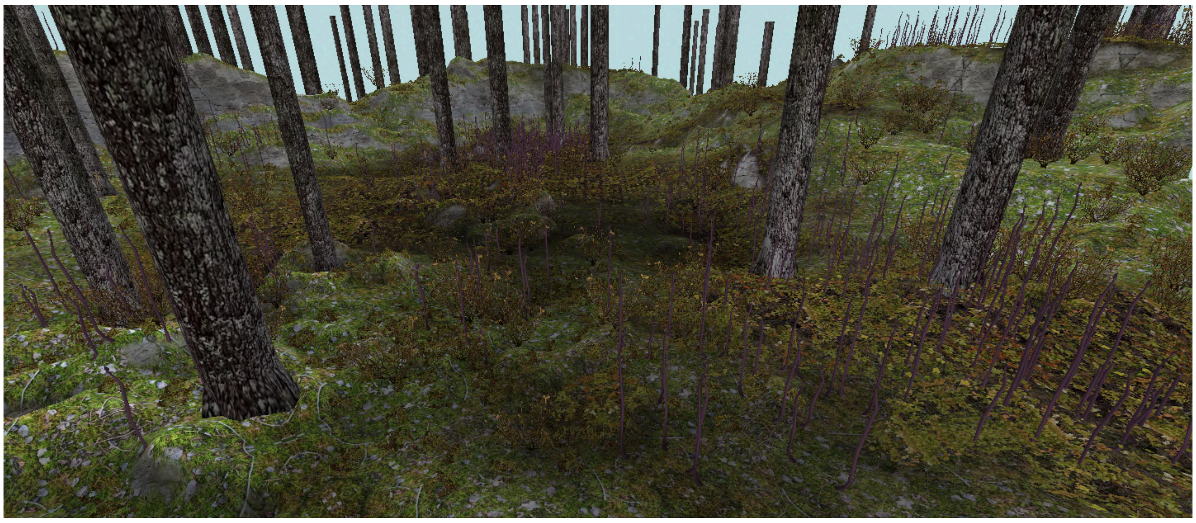
Procedural Undergrowth

This demo creates procedural but deterministic renderings of the flora underneath a forest's canopy for use in video games. My implementation proposes one method of rendering the ground which uses multiple height maps and shader discards and combines this with traditional instanced rendering of foliage.



Introduction

In many games I've played the forest areas seem empty and lifeless compared to the lush and detailed real life forests. While this is certainly partly due to intentionally sparsely placed foliage for optimization reasons, I also believe that a large part is in the lacking undergrowth and too plain forest ground which I tackle in this demo.

The project work consists roughly of 3 parts: In the abstraction subsection I talk about code structure and abstractions to work with OpenGL. In the terrain and entities subsection I explain how I render the ground and the model based foliage respectively. I finish with a discussion of design choices and a conclusion with further work.

Implementation

Everything is implemented in Rust 1.76 and OpenGL 4.1 and its full source code is available[1]. When the program is started a deterministic scene loads up and other scenes can be shown with the left/right arrow keys. All scenes share an artist created scene map which determines some of the larger structure. While the program is running, the otherwise static camera bobs around to create parallax depth clues and liveliness.

Abstraction

My main file `main.rs` the `renderer.rs` file are based on the windowing library `glutin's` example[2], since the separate render thread of the gloom-rs template[3] doesn't

[1] https://github.com/Indeximal/UnderstoryScene
[2] https://github.com/rust-windowing/glutin/blob/e1bf1e22a3e2f0e3dc4213f85c10f33049ce8d77/glutin_examples/examples/window.rs
[3] https://github.com/pbsds/gloom-rs

work on my mac. They are responsible for setting up the windows, viewport, resizing, clearing before every frame, getting keyboard events and rerendering when the scene is changed. The `scene.rs` file is meant to be a one stop shop to change the visuals, including creating the terrain and entities, but also determining the camera position, direction and background color.

The next set of files are highly needed since OpenGL is idiomatically very different from rust, so having a robust abstraction makes working with it way nicer and less error prone.

`texture.rs` has a structure `Texture` which only holds the texture id, but all the texture relevant calls go through its methods and the constructor function provides a generic, but safe interface to load textures from RAM onto the GPU.

The `ShaderBuilder` and `Shader` structs provide a similar abstraction of a shader program, but were improved from the gloom-rs one to not require unsafe function signatures and properly handle null terminated error strings.

The `VAO` in `mesh.rs` holds both the vao id as well as the ids of all associated vbos and are managed by two specialized usecases: One of them has positions, normal and uv attributes while the other one additionally has a model and normal transformation matrix for each instance. More in the entities subsection.

Lastly the `assets.rs` file is responsible for loading all the resources from disk which don't change with the scene, which saves about 250 ms of 600 ms loading time. Assets are stored behind reference counted smart pointers, which the entities clone. Importantly, the cleanup of old data is handled by the rust ownership system since the destructors of these abstractions also delete the resources on the GPU.

All in all I had a lot of fun creating idiomatic rust abstractions and it was really easy to use afterwards. However, they are still very tailored to my use case and creating general purpose abstractions is left for public libraries, such as wgpu[4].

Terrain

Most of the screen space is taken up by the terrain. At its core the terrain is a 256 by 256 quad grid mesh whose heights are modified by the vertex shader based on a height texture. It is then shaded with axis projected mirror-repeat-tiled textures.

In my observations of the forests here in Trøndelag I noticed how a lot of the structure of the ground is given by soil covered rocks. Therefore my goal for the height map was recreating something resembling rocks, which turned out to be fairly difficult. For creating the height map I composed procedural noise expressions from the noise[5] library and self implemented expressions. After many iterations, the base rock imitation landed on a manhattan worley noise, which gives regular ridges, masked with fractal value noise above a certain threshold to hide the regularity and create flat patches. This is summed fractally from multiple frequencies and additionally a simple value noise and the red component of the scene map are added to provide details and artist control respectively. The noise function is then both sampled into an OpenGL texture and stored CPU side to appropriately place the entities on the terrain.

[4] https://wgpu.rs
[5] https://docs.rs/noise/latest/noise/

The scene map is created such that the backdrop is elevated and occludes the nothingness behind. To maximize the onscreen portion of the quad, the camera is standing near one corner and looking towards the opposite corner.

Additionally to the height displacement the vertex shader also calculates the normals using finite differences. These normals are used in the fragment shader to calculate weights by taking each component to the power of 8. For each axis a different texture is then sampled based on the other two world space coordinates and the weighted average is calculated. For instance `weights.z / total_weight * texture(terrain_albedo_xy, world_position.xy, -1.5)` is the contribution of the top down texture. Mip mapping is used to hide the noisy artifacts that stem from high resolution textures. Note how the bias -1.5 is used in every texture lookup to avoid visible blurring. The textures are 1024 or 512 pixels sidelength covering approximately half a meter, are configured to mirror repeat tile, and are cropped and edited from photos I took specifically for this purpose.

Each fragment is shaded using a 0.5 weighted ambient term and a 0.5 weighted diffuse term with a principal light direction. This corresponds to being lit by a hemisphere[6], which models the canopy covering most direct sunlight and indirect light coming from all directions above. I hypothesize that this scene would greatly benefit from SSAO, which with deferred shading is out of the scope of this demo. However, I added an additional darkening term based on height to make sensing the elevation easier.

So far nothing special, two additional features make this more interesting. Firstly, I am using two top down textures variants and blending between them based on another fractal value noise map. The first, more frequent, variant is moss while the second one is needle covered soil. This breaks some of the tiling artifacts and creates more diverse regions.

Secondly, I stack another layer onto the ground which uses the same shader and quad, but a height map that has additional noise of 2 to 10 cm added and partially transparent textures. The shader discards all fragments with an alpha value below a threshold while the others are rendered fully opaque, which avoids blending complications and allows for intersecting transparent geometry. This layer only has a manually alpha masked berry bush leaves texture above the soil. The idea here is that this way parallax effects and occlusion of the roots of other entities can highlight depth and softness of the ground. In practice this is barely visible from the almost stationary viewpoint and it introduces some high frequency noise as pixels can just appear and disappear.

All together the gradient based texturing, the layers, the elevation darkeing and Phong inspired shading give the viewer a solid grasp of how the terrain looks three dimensionally and creates a nice ground for the entities to stand on.

Entities

On top of the ground there are four kinds of foliage I spawned: Trees, saplings, shrubs and bushes. All use the same technology in `foliage.rs` and are differentiated only by the `ShrubEntitiesBuilder's` settings of density, looks and scale randomization limits.

[6] If the normal and light direction align, the dot product is 1 and the diffuse and ambient term should add to 1. If they are perpendicular, i.e. dot product is 0 then they should still be light by a quarter sphere, which is exactly half the irradiance. Therefore the ambient weight has to be 0.5 and so does the diffuse.

To generate the positions of the entities, first a low detail value noise function each is created and multiplied by the green component of the scene map for small foliage and the blue one for trees. This lets the artist control how overgrown a region is. The function is not loaded into a texture, instead `generate_points_on_distribution` samples the function at every chunk of a 100 sided grid, multiples the value with the area and density to get an expected number of entities in this chunk and draws from a random variable with this expectation value. The drawn number of positions are then uniformly distributed within the chunk. However, since the noise function has high variation in its average value, I have found the density to be insufficient to control the number of trees. Therefore a limit N can be set such that after drawing samples, at most N random samples are returned. In addition to the positions, a scale value, an height scale value and a rotation value are uniformly drawn, based on configuration.

The struct `InstancedMeshesVAO` is used to render many identical triangle mesh models, but at different positions, making it vastly more efficient than invoking a draw call for every entity. It makes use of the `glDrawElementsInstanced` and `glVertexAttrib-Divisor` functions to have position, index, normal and uv vertex buffer objects be associated with triangles, but the model- and normal transformation matrices attributes associated which each instance. In the shader the model matrix can be accessed like any other attribute: `layout(location = 8) in mat4 model_mat`. However, when creat-ing the buffer, an attribute pointer has to be defined for every column, as the max size of an attribute is 4 components. All vertex buffers are filled for every scene reload, which could be optimized, but it only takes 2-4ms per entity type.

The models are loaded from obj files exported from blender. I found creating the models myself to be quicker and more satisfying than searching for fitting license-free models online. A had some prior experience with blender years ago and was astonished that in 30 minutes I had managed to create my first sapling model. It needed to be further simplified afterwards, as with 200 entities, each 120 triangles, my mac struggled. This is probably due to many small triangles, for which the GPU is not optimized [7]. Now the sapling and tree each have about 45 triangles, and the bushes and shrubs work with 6 and 3 intersecting triangles and transparent textures respectively. They are UV mapped to man- ually alpha masked pictures I took.

While the foliage is conceptually very simple, with the randomization a lot of instances can be spawned (totalling 1000 to 2000) to create a lot of detail and it looks very appealing.

Discussion

The project had its ups and downs. I started with a basic terrain, which had the problem of very obvious tiling artifacts. I fixed this by sampling the texture twice, once by rotating the coordinates first. However, I later reverted this as it introduced a blurring effect which made it look very bland. I also learned to use less blandly colored textures by turning up the saturation.

[7] When Optimisations Work, But for the Wrong Reasons by Simon Dev, https://youtu.be/hf27qsQPRLQ

As needed I created the rust OpenGL abstractions with help from my experience of many years sporadically watching game dev youtubers such as Sebastian Lague[8] or ThinMatrix[9] and memories of blog articles such as the GTA V graphics study[10]. For specific questions I either looked up the OpenGL documentation and the opengl-tutorial series or I asked GitHub Copilot's chat feature. GitHub Copilot was also useful in autocompleting repetitive tasks and filling in OpenGL function calls, as well as translating C++ examples found online or from the exercises to Rust. However, when asking it to make refactoring choices, implement algorithms or write documentation it was mostly disappointing. I estimate less than 30% of the code was touched by copilot and less than 5% was written by it.

A lot of the ideas of what to do next, what is important and coming up with techniques and mechanisms came while studying the forest during hikes. For shading I often only applied the changes to one half of the screen to have a before and after comparison. The initial proposal also mentioned mesh generation and geometry shader, but I soon turned away from these ideas for this demo, as I do not deal with any level of detail, which means the geometry is static and can be pre calculated or traditionally modeled.

There were two major tradeoffs I discovered along the way. Firstly, I had to decide where to use existing libraries and where to implement the code myself. For noise I decided to use a library to not have to deal with hashing and interpolating, but used my own logic to generate the samples from the noise function and still read a lot of internal code.

Secondly, there was some tension between how much control an artist has and how resilient it should be with suboptimal scene maps. For example the height should be continuous as it is directly used, which requires more work for the artist, but also allows very granular and steep changes. Also no parameters are currently loaded from a configuration file since they are often without intuitive meaning and hard to change without knowing the inner workings anyway. Therefore I only partially achieved my goal of letting the scene be artist controlled.

Conclusion

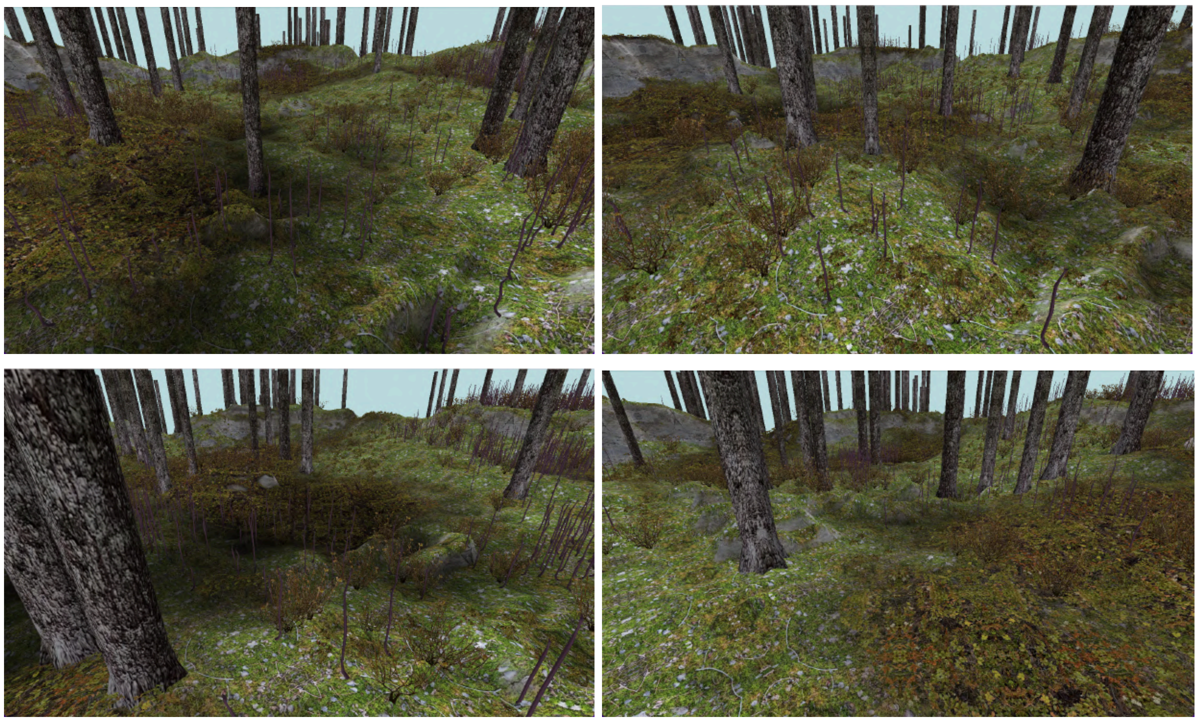
In order to make this demo into a paper, a lot of benchmarking and research of prior work would be needed. However, for what this is meant to be, which is eye-candy, this project turned out very satisfactory. It uses tried and tested height displacement maps for the terrain and instanced rendering for foliage as well as more experimental shader discard based transparency. This shows despite my initial assumption that complicated methods are not required for interesting scenes and instead composing simple and effective techniques is where both the programmer and GPU shine.

In further work I suggest implementing deferred shading and SSAO, since the local illumination is not sufficient to capture the rich light interactions in a detailed environment. Additionally anti aliasing such as MSAA should be implemented to ameliorate the high frequency visual clutter. For a game application with a moveable character one should look into introducing levels of detail for the techniques described here or refer to existing virtual LOD systems such as Nanite in UE5[11].

[8] https://www.youtube.com/@SebastianLague
[9] https://www.youtube.com/@ThinMatrix
[10] http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/
[11] https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf

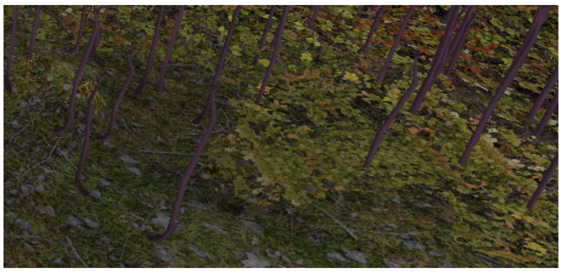
Appendix: Figures

Selection of scenes generated

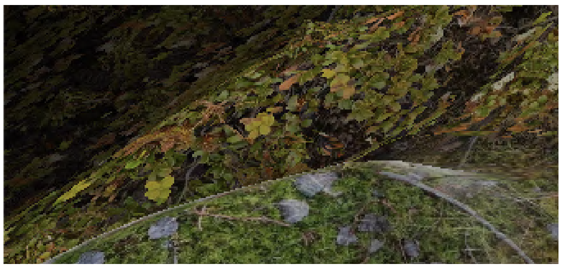


Special highlights

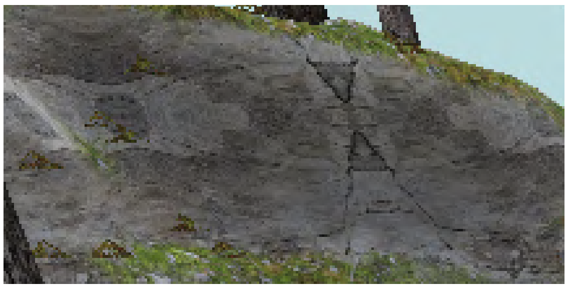
- Roots of the sapling hidden by the bushes.



- Shading based on height

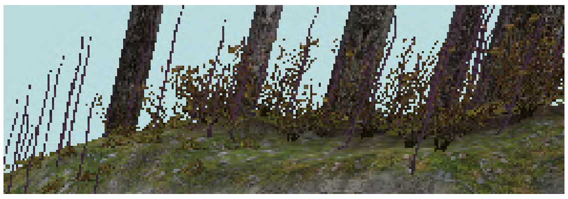


- Tiling artifacts

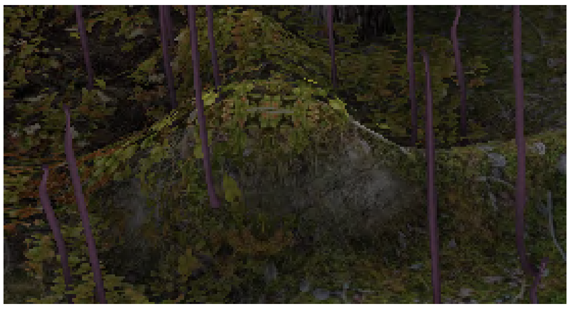


- Missing canopy in the background

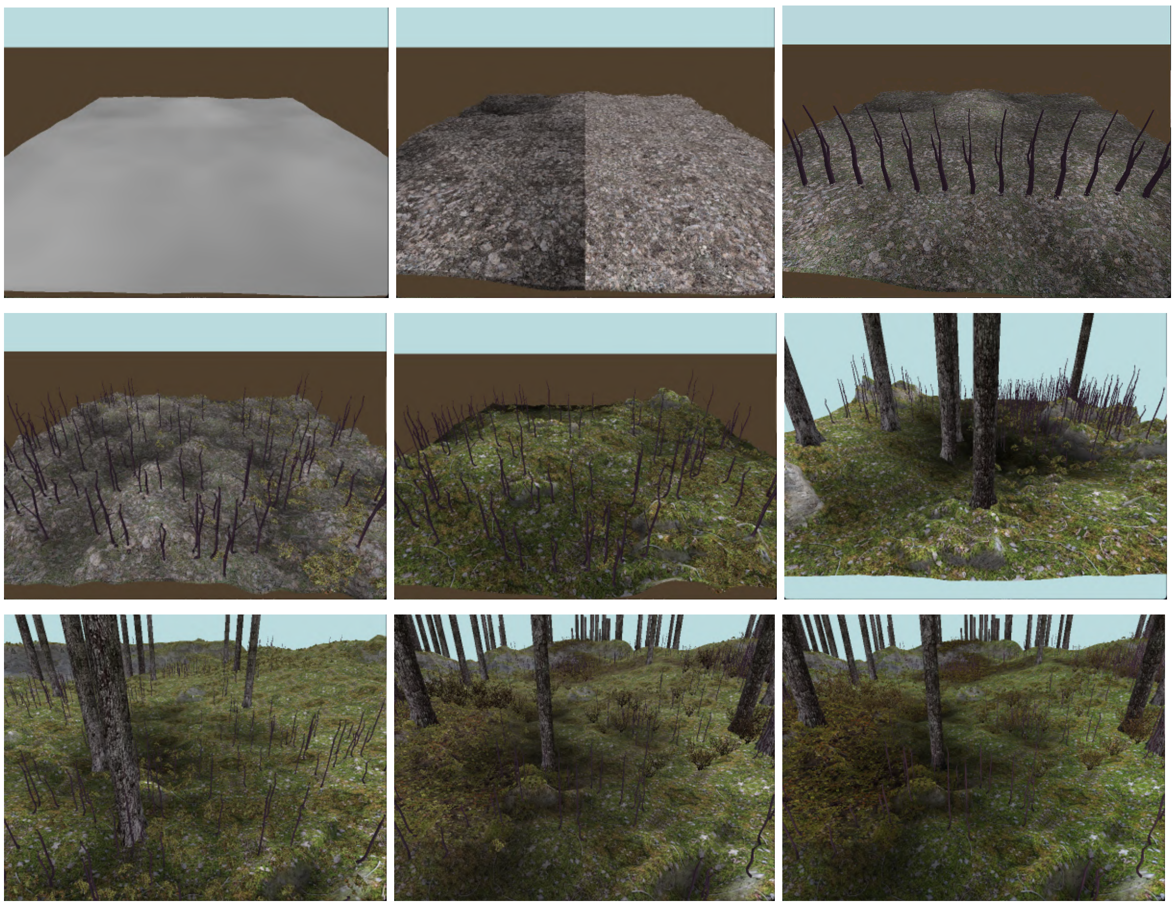
- High frequency noise in distant transparent textures



- Second layer wrapping around rock

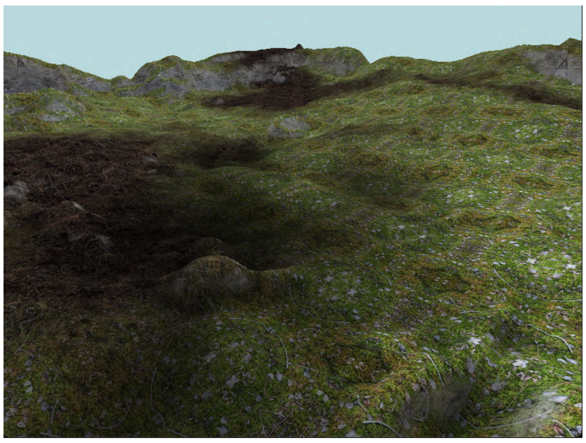


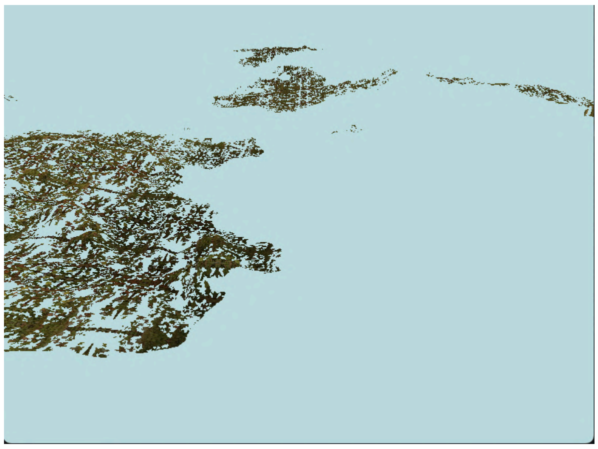
Selection of screenshots during the development history

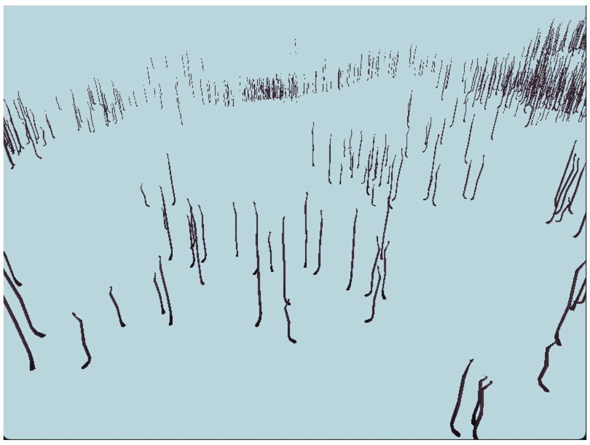


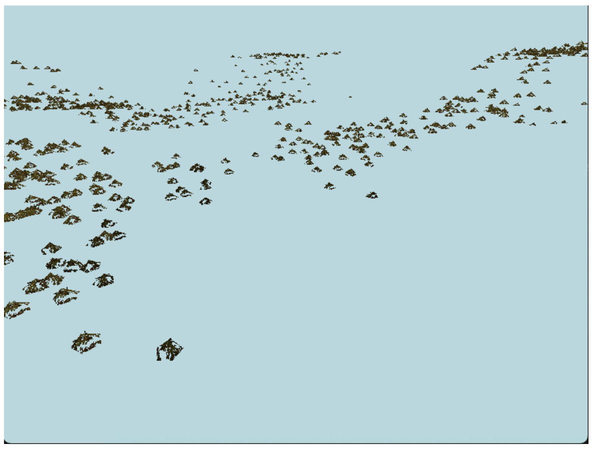
See a gif of a screenshot of every commit in the git repository.

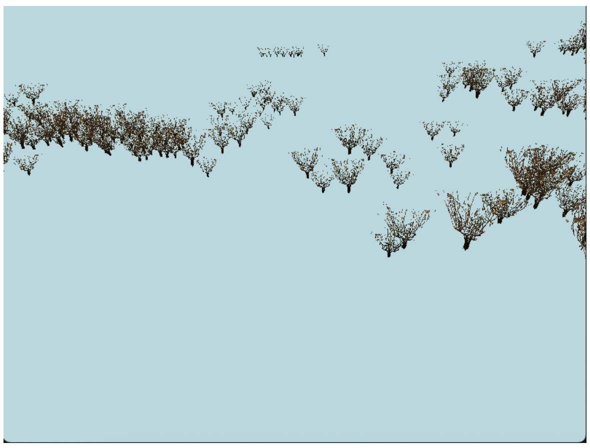
Individual components

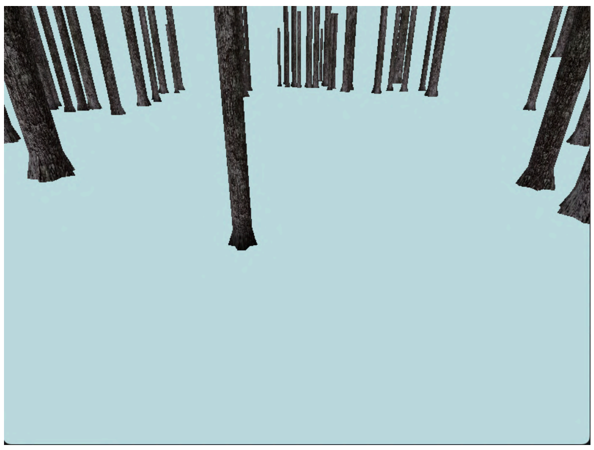

Ground layer


Bush layer


Saplings

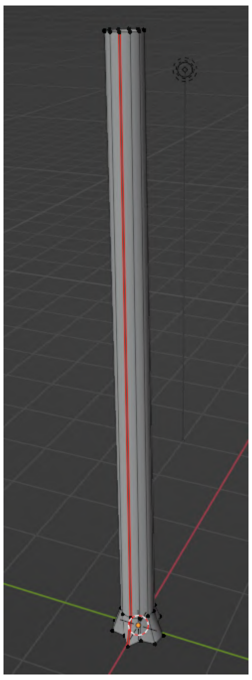
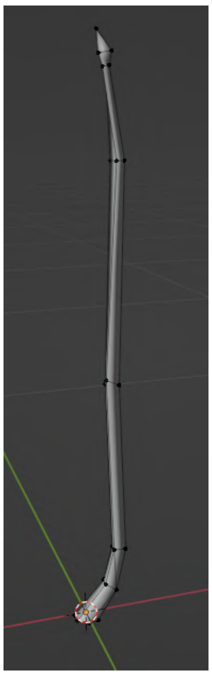
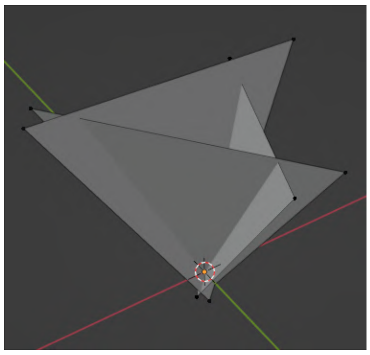
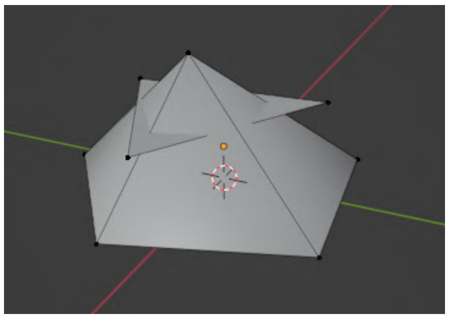

Bushes


Shrubs

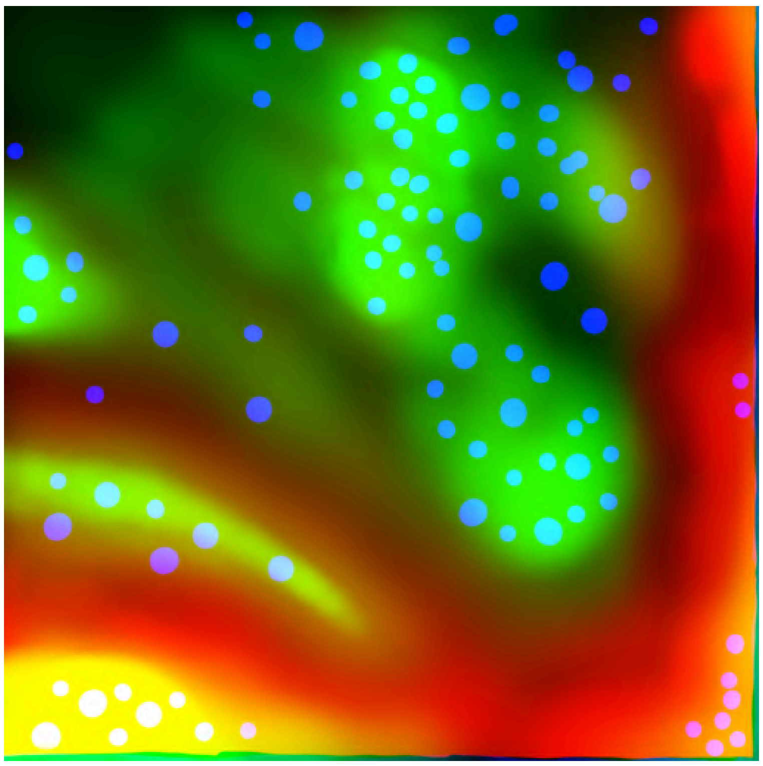

Trees

Textures

Blender Model Screenshots

Scene Map



As seen from the top left corner. Height in Red. Small foliage density multiplier in Green. Tree density multiplier in blue.