

Computer graphics Project 2023:

Natural scene in Three.js

By Mateo Van Damme and Casper Haems

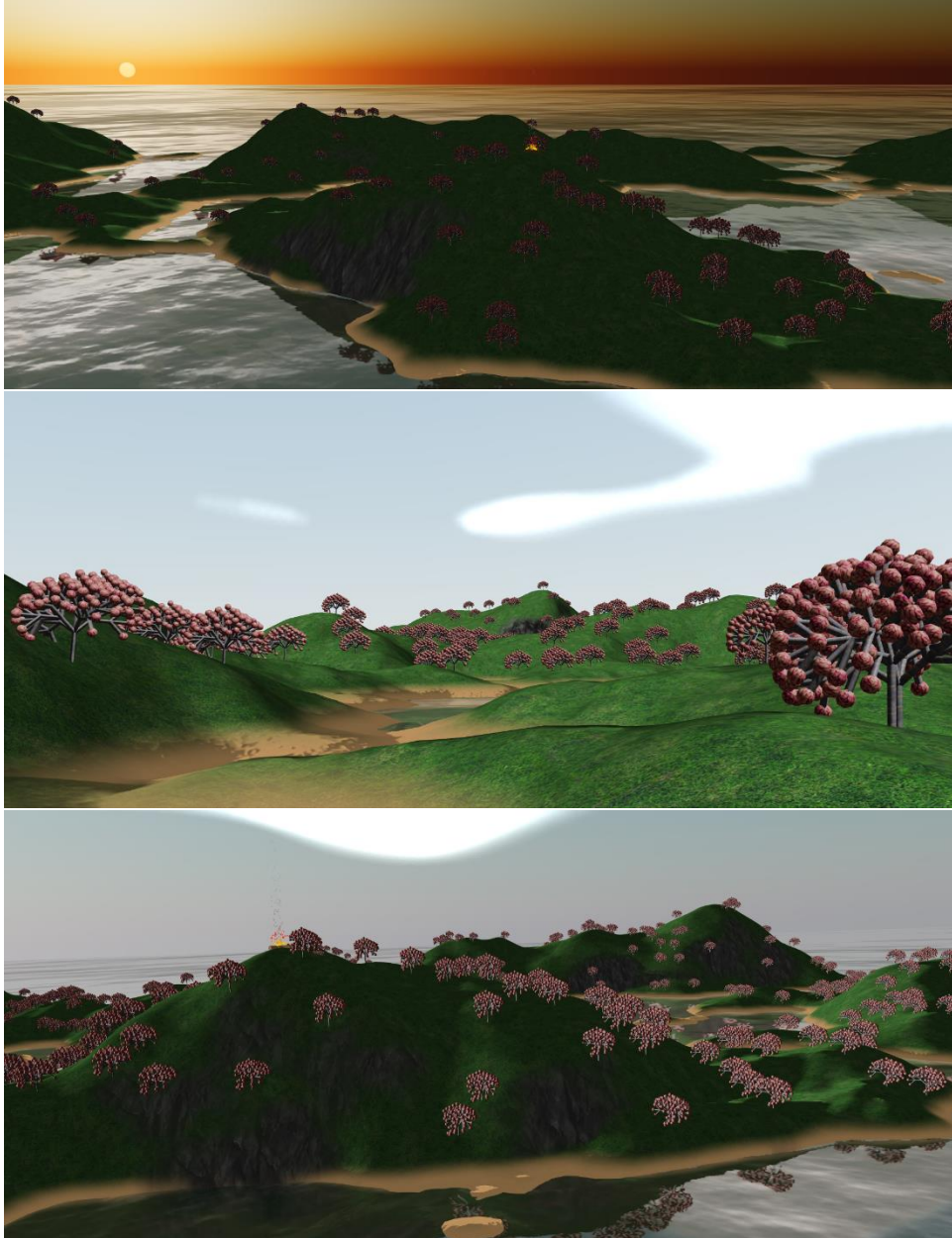


Figure 1: Natural world demo

Introduction

Our project is presented as a simple Bootstrap website where the user can switch between different *demo's* using the navbar:

Natural scene generation Natural Boids Fireflies 2 Fire Perlin LOD Infinite terrain Tree Clouds Tree 2D Fireflies Water Terrain Fractals Cube

Each demo presents a different part of the assignment and the *Natural* demo ties them all together in one natural landscape. During the development the project was continuously tested on desktop and mobile to ensure satisfactory performance.

The project is always available online at the following URL: <http://mcabla.github.io/3D-rendering>

To run the project locally host the root folder on a local webserver and click index.html. An internet connection is required for the Three.js library and Bootstrap.

At the end of each demo it is specified what existing code was used (if any) and which student worked on it.

General architecture (Casper)

The example code provided was used as a base for our project. We first started by refactoring the code to make it more modular. This allows us to load all our different test worlds as modules, loaded by the class Main (found in 'src/main.js'). Main is also responsible for starting the main game loop and renderer, which is reused for all our worlds. Additionally, the Stats.js module is also loaded in main so we can easily keep track of our performance during development.

