Universidade do Minho

Procedural Generated Terrain

Bruno Ferreira PG41065, José Fernandes A82467, Vasco Figueiredo PG41102

30 de janeiro 2020

Índice

1. Introduction

Procedural generation is used for created randomly generated assets and 3d models. One of the things that can be created with procedural generation is the generation of terrain. Procedural Terrain is commonly used in video games and in other areas in order to create vast landscapes in mere seconds which can then subsequently be used in creating various different scenarios.

In this report we describe all of the steps we took to implement procedurally generated terrain. We also cover the implementation of grass which was placed in our generation and the application of post-processing applied to the sky-box to give the skies some life.

2. Terrain Generation

2.1. Height Map

For modeling the terrain we used a grid made out of quads. We then attributed heights to said grid based on their position.

For calculating heights we used Perlin noise which is a gradient noise developed by Ken Perlin which provides us with a pseudo random value which always returns the same values when given the same parameters.

The perlin function we used received the following parameters: the position, a dimension value that could be used to scale the function and a seed value that we used to get different outputs. We scaled the position by 10 when developing the height-map to make the peaks more distant one from another and later increased the scale by 4 both vertically and horizontally after including textures and multiplying the result to get higher peaks.

The return value of the function for each point was used as the height value.

Because the transitions of the height values were too smooth we decided to roughen up the terrain by stacking multiple versions of the noise (octaves). With each version of the noise having a different "seed" to prevent effects caused by overlapping the similar positions of the noise at different scales.

The formula used for this was the following with i being the number of octaves:

$$result = \sum_i \frac{1}{2^i} * perlin((x, z), 2^i, seed + i * 100) \tag{1}$$

In figures 1 and 2 we can see that by stacking more octaves we can get more detailed results. The number of octaves stacked can be changed in the NAU3D interface.

The height of the points was calculated in the tessellation evaluation shader.

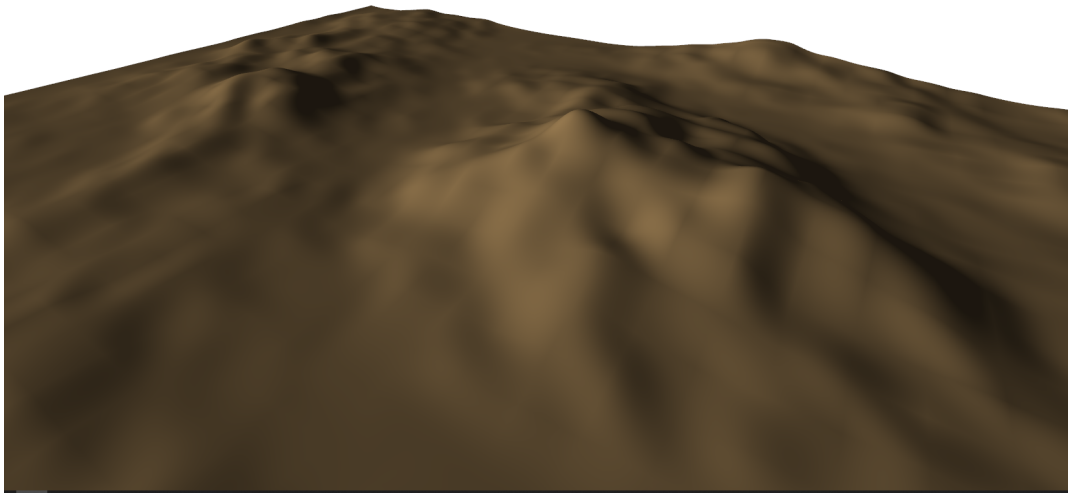


Figure 1: Terrain with 5 octaves

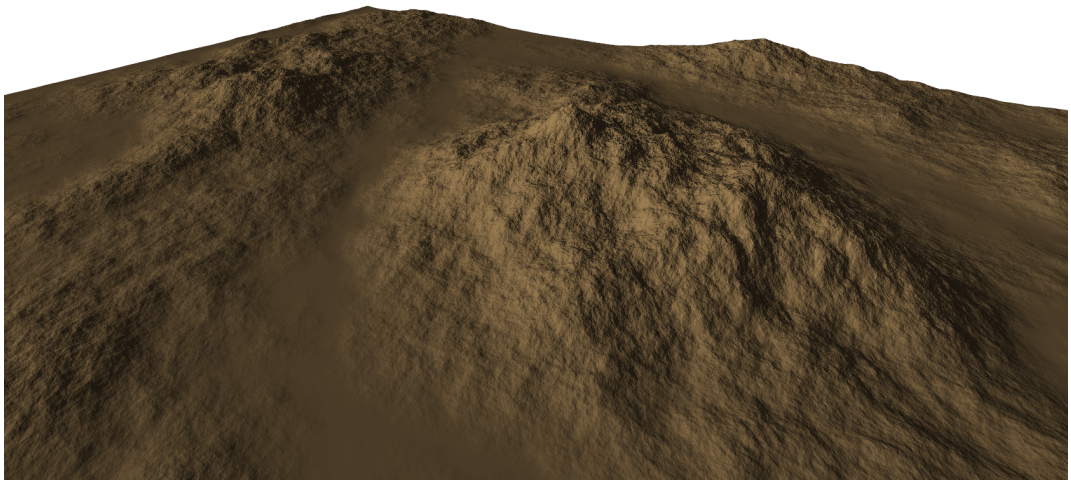


Figure 2: Terrain with 10 Octaves

2.2. Normal Mapping

In the figures from before the differences from using different noises are more easily noticed because of the uses of normals. These normals were calculated in the fragment shader which allows us to have details inside the triangles.

These details contribute to lessen the visual impact of using lower triangle counts and also enables the use of higher counts on the number of octaves. The downside of doing this on the fragment shader is the increased computational cost.

The normal vectors where calculated by adding the cross product of the forward and right vectors with the cross product of the backwards and left vectors and normalizing them. We calculated 2 cross vectors because we were getting some weird looking illumination when calculating only one because the normal vector in peak points wouldn't be facing up.

2.3. Infinite Terrain Illusion

We were able to obtain the infinite terrain illusion by adding the position of the terrain to the camera before calculating the heights. This was done in the vertex shader. We also needed the points to be in the world position because we used multiple quads with different translates. As a result we multiplied the position by the model matrix before adding the camera position in the vertex shader and only multiplied the position by the projection and view matrices in a later step.

To prevent triangles from changing heights when moving we added only the integer part of the camera's position to the points.

To simulate infinite terrain we generate positions and normals every frame which is itself really taxing should the generation be static we could just render all of these values into textures and then load them which would avoid these calculations and thus increase performance

2.4. Smoothing the terrain

Like we can see in figure 3 using perlin only gives us an uneven surface, it does not however give us clearly defined peaks. We can get more clearly defined peaks by multiplying the height result by a scalar like we can see in figure 4.

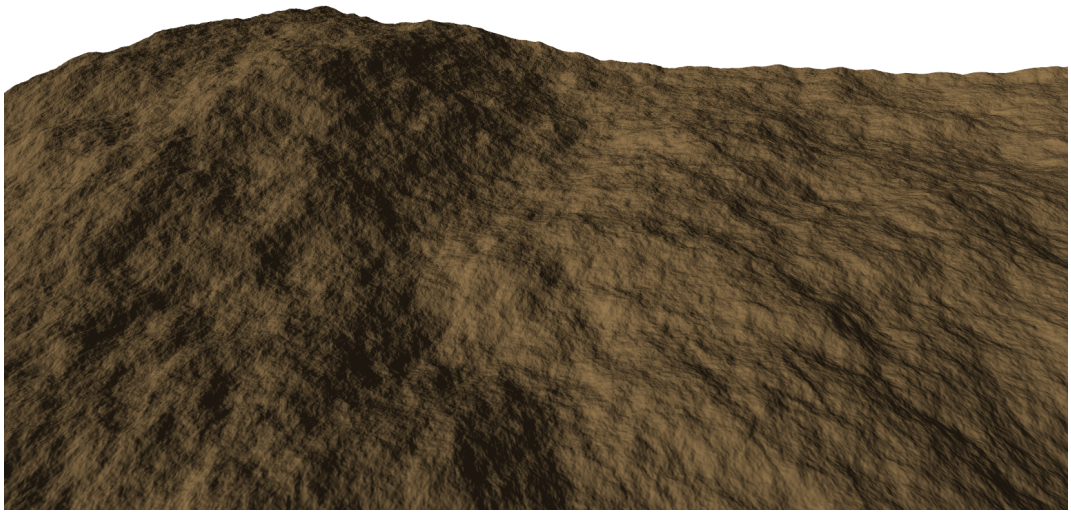


Figure 3: Terrain with Perlin Noise

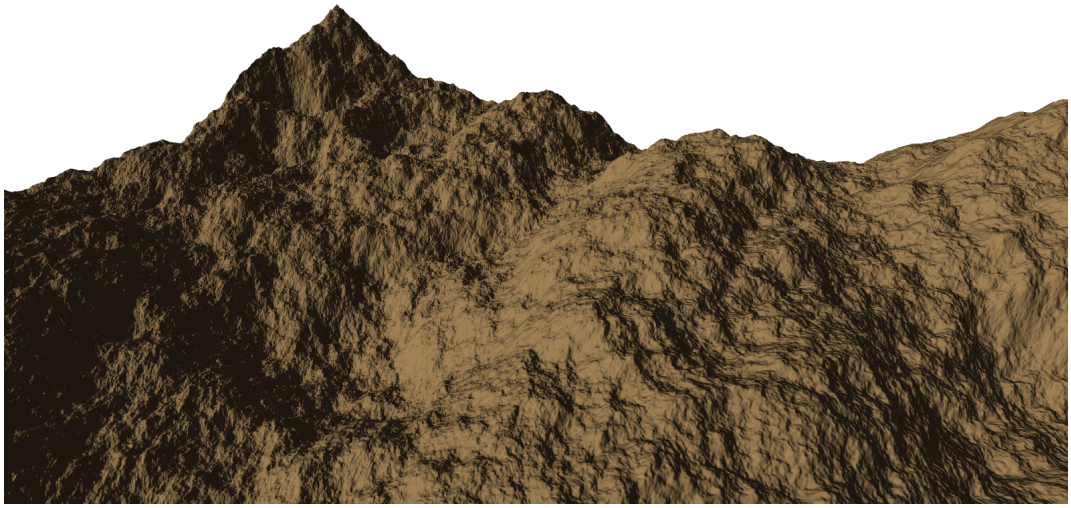


Figure 4: Terrain with result multiplied by 3

2.4.1. Cubic and Square Functions

While the results of multiplying the terrain by a scalar did give us higher peaks and lower bottoms (since our Perlin function returns negative values) it did not give us smooth areas in the middle like the ones we were seeking. As a result the terrain generated was always mountainous.

After experimenting we decided to smooth out lower values while increasing the higher ones by applying a function to the result. In figure 6 we can see the result of calculating the square of the noise and in figure 7 we calculated the cube.

We decided to go for the cube of the noise because it created a smoother curve at lower values which gives us smoother planar surfaces at lower level of noises and it gives us higher peaks, since our peaks are higher than 1 due to the stacking of the noises.

It also provides us with depressions on the terrains as opposed to the square function which made all values positive (which gave us more mountains).

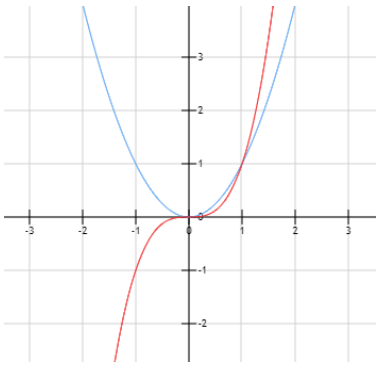


Figure 5: Cubic and Square Functions

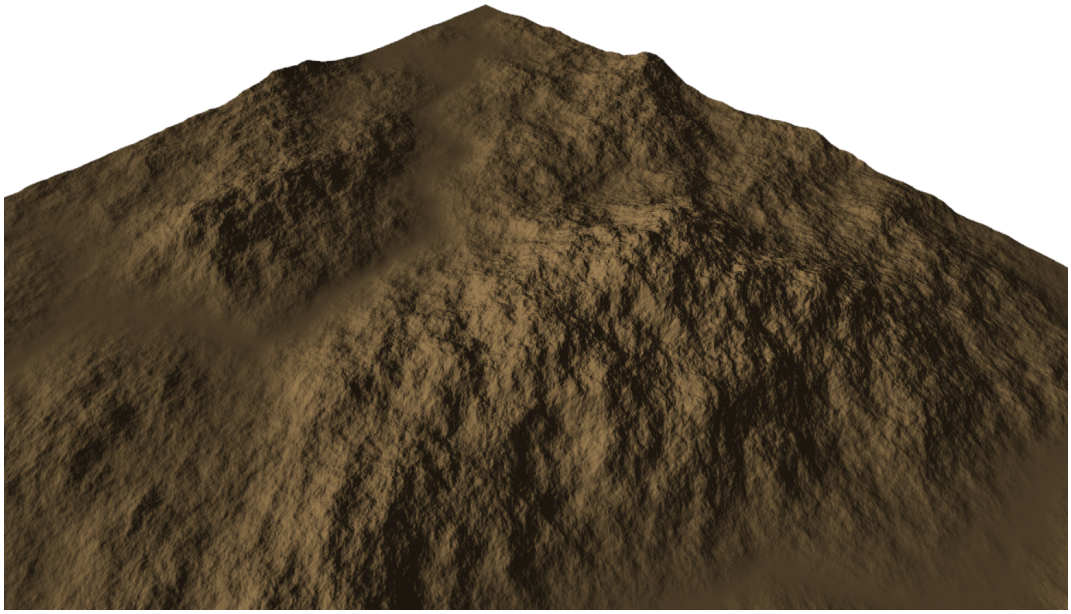


Figure 6: Terrain with result powered by 2

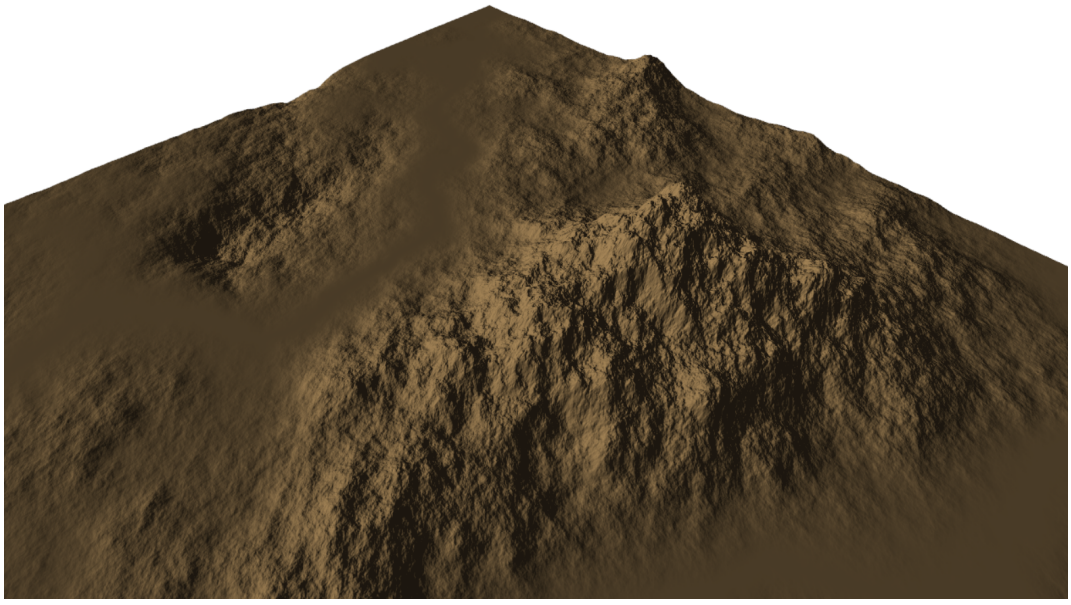


Figure 7: Terrain with result powered by 3

2.5. Tessellation

We implemented tessellation of the terrain which enabled us to vary the number of triangles on screen in run-time. Another motivation for implementing tessellation was to enable the control of grass that appears on screen which will be described later in the report.

It also enables us to have more triangles close to the camera while having less away from the camera which gives us control over the level of detail.

We also implemented tessellation based on the distance from the camera, but, instead of doing a linear tessellation based on this distance, we used an easing out quintic function that is described as the next formula.

$$TessLevel = 1 - (t - 1)^5$$

This gives us high levels of tessellation at near and medium distances but quickly drops the tessellation at further away distances. This is good because at these distances the low amount of geometry does not affect much as it's too far to be seen.

Something to take into consideration is the values of the outer levels which need to be match the neighbour quads so that there isn't any gaps in the grid. We did this by calculating the intermediate point of two vertices of an edge and then using it to calculate the distance to the camera. Because at the start before applying noise, the corners of different quads are virtually in the same location which will result in the same outer tessellation level.

Some of the parameters used to control the tessellation done can be changed in run-time in the NAU3D interface. These are the maximum ilevel and olevel and the near and dist values on the equation.

In figure 8 we can see the tessellation in action from above, the low triangle count is there because of the distance, and in figure 9 we can see that, aside from the side, the low triangle count is barely noticeable because of the angle at which are looking and because the normal maps are being calculated per pixel instead of being calculated per vertex.

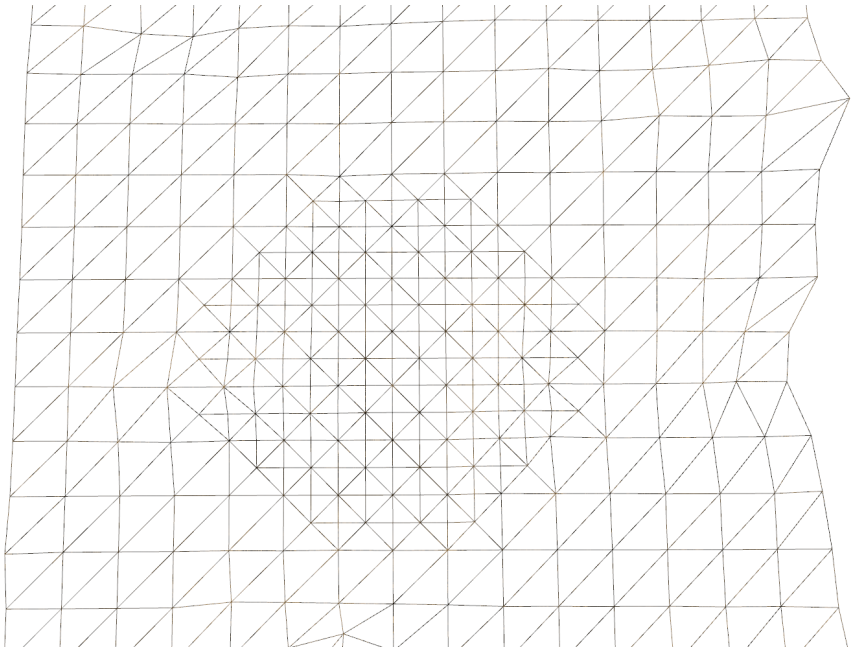


Figure 8: Tessellation viewed from above (low triangles because of distance)

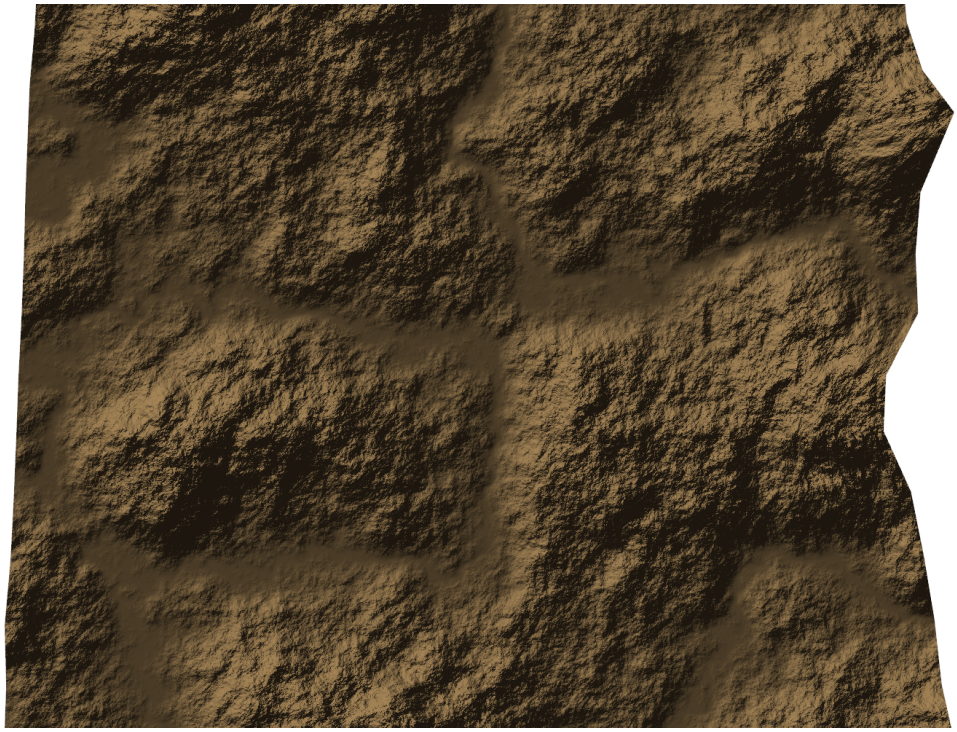


Figure 9: Terrain rendered from above with tessellation

With this level of detail implemented, we also created a check that if a quad was deemed to be outside of clip space, both the outer and inner levels would be set to zero so no geometry would be generated and hopefully optimizing it. However, as we got very close to the surface, geometry would be disappearing close in front of us because that geometry belonged to a quad that was behind the camera. We were not able to find a good value that would make this artefact not happen yet still optimize by not creating unneeded geometry. For this reason, we had to discard this functionality.

2.6. Texturing

Our initial idea for texturing was to do a separation based on world height of the fragments. By starting with water, which was just a color since we're obviously not going to try and simulate water with textures, sand, grass, rocks and then snow, we also added for each of these layers a second set of textures that would correspond to surfaces with a big slope, determined by the vertical component of the calculated normal. Only for the grass and snow however, whereas grass would have a dirt texture and snow would have the same rock texture from the layer underneath. This was a trivial choice and textures for the other layers can be easily implemented.

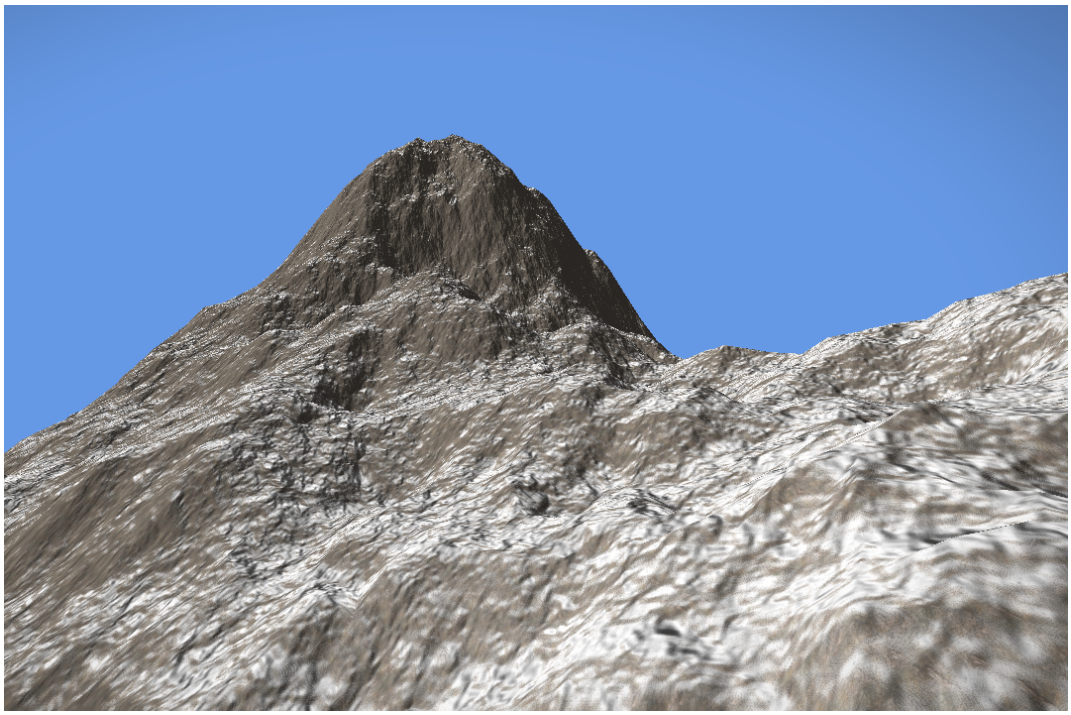


Figure 10: Snow layer with rock texture where the slope is too big for snow to be there.

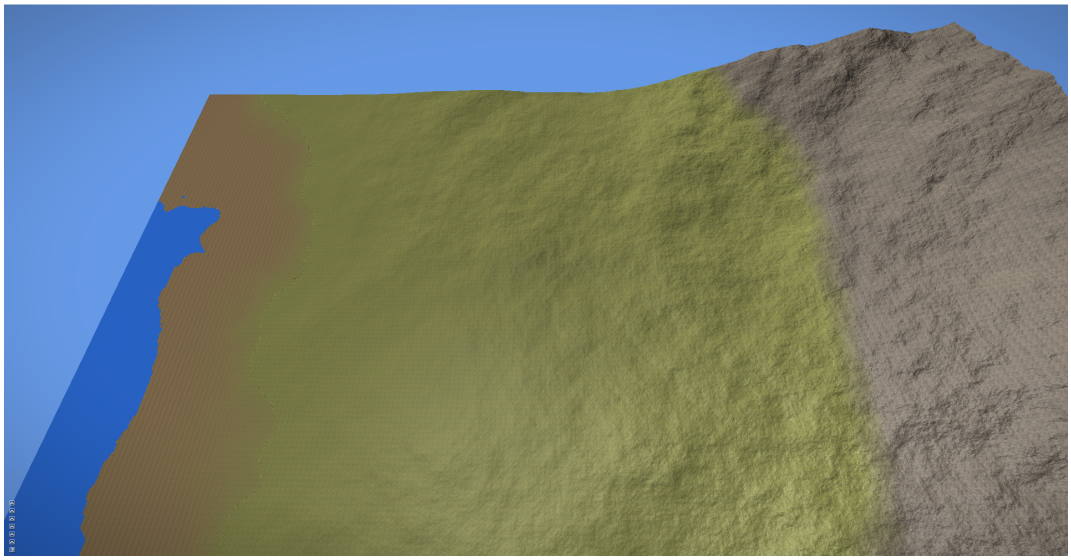


Figure 11: Texturing showing water, sand, grass and rocks. Notice how grass does not show any dirt because the surface is too flat for it to appear.

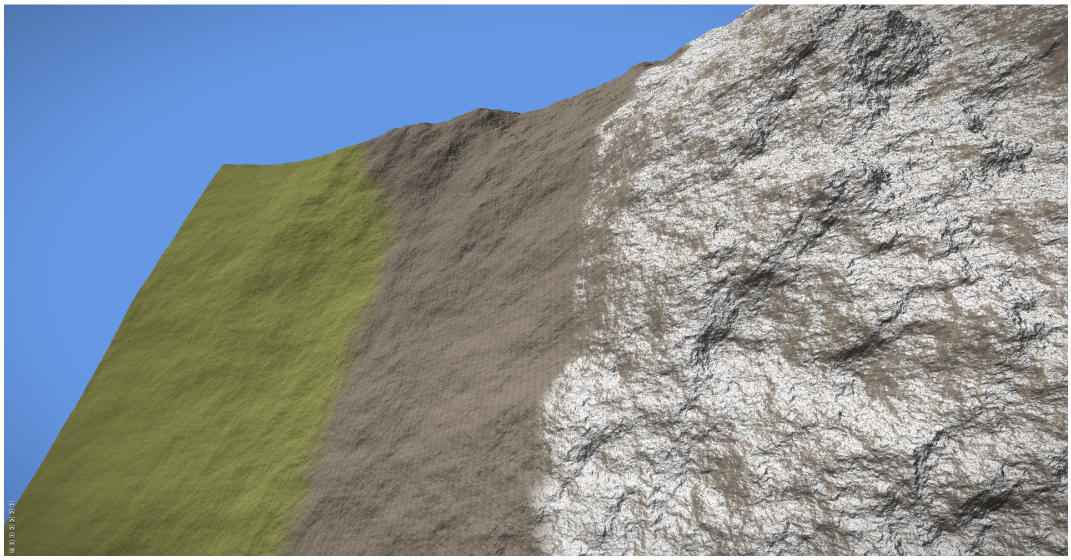
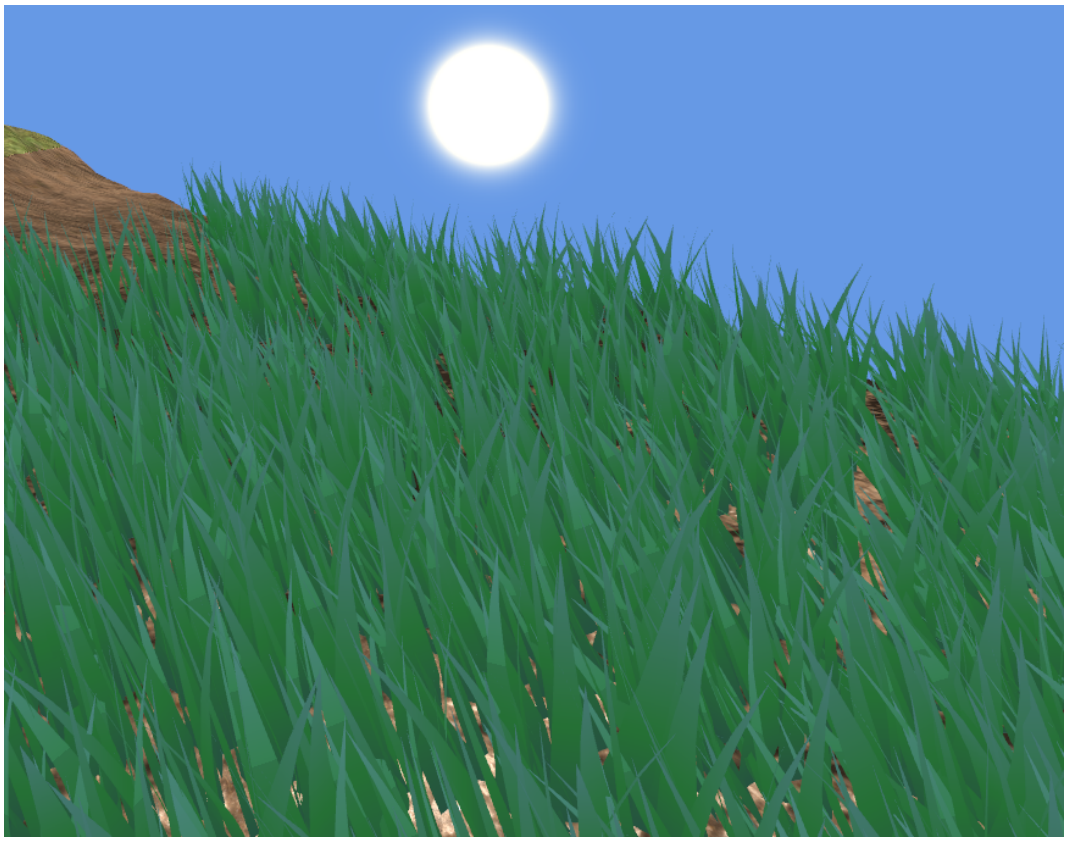


Figure 12: Texturing showing grass, rocks and snow.

When going from a layer to another, we have a small window where both layers are mixed into each other to give off a natural feel instead a very sudden change of textures. To add onto this, we noticed from afar these separations between layers looked too perfectly horizontal, so we added perlin noise vertically to decided where the mixing of layers should occur.

Roughness and normal maps were something we at some point of the work we were using. However, due to problems when combining our work together, we left it out to solve everything and eventually never added this feature back. It's not too problematic however since we are able to have very fine noise to sort of replace it. It would still have been good to combine both the normal maps with this very fine noise, as it shouldn't be too hard since we eventually ended up calculating the tangent and binormals on the geometry shader.

3. Grass Shader



To create a more interesting world, we decided to implement a grass shader.

A grass shader can be created be utilizing the Geometry Shader. The Geometry shader resides between the Vertex Shader (or the Tesselation shader if active) and the Fragment Shader. This shader program receives a certain number of primitives, and uses them to create new geometry. While it isn't as scalable as the tessellation shader, the geometry shader allows developers to create things, others shaders can't.

A grass shader is a shader that tries to recreate grass, either small patches, or long fields of grass. The biggest two problems grass shaders face are optimization and realism. A grass shader can be implemented in multiple ways. The first way is by using billboards.

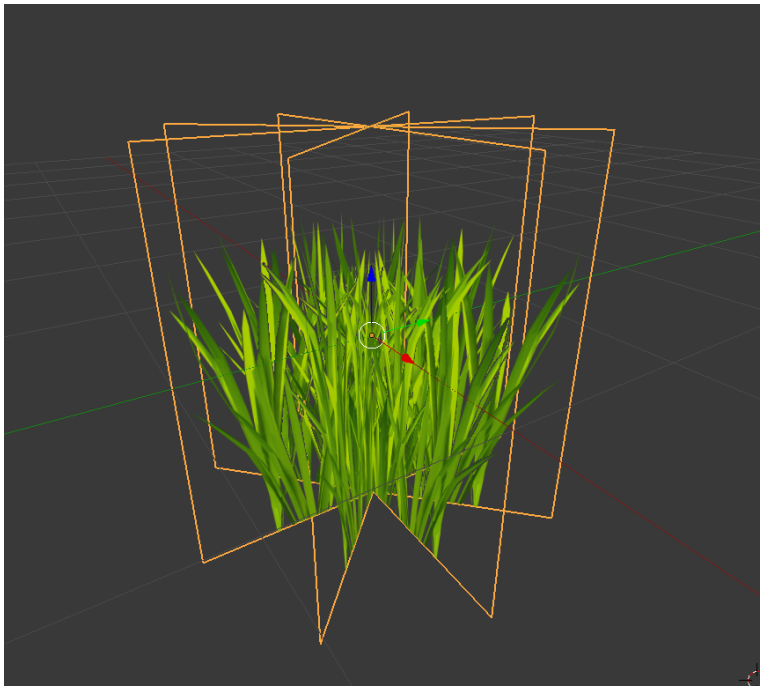


Figure 13: Grass created with billboards

In this implementation, we generate one or more squares. Each square will be created around the y axis, and then a texture will be attributed to it. This method can create good results, but has several limitations. Wind animations are hard to simulate and usually don't produce good results. The quality of the grass is also limited by the quality of the texture.

The next two implementations are similar. They take a central point, and generate points around it, in order to form a triangle. This triangle can then be as complex as necessary.

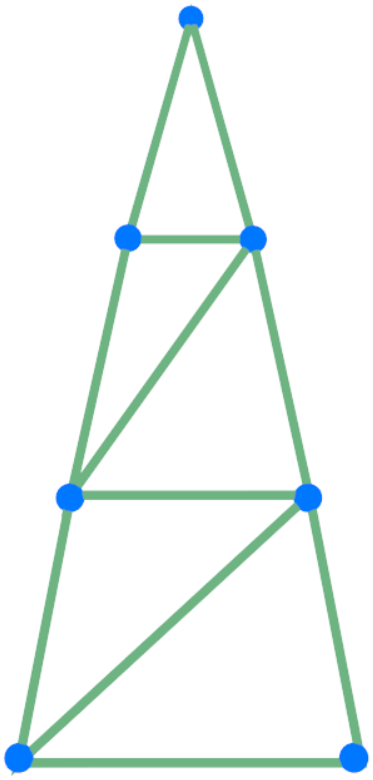


Figure 14: An example of the points requires to create a triangle

The biggest difference between these two is where they get the points from. One of the implementation receives the primitives given to it, and randomly selects a position inside these primitives. The amount of selected points can then be changed in order to optimize the program. The second way (and the way we did it) was by only using the primitives given by the vertex or tessellation shader. The biggest difference is that this implementation relies on the tessellation to create a enormous amount of close points so it looks like there's a lot of grass. To optimize the first implementation, the number of points generated has to be controlled, while on the second implementation, we can optimize by telling the tessellation shader to create fewer triangles if the camera is far away. This means that the ground near the camera position has much more triangles, than the ground far away from the camera position. While the have similar performance, we chose the 2 approach because we were already using a tessellation shader active, so we decided to use it. While it does in fact, create enough vertices to simulate big patches of grass, there are some visible problems.

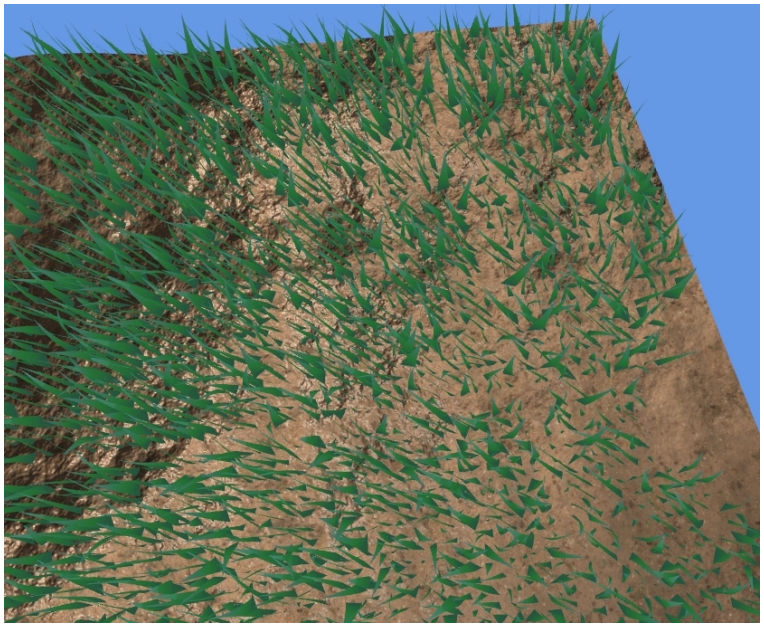


Figure 15: An error in our implementation

As we can see, as we get higher, and consequently, more distant from the ground, the tessellation shader creates fewer divisions. This means that only the plane points are used to create grass, creating a field of grass with a very geometric distribution.

As a side note, it's in this shader where we pass a boolean value onto the fragment shader which will let it know whether it's processing a grass fragment or a terrain fragment. This value had to be marked with the identifier *flat* so it wouldn't be interpolated and instead kept as a boolean value.

3.1. Implementation

The logic behind our algorithm is simple. Firstly, we need to find the tangent matrix. This is because we want to able to modify each blade of grass rotation, direction, curvature, etc, without having to consider the leaf's angle (or the angle of the terrain from where the leaf is being generated from). To create this matrix, we need the primitive normal and tangent, so we can calculate the binormal. With this three variable we can then create the tangent matrix. After creating this matrix, we need to create several matrices and generate random values. We created one for the random direction of each leaf, one for a random forward curvature and one for the wind. We also generate randomly values for the blades width, height and curvature. The wind is also a big part in creating a realistic grass. The way we did it was by using a wind map (similar to an height map). In this map, we got a value for both x and y axis (based in the primitive's uv coordinates), which then will be used to identify the wind's direction. The wind matrix will then rotate the leaf in that direction, with a (modifiable) force.

After calculating this values, is time to generate the points. Since the geometry shader is also responsible of drawing the geometry of the terrain, we first have to create the triangle of the terrain. But then we can finally create the vertices of the leaf. Just as previously shown, the leaf is constructed in "subdivisions". The number of subdivisions can be altered, and it increased the realism of the blade, but cost more performance-wise.

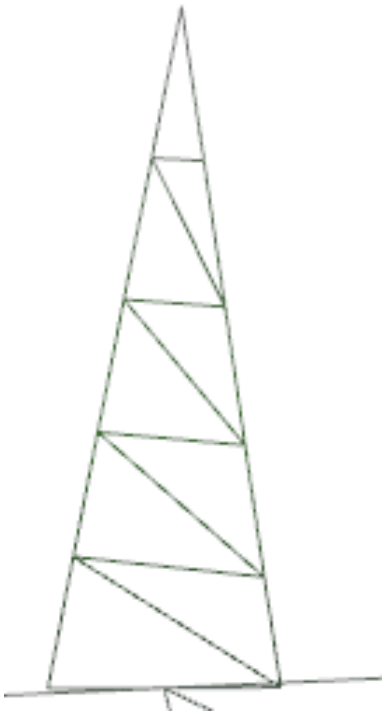


Figure 16: A leaf generated by our code

In this figure, we can see a leaf generated by out code. As we can see in the figure above, we indicated the geometry shader to create 5 subdivisions for each blade, which results in a total of 9 triangles per leaf. But since we need the leaf to be seen in both sides, we need to draw the same leaf, but declaring the points in a different order. This way we create a leaf that is drawn to opposite directions. But this also means that the number of total triangles created duplicates.

But since we also want the blades to be colored, we give each leaf vertex, uv coordinates. This uv coordinates will then be processed by the fragment shader. We also pass the wind strength (we will explain shortly).

3.2. Fragment Shader

The only job the fragment shader has for the grass is painting each leaf. Since each leaf vertex gives a uv coordinate, the fragment shader only has to select the corresponding color from a texture.

Figure 17: The texture used to paint the grass

The texture used was created specifically to color leaf blades. It starts with a black color, fading to green, and then fading to white. This texture can be used to give a "lighting" effect to the blades, since the bottom of the leaf is usually darker, while the top is much brighter.

One way we can better emphasize the wind passing through the leafs, is by highlighting the top of the blades. Since we made the geometry shader pass the value of the current intensity of the wind, we

can check if the pixel in questions is part of a leaf currently being blown by the win. If in fact it is, the fragment shader will intensify, slightly, the color of the tip of the blade. Even though its barely noticeable, this effect can improve the feeling of the wind passing through the leafs.

For fun purposes, we also decided to allow the colors of the grass to be changed in run time. Changing colors of the grass can be a fun way to mess with the scenery, since changing the color of the grass can change the aspect of the terrain significantly.

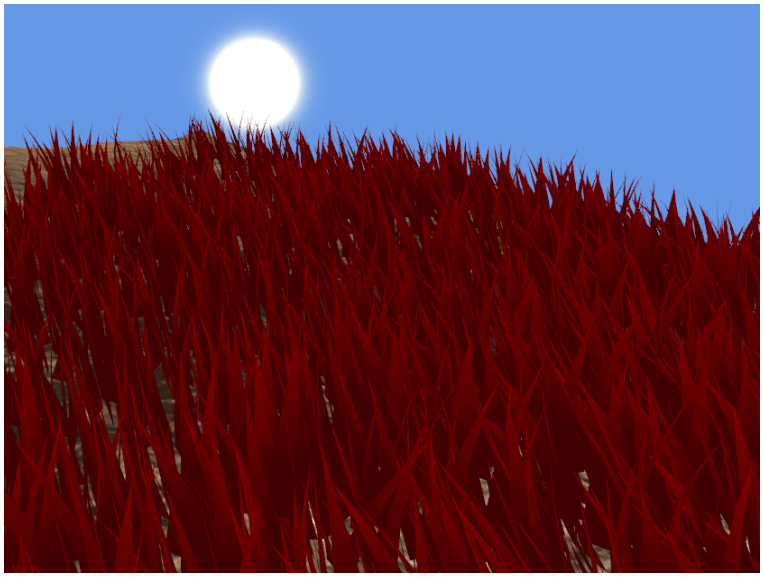


Figure 18: Red grass

Figure 19: Light green grass

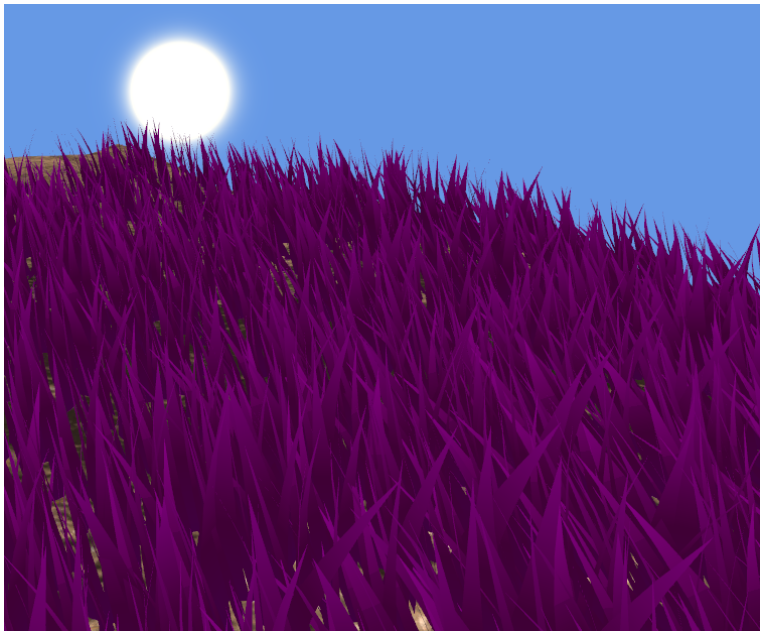


Figure 20: Purple grass

4. Skybox

Since the procedural terrain felt out of place with nothing around it, we created a skybox to fill everything around it. Because of the infinite procedurally generated terrain we created that would move along with the camera, we made the vertices be rendered with the depth as the maximum value of 1 and removed any translations from the view matrix so that these would follow the camera.

Originally, we used a cubemap texture for it but we wanted more control over it. For this we made a fragment shader that would draw the sun dynamically according to the light direction by simply inverting it and using it not as a direction, but as a position of the sun relative to the camera.

With this the screen position of the sun is calculated and passed onto the skybox's fragment shader which then uses a gaussian distribution to create the sun. We tried ways to render the sun such has a simple circle that's either completely white or not, a fade based on pixel distance, but the gaussian distribution was chosen due to the strong inner color because it goes past the value of one yet it still creates a nice outer fade.

Figure 21: Sun with a strong Y direction

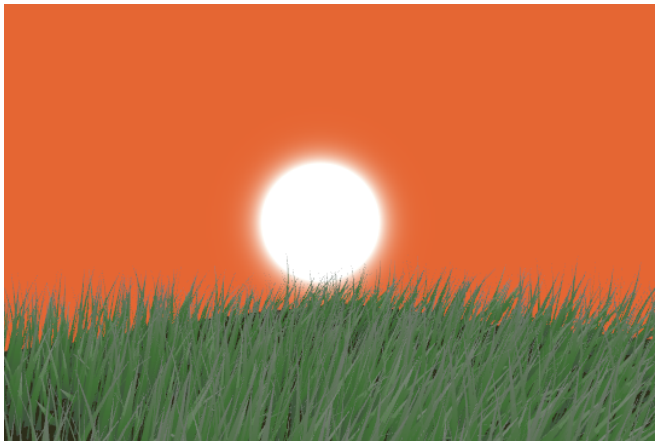


Figure 22: Sun with a Y direction of zero

Since it was easy to do, we have two different colors which are mixed with the Y component of the sun's light direction. It's far from being realistic and we weren't planning on doing it but it's a very simple add-on with the way it was implemented.

The sky's color also fades into a slightly darker color as the distance from it to the sun increases, although with the current values we chose it's not very noticeable.

5. Post Processing

To make the rendering better looking, we decided to implement post processing along with the project. The objective was to simply put the terrain and grass in a better light by emphasizing them and not outshine them.

5.1. Screen Space God Rays

With this intention, since we had a sun already, we thought that it would look very good to have the grass and mountains interact with the light coming from the sun. God rays are scattered light that are very susceptible to objects standing in the way of the viewer and the light source. For this, we chose to implement a screen space approximation of this phenomenon.

By getting the result of the rendered geometry as a texture and passing it into a new fragment shader, because we had used a gaussian distribution for the sun and chose values that make the inner part of the sun very strong, its color values have a strong intensity which allows us to get the corresponding pixels of the sun by performing a intensity threshold on the texture.

$$sun > red + green + blue + threshold \tag{2}$$

This is a simple way of doing it and it works perfectly in our case due to how strong the intensity of our sun is. The *threshold* value is there to differentiate what could just be a non clamped strong specular component from our sun.

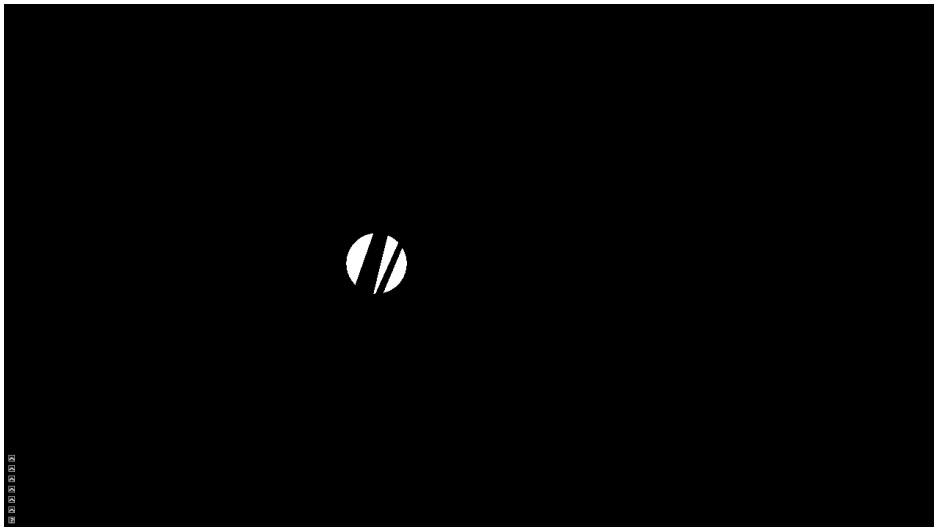


Figure 23: Texture with the threshold applