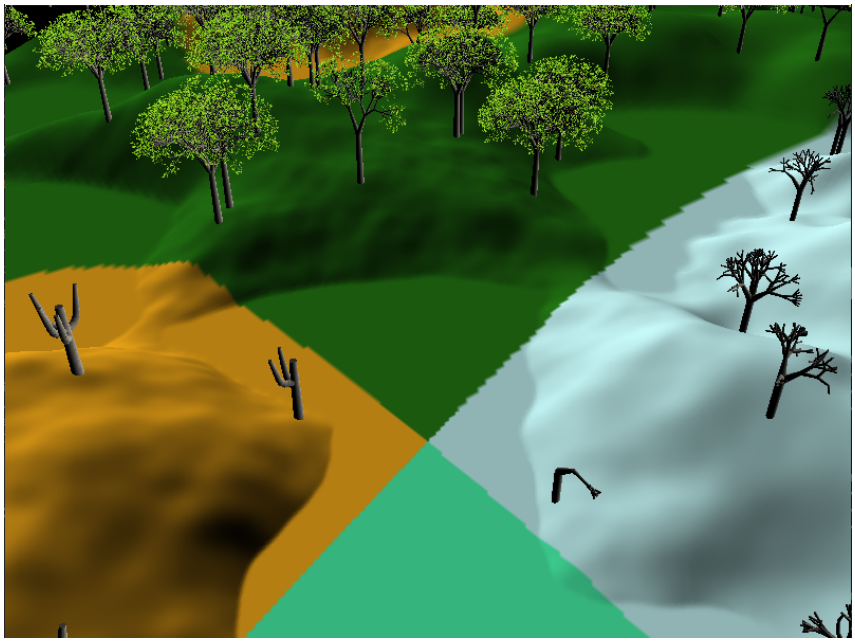


Figure 9: Procedural trees varying over different biomes.

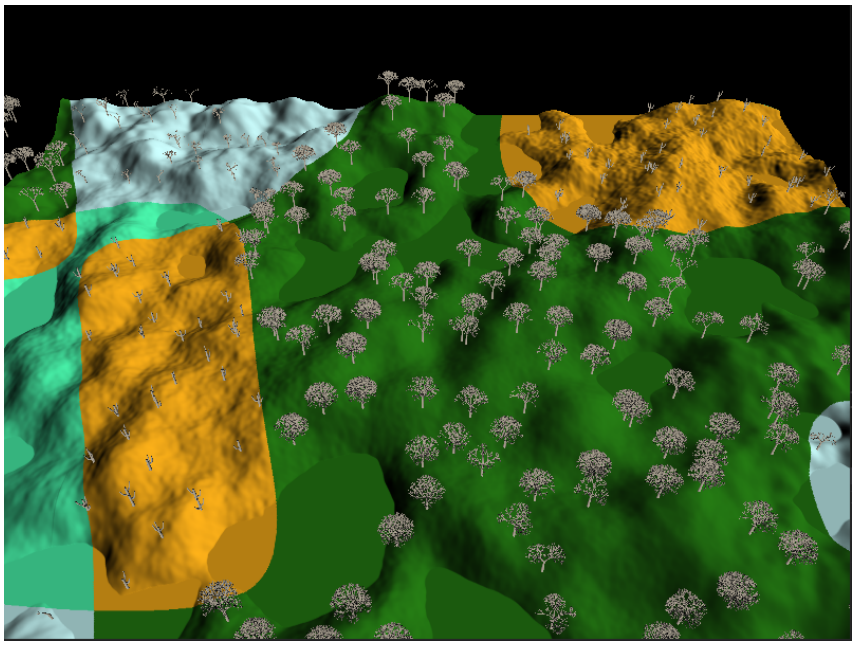


Figure 10: Result from Poisson tree placement with leaves turned off for better visibility.

4 Discussion

Even though only one set of Poisson points are used in the program to spawn trees at given locations. All underlying structure largely disappear from the validation step and its very difficult to find any form of periodic repetition of the tree placement as can be seen in Figure 10. This makes the Poisson generation a very suitable option for placing objects procedurally in a scene, furthermore, one can generate new points for each type of biome where the radius depends on parameters such as moisture and temperature. This could lead to more realistic biomes for example in a pine wood the points would be placed much closer together than for example in the desert.

From the result in table 1 it is clear how effective instancing can be when a model has to be drawn multiple times. Already at 60 000 rendered billboards the frame rate begin to drop below an acceptable level without instancing. But with instancing we can have upwards of a million quads or even more before the frame rate is noticeably affected. At a distance there will be no noticeable difference between a 3D model and a textured quad leaf, it is only at a close range that you can see that it has been faked and the flatness of the billboard appears. Considering the improved rendering times and how easy billboards are introduced to the program this is a small price to pay.

LOD for trees has not been implemented, however, I have some ideas on how that would be achieved. The way the LOD system works in the project is by using discrete LOD-models that are rendered at different distances. This means that each tree must contain references to a set of lower polygon models. Due to the sheer number of trees in the world this can quickly become inefficient. Instead of generating a unique procedural tree every time, one could pre-generate a set of trees. This has the benefit of reducing memory consumption quite a lot as we now have a finite number of trees to generate LODs for. Furthermore, one could create a procedural tree foundation and then manually tweak the tree as desired in a 3D computer graphics software. With a limited number of tree models to choose from we could see some benefit in the frame rate of using instancing to place the trees as well, although exactly how this would work with the LOD system is not clear. Choosing the LOD for a tree would be similar to how a terrain chunk is chosen; by computing the distance to the camera. However, a faster method would be to let the LOD of the terrain chunk itself determine the LOD for all trees placed there. As most trees in a chunk will have approximately the same distance to the camera we can avoid many unnecessary computations. Trees furthest away can be replaced with a textured billboard to save polygon count and further reduce rendering times.

References

[1] Sean Barrett. *stb image reader*. https://github.com/nothings/stb. [Online; accessed 10-01-2022]. 2017.

[2] DMGregory. *Generate biomes perlin noise*. https://gamedev.stackexchange.com/a/186197. [Online; accessed 04-01-2022]. 2020.

[3] Andreas Engberg. *ProceduralGeneratedTrees*. Version 1.0.0. Dec. 2022. URL: https://github.com/engbergandreas/ProceduralGeneratedTrees.

[4] Jon Gallant. *Procedurally Generating Wrapping World Maps in Unity C#*. www.jgallant.com/procedurally-generating-wrapping-world-maps-in-unity-csharp-part-4/. [Online; accessed 04-01-2022]. 2016.

[5] Bilal Himite. *Replicating Minecraft World Generation in Python*. https://towardsdatascience.com/replicating-minecraft-world-generation-in-python-1b491bc9b9a4. [Online; accessed 04-01-2022]. 2021.

[6] Sergey Kosarevsky. *Poisson Disk Generator*. https://github.com/corporateshark/poisson-disk-generator. [Online; accessed 10-01-2022]. 2014-2021.

[7] Colby Newman. *Generating complex, multi-biome procedural terrain with Simplex noise in PSWG*. https://parzivail.com/procedural-terrain-generaion/. [Online; accessed 04-01-2022]. 2019.

[8] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer Science & Business Media, 2012.

[9] Ingemar Ragnemalm. *GLUGG - OpenGL Utilities for Geometry Generation*. https://computergraphics.se/demopage/glugg.html. [Online; accessed 10-01-2022]. 2013-2021.

[10] Herman Tulleken. *Poisson Disk Sampling*. http://devmag.org.za/2009/05/03/poisson-disk-sampling/. [Online; accessed 08-01-2022]. 2009.

[11] Joey de Vries. *Learn OpenGL*. https://learnopengl.com/. [Online; accessed 27-12-2021]. 2014.