Demo's

Perlin LOD (Mateo)

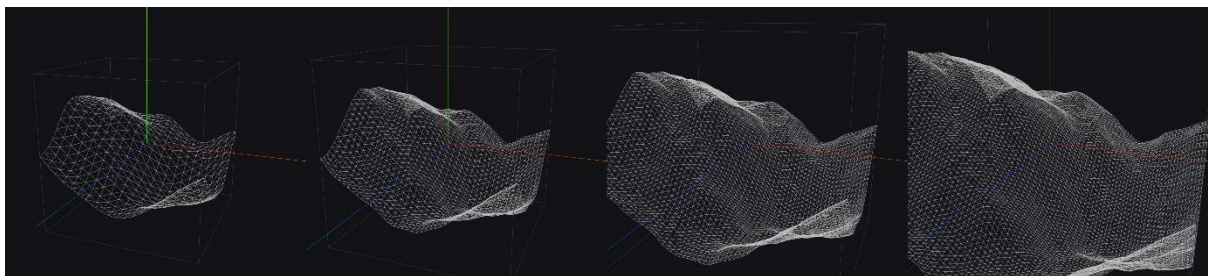


Figure 2: One chunk of automatically generated terrain with 4 levels of detail (LOD) depending on camera distance.

To present a natural scene, realistic terrain is required. In this demo the goal is to generate one chunk of realistic looking terrain using Perlin noise. This demo uses the class Chunk to generate terrain and changes the LOD based on the camera distance to the terrain. The terrain can be either a wireframe or a material supplied via the class constructor.

Terrain generation

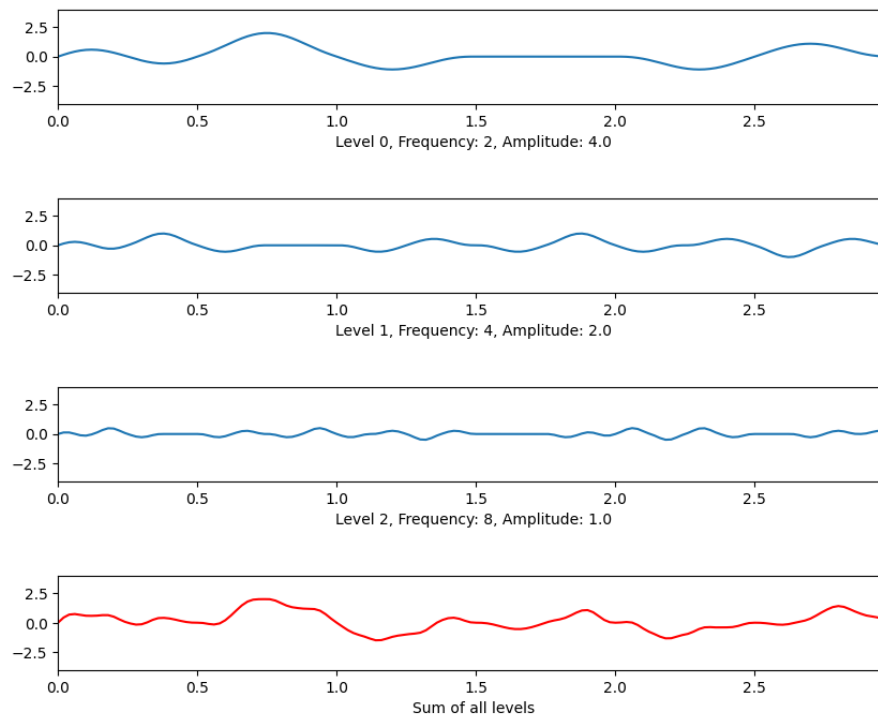


Figure 3: 1D terrain generation with an LOD of 3. The 3 levels are shown top to bottom with the final result displayed at the bottom in red.

The terrain generation is done by combining different layers of Perlin noise using addition. The LOD determines how many vertices are used for the chunk as well as the number of layers in the final result.

In Figure 1 the algorithm is explained in only 1 dimension. The LOD is equal to 3 which means 3 different layers of Perlin noise are used. When the number of the layer is higher the amplitude of the Perlin noise shrinks and the frequency of the Perlin noise increases. This results in lower levels providing a smooth baseline and higher levels providing many small details. Because the amplitude of the last layers is small they don't replace the layers before it, they only add more detail.

In Chunk this same principle is applied but in 2D instead of 1D.

Terrain function

A final modification is made to the implementation. At each level the value of the Perlin noise is transformed using a function called `terrainFunc(level, val)`. By default it just returns *value* unchanged but it can be overwritten to add extra character to the terrain.

In the *natural* world demo this function is overwritten with a custom one that only changes the value on the first layer by flattening the terrain under a certain y-level. On every other level the val is returned unchanged. This has the effect of creating more flat ground near water level. The end result is that the shoreline of the islands becomes more interesting:

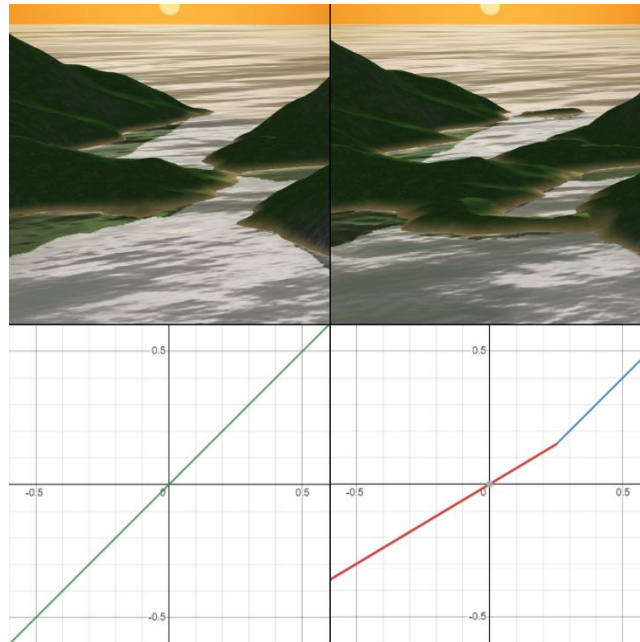


Figure 4. The top left shows the original terrain and the bottom left the default terrain function. On the top right the new terrain is shown and in the bottom right the custom terrain function is plotted for the first level. The red part is less steep and the blue part is as steep as before but shifted to the right slightly.

Parameters

The terrain generation parameters `baseFreq`, `freqGain`, `baseAmpl`, `amplShrink`, `terrainFunc` can be changed from the defaults but if one is changed all of them have to be assigned a value. The function `calculateLOD()` calculates the right LOD for a certain camera distance. This function can be overwritten to change which LOD is used at which distance.

Credits: The demo was written completely by me but Inspired by this example [threejs.org - terrain example](https://threejs.org/examples/terrain/)

Infinite terrain (Mateo)

The previous demo provided one chunk of realistic looking terrain. For the *natural* demo we decided to implement infinite terrain generation so the player can explore the world indefinitely in all directions. This means that chunks have to be dynamically loaded in and out memory and fit together seamlessly. The logic for this demo is implemented in the `ChunkManager` and `Chunk` class.

The `ChunkManager` loads a grid of $n \times n$ chunks around the camera at all times. The value for n can be chosen with the `viewDistance` parameter in the constructor.

When the camera moves, the `ChunkManager` calculates if chunks should be removed or added. It also commands each chunk to update its LOD, based on its distance to the camera.

This means that at any given time it is very likely that chunks with different LODs are loaded in next to each other. See figure below:

1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	2	2	2	1	1
1	1	1	2	3	3	3	2	1
1	1	1	2	3	4	3	2	1
1	1	1	2	3	3	3	2	1
1	1	1	1	2	2	2	1	1
1	1	1	1	2	2	2	1	1
1	1	1	1	1	1	1	1	1

Figure 5: A possible arrangement of loaded chunks when the viewDistance is set to 8. Different levels of LOD are visible ranging from 1 to 4. The player/camera is currently hovering above the red tile which has an LOD equal to 4.

This can cause gaps in the terrain when the chunk further from the camera has a lower LOD. Higher LODs means that more layers of Perlin noise are stacked together. If the extra layer on the chunk that is closest adds a dip in the terrain this dip will not be present at the other side of the boundary in the chunk with a lower LOD.

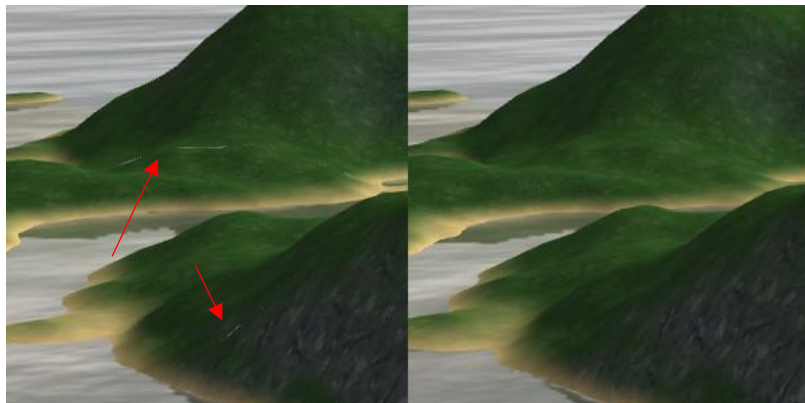


Figure 6: On the left gaps are visible between chunks as white streaks. On the right a fix is applied and the gaps are gone.

The problem can be mitigated by slightly raising chunks that are close to the camera, compared to chunks that are further away. This is implemented in the Chunk class where chunks with a bigger LOD get a bigger vertical offset. The end result is that small gaps in the terrain disappear because the closer chunks are staggered above the gaps in the further away chunks.

One drawback of this solution is that the terrain will rise the closer it gets to the camera because the LOD is increasing. When flying over the terrain, it looks as if the hills are growing slightly as they get closer to the camera.

Credits: Implemented completely by me (Mateo).

Boids (Mateo)

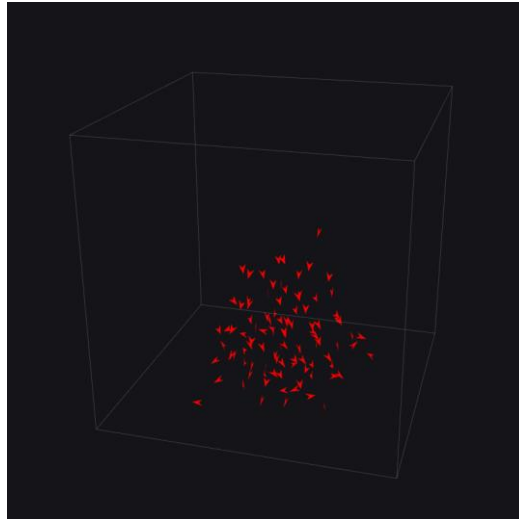


Figure 7: Boids demo

This demo shows boid behavior. Boids are mini organisms that all follow the same rules and together mimic natural swarming behavior that is seen in nature in flocks of birds or schools of fish. Our boids are implemented in the BoidManager and Boid class. Each tick the boidManager updates all his boids. Updating a boid means applying a set of rules that each exert their own force on the boid and affect its velocity.

Some rules can be applied to a boid without taking the other boids into account. Other rules require looking at other boids. The former executes in $O(n)$ time and can be computed quickly. The latter category needs to be optimized for efficiency because it executes in $O(n^2)$ time.

Boid rules

The rules are applied in this order. When a configurable parameter is used it is mentioned like this: `configurableForce`

1. **Avoid walls and floors.** There are currently two modes:
 - a. Floormode off: In the first mode the boids are contained in a cube. When they leave the cube they are pushed back to the center with force: `centeringForce`
 - b. Floormode on: This mode is equivalent to freeroam. This is visible in the *Natural* demo. The boids are pushed away from the floor with `centeringForce` and there is a weak attraction to the world center.
2. **Collision avoidance with other boids:** If boids are within a range called `minDistance` they are considered neighbors. Neighbors repel each other with a force called `avoidForce` that scales inversely with the distance between them. Similarly to how two equal poles of a magnet repel each other.
3. **Conform direction:** Neighbors also try to match each other's direction/heading. This is done by adding the velocities of each neighbour to the current velocity. The neighbours velocity is multiplied by parameter `conformDirection` and also scaled inversely with the distance between neighbours.
4. **Attraction to the center of the swarm.** The center of the swarm is calculated by taking the average of all positions also called a *centroid*. This calculation is performed each tick and

executes in $O(n)$ time. Each boid is constantly pushed towards the center with a force called `attractForce` that is scaled proportionally with distance. This means that a further a boid strays from the center the harder it is pushed back.

5. **Gravity.** Boids are constantly pulled downwards with a force called `gravity`. This prevents boids from flying up too far in floorMode.
6. **Normalize velocity:** All boids move at a constant speed at all times. After applying all the forces the new velocity is normalized to a constant speed called `constantVel`.

The collision avoidance with other boids rule (2) and the confirming direction rule (3) take into account neighboring boids and are therefore $O(n^2)$. To limit expensive computations the following shortcut was devised: Before applying rule 2 and 3 the distance to the other boids is calculated. If the distance to the other boid is greater than `minDistance` it is not considered a neighbor and rule 2 and 3 are not applied to the boid. This shortcut works because the repelling force and the force to confirm direction are scaled inversely with distance. So even if the rules were applied the effect would be negligible.

Other optimizations such as dividing the search space into an oct tree are possible but are out of the scope of this assignment. The boid system has been tested up to 1000 boids and with that amount it ran at 60 FPS on a laptop and 30 FPS on a mobile phone which is satisfactory for our purpose.

Development

When developing the boids, rules were added over time. This allowed us to observe which rules are most important for good boid behavior. At first the only rules were avoiding the walls (1) and normalizing the velocity (6). The first swarming behavior was observed after adding collision avoidance (2) and attraction to the center of the swarm (4). With these rules the boids had a closer resemblance to a swarm of angry bees than to a flock of birds. This is because without conforming to the direction of nearby boids (3) each boid just plunged towards the center of the swarm until it was pushed back by other boids.

At this stage I mitigated this flaw by making a single boid the center of the swarm. This workaround worked but the “leader” tended to get stuck in corners because all the other boids followed it and ended up crowding around it in a corner.

After turning off collision avoidance for the leader boid the boids acted as a swarm but they were still too rigid /unnatural in their behavior. This was because the direction of the swarm was only determined by the leader boid which was just aimlessly flying around until it hit a wall and bounced back.

When rule 3 (confirming to the direction of neighbors) was introduced all workarounds became unnecessary and were removed. This rule was the most difficult to figure out conceptually and implement but was essential for good boid behavior. The concept of a leader boid was removed and all boids were now equal. Finally rule 5 was added to prevent the boids from flying up too much in floorMode.

Credits: I (Mateo) wrote all the code myself but was inspired somewhat by this implementation: <https://github.com/juanuys/boids>

Trees (Casper)


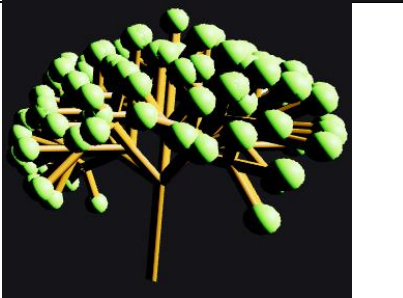
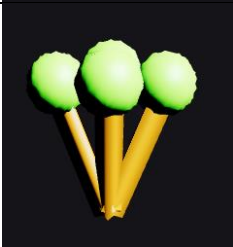
The trees in our world are generated using a Lindenmayer system (L-system), which is a type of formal grammar used to model biological structures like trees and plants. The L-system is defined using a set of rules. These rules are applied repeatedly to generate a string of characters.

Each character in this string represents an action, such as moving up, spawning a leaf, rotating in a certain direction or making a branch.

The resulting string is then converted into a series of branch instructions, which are used to generate a 3D model.

The development of this system was quite challenging because we wanted it to be quite modular. For example, you can see that we are able to pass different meshes and materials to the generator. This resulted in some strange trees to be generated during development because of some orientation problems with our different meshes. In the end we fixed our orientation problems by putting every branch in a group object. Giving the branch a set direction and then just applying the L-system rules (rotation and position) to each group object.

Credits: I (Casper) wrote the whole implementation by myself.

Maple tree rules = { 'A': 'EEC', 'B': '[+C][-C]/[C]*C', 'C': 'EF[+C][-C]/[C]*C^^C', } branchAngle = $\pi / 4$ -> angle between branches branchLength = 0.04 -> length of each branch iterations = 5 -> number of times to apply the rule	
Cherry blossom rules = { 'A': 'EF', 'F': 'E[+F]/[-F]*F[+F]*F[-F]/[*F]', } branchAngle = $\text{Math.PI} / 4$ branchLength = 0.040 iterations = 4	
Flower rules = { 'A': '[+F][-F]/[*F]F', } branchAngle = $\text{Math.PI} / 8$ branchLength = 0.040 iterations = 1	


Spruce rules = { 'A': 'EEEEB', 'B': 'CFC^B', 'C': '[+++++++R]----- S]//////////T][*****U]', 'R': 'R-F', 'S': 'S+F', 'T': 'T*F', 'U': 'U/F', } branchAngle = Math.PI / 16 branchLength = 0.040 const iterations = 6	
--	--

Table 1: Different tree types

Fire - Particle system 1 (Mateo)



Figure 8: Fire demo

The fire demo shows an implementation of a fire particle system that is simple in implementation and provides a pixelated, though aesthetically pleasing campfire that could be used in games that don't aim to be hyper-realistic but instead rely on art direction and style like Minecraft. The fire particles demo is implemented using the THREE.Points class. A couple logs are added under the fire to complete the campfire.

When implementing this demo the main challenge was making sure the fire looked convincing at different scales and deciding what to implement in Javascript and what to implement in GLSL.

In the end it was decided to implement all the motion in Javascript and the calculations for particle size and particle color in GLSL. This has a couple of advantages: calculating positions in Js is just a lot easier than doing it in GLSL code. Secondly interpolating what the color should be for 10K particles is done more efficiently by a GPU than by a CPU.

In Three.js buffers are used to efficiently store and manage data for rendering geometries. For the fire particle system, 3 different buffers are used:

1. **Position buffer:** This required buffer stores the position for each particle.
2. **PositionStart buffer:** This readonly buffer stores a stationary start location for each individual particle. The user specifies the desired location of the fire at startup and random points around it are calculated. Later each particle will always add its startPosition to its relative position so the fire appears at the correct location.
3. **ID:** Each particle gets its own random number between 0 and 1 at initialization of the fire. This random number is used at various points when determining the position, color and size to make each particle appear different. Because the id determines everything about the particle it is essential that particles next to each have a different id. See position section.

To calculate the position and appearance a distinction is made between fire and smoke particles. The ratio can be configured with the parameter `fireToSmoke` ratio that should have a value between 0 and 1. Because each particle also has a static id between 0 and 1 we can threshold the id to determine whether the particle is fire or smoke. Example: If the ratio for `fireToSmoke` is 0.8 only 20% of the particles will have an id over 0.8 and will be considered smoke.

Position

The positions need to be calculated at each tick for all particles. This is done in Javascript. All particles follow a spiraling pattern in an upwards direction. The modulo of the y coord is taken so the particles respawn at the fire when they travel too far upwards. Each particle moves towards positive y and circles in the x-z plane based on functions that take into account the current time and the particle's id. Smoke particles move slower and their spiral has a tighter radius. To make this simple motion less obvious each particle's spiral is centered around its startPosition which is different for each particle. The speed in the y direction and rotational speed in the x-z plane are also depends on the particle's id. Finally each particle is at a different vertical y-level and phase angle in the x-z plane. It is crucial that particles with a similar id don't appear near each other because the id also plays a role in the size and color of the particles.

To accomplish this the id is multiplied with a large prime number. This works because it spreads the ids in the y-direction and x-z circle. Remember that the modulo is taken of the y-coord so multiplying can't make the particles fly away too far.

Color and size

In THREE.js a ShaderMaterial can be used to provide your own custom shaders for rendering. Both the fire and smoke particles in this demo use the same custom shader called *particleFire*.

In the shader the id is again used whether the particle is a smoke or fire particle. Both particles get smaller the further they are from the camp fire but smoke particles start smaller and shrink more slowly. This has the effect that the plume rises above the fire. The particle's id also plays a role in the shrinking rate, otherwise all particles at the same y-level would have the same size.

The color is calculated in a similar matter. Smoke particles are always grey and fire particles range from yellow near the logs to red at the tips of the flame. Again the id affects the speed of the transition to give the particles more individuality.

Finally, if particles are too small they are not rendered.

Credits: Written completely by me (Mateo).

Fireflies - Particle system 2 (Mateo)

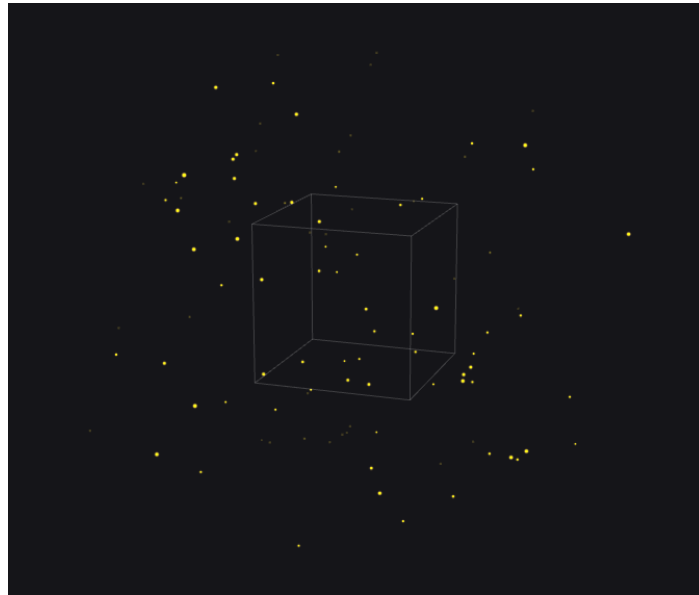


Figure 9: Fireflies demo

Two additional particle systems were implemented in the project: Fireflies and Clouds. Because they are conceptually not that different from the first one the discussion will be very brief.

Fireflies implements a swarm of fireflies that dance around each other. Conceptually it is very similar to the *Fire* demo except that everything is now implemented in JavaScript. It also uses the `THREE.Points` class and a `ShaderMaterial` to represent one particle. The shader for the particle called *materialParticle* is more generic compared to the *fireParticle* `ShaderMaterial` used in the *Fire* demo, it only displays a texture with the right opacity, the color never changes.

Credit: Implemented completely by me (Mateo).

Clouds - Particle system 3 (Mateo)



Figure 10: Clouds demo

This demo aims to provide clouds using a point system. The same particle material is used as in the *Fireflies* demo but with a different texture. It works by displaying particles in a grid above the camera. When the camera moves in the x-z plane the entire grid moves with it. The y-coordinate of the clouds is always the same. For every tick each particle in the grid calculates its correct size.

The particle's x and z coordinate are added to the current time and given to a Perlin noise generator. The result is thresholded so clouds don't appear everywhere but only in patches. By considering the position as well as the time the clouds move, even when the camera is stationary. This makes it look like wind is blowing the clouds away. If the shader detects that a particle is too small it discards it.

Natural (Casper)

This is our main world and puts all the different parts together in one natural scene.

Terrain shading (Casper)

To give the terrain some textures, a custom shader was written that maps textures onto the terrain based on a few rules.

To explain these rules. We have made a simplified shader with a unique color for every texture.

1. The first rule is selecting a grass texture if the terrain is above $y = 0$. Below $y = 0$ the terrain will be sand. This way, we can generate islands with beaches. In another part of the report, a water shader is discussed, which plays nicely with our terrain shader.
2. Next, we map a rock texture to the terrain if the terrain is too steep. This allows us to emulate cliffs based on terrain steepness.

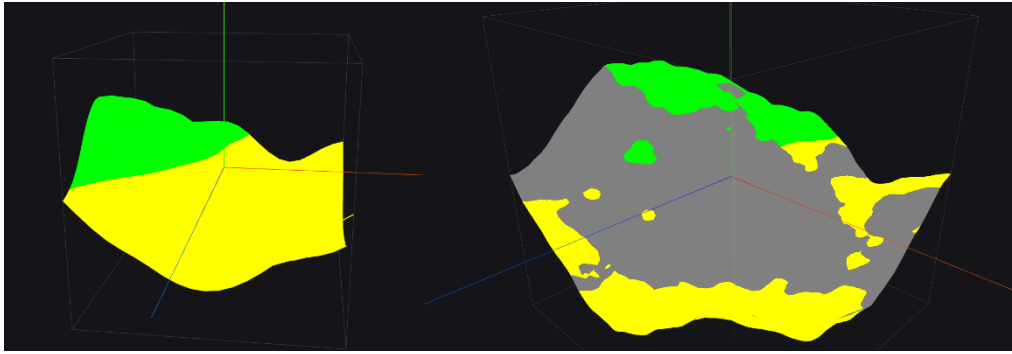


Figure 11: Height-based texture mapping + steepness-based texture mapping

One can easily see that the borders between the textures are quite hard. To mitigate this, a smooth step is supplied instead of a basic step function. Now, replacing the colors with their respective textures, a nice terrain can be displayed.

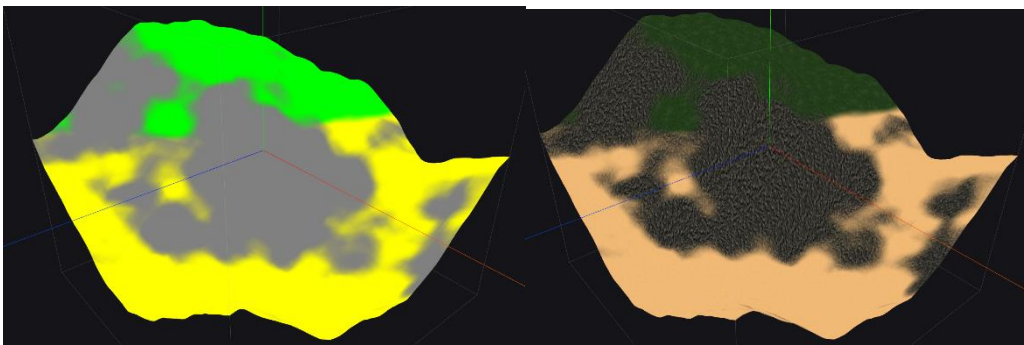


Figure 12: Smoothing borders between textures + textured version

We are, however, not yet finished. Some basic lighting should be implemented. For this, a few new uniforms have been introduced. First, there is an ambient color. Second, the sun color and its direction. Both colors also have an intensity variable.

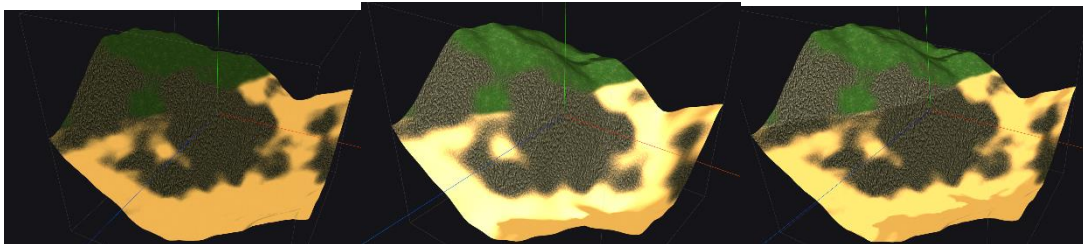


Figure 13: Ambient lighting + directional lighting + capping sunlight diffusion below grass level

Notice that the sand texture tends to be too bright. To fix this, a simple brightness capping was added below the grass level. This simple fix has the unfortunate effect of also darkening the rock texture below the defined height. This, however, does not form a problem, because these rocks will be at and below the water level anyway. This can even be interpreted as a bit of wetness on the rocks.

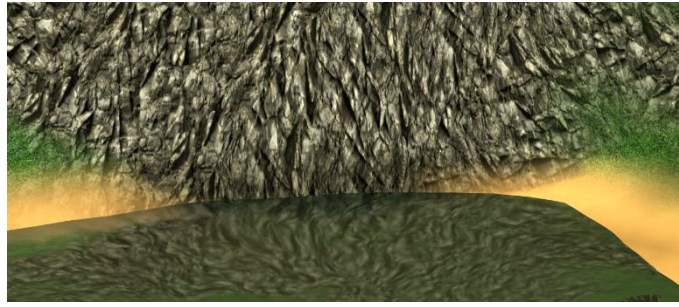


Figure 14: Capping sunlight diffusion at water level also simulates wetness on our rocks.

One final touch to finish our terrain shader is adding support for normal maps. A normal map is a type of texture that encodes surface normals in RGB values, allowing for the simulation of finer surface details and lighting effects in 3D graphics. The RGB values of the normal map represent the X, Y, and Z components of the surface normal at each texel, and are used to perturb the lighting calculations for the surface, giving the illusion of greater surface detail and complexity.

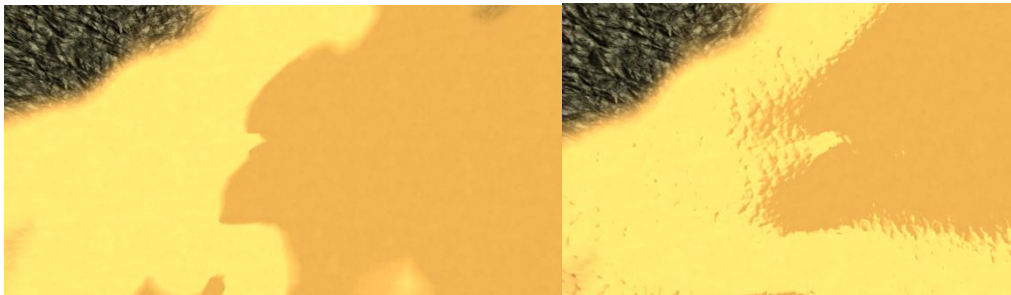


Figure 15: Topdown view without normal map + with normal map

Not only does this give our sand more details. It also improves the lighting realism. This can be seen with our rock texture. Certain parts of the rocks are more lit then other parts. This depends on the direction of the sun. You can play with the azimuth control in the demo to see this in action.

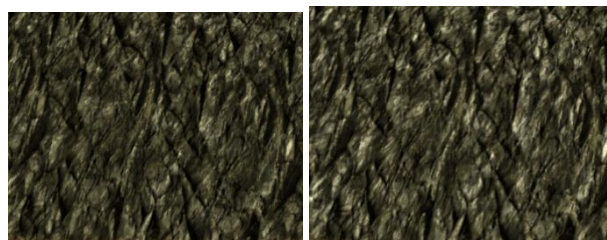


Figure 16: The same rock texture, lit with the same sun, but from a slightly different direction.

Credits: I (Casper) wrote the whole implementation by myself.

Problems during development

This was the first time that we wrote a shader for our project, so a lot of trial and error came into action. Here we learned what each variable does and how its value maps relative to the camera or the world. Everything we learned here was useful in the other shaders we discussed above. For example, for our particle systems.

In the screenshot below, you can for example see what happened when we accidentally used the screen height instead of the terrain height. Please note that the textures used below are placeholders generated with Bing image creator.

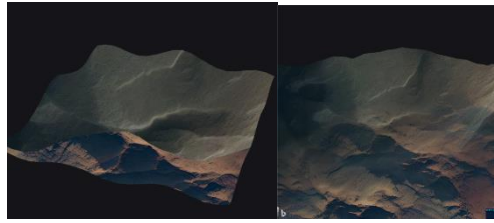


Figure 17: Textures based on the y coordinate of the screen, not of the terrain.

Also, the steepness rule didn't work from the first try. Researching a bit about the required matrix transformations fixed the issues.

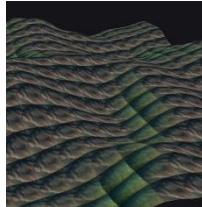


Figure 18: Every left side of the hills is grass, everything else is rocks.

Texture synthesis (Casper)

Even though our textures are seamless and smoothed between each other, one important problem still occurs. When zoomed out enough. Repetitive patterns can be seen. This can be prevented by using texture synthesis. There are multiple ways to synthesize an image. One could for example generate a bigger texture from one input texture. However, the patterns can still be spotted when you zoom out even more. This is why we choose to generate new textures and map these on our terrain in a pseudo random way.

Both the rock and sand textures have a normal map in our demo. The grass texture does not, so we limit our texture synthesis to only the grass texture.

For generating new textures based on an existing texture. We used the following tool:

<https://github.com/texturedesign/texturize>

Now that we had a second grass texture, we had to pseudo randomly select a texture between the two.

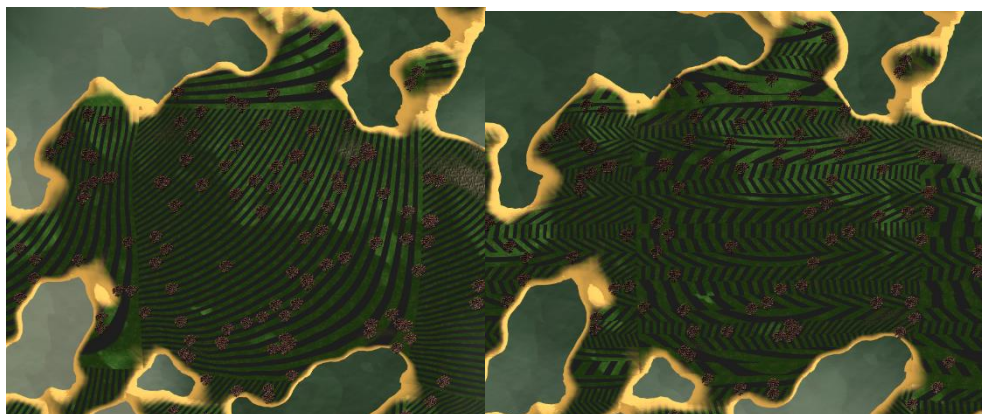


Figure 19: Basic pseudo random selection (with gray as the second texture) + even more random selection

When setting the second texture, a new problem occurred. The edges were glitching out in the distance. This was easily fixed by setting a small smoothing between the two textures on their edges.



Figure 20: Glitched borders + smoothed borders.

Now that we had a working terrain shader, a few abbreviations were made. For example, the infinite terrain world uses a similar shader, but without the normal mapping, sunlight capping, texture synthesis and with other textures and default parameters.

Credits: I (Casper) wrote the whole implementation by myself.

Tree spawner (Casper)

It would be too big of a performance hit to generate every tree at runtime in our natural scene. Which is why the export functionality is added to the tree world. We chose to export the cherry blossom tree to a model file. Then we used blender to merge the meshes together as much as possible.

In our chunk code, we now calculate a second Perlin noise in the existing generation loop, with a higher frequency than the terrain noise. Whenever the output is greater than a certain threshold and a few other conditions are met, a new tree is placed.

Note that we only use one tree model. To improve the performance and save resources, we use instance meshing per chunk. Only after all the possible tree locations have been calculated, is the instance mesh created. This is because the constructor requires the number of trees that it should create.

We do not have different LoD versions of our tree. Which is why we run the tree spawner together with our terrain generation every time the LoD changes. For every higher LoD, more trees are allowed to spawn. This results in a few trees in the distance and more trees close to the camera.

Credits: I (Casper) wrote the whole implementation by myself.



Figure 21: Top-down view with more trees in the center.

Problems during development

Even though our model has been merged using Blender, it still contains two geometries. For this reason, two instance mesh objects have to be created. One for the crown of the tree and one for the trunk and its branches.

Another disadvantage of these two objects is that they are not standing upright in the model file. At first, we tried to manually fix the rotation and position, but this did not yield us the results that we wanted. After a lot of searching, we found that all these variables can be found within the model itself. Their parent object in the model file fixes this problem, but the fix does not transfer to our instance mesh. Manually setting the rotation and position by the values we now have found, fixed this annoyance.

Water shading (Casper)

For our water, we used the default water shader supplied by Three.js. One common use case for custom shaders in Three.js is to create a water surface that simulates the appearance of waves and ripples. This is typically achieved using a combination of a normal map and a noise texture, which work together to create a realistic water effect.

By applying a custom water shader to a plane geometry, developers can add a realistic water surface to their 3D scenes, complete with reflections and refractions that respond to the angle of the camera. This technique can be used for a wide range of applications, from games and simulations to visualizations and artistic projects.

This shader works by mapping a normal map of water to a plane, applying a watercolor, the color of the sun and its direction. Note that this shader does not use a noise texture.

It is a really nice and extended shader with support for reflections, distortion etc. But there are a few problems with it. One problem is that some stuff from behind the terrain, can be visible where the terrain hits the water. This is caused by the distortion.



Figure 22: Reflections on the water result in small glitches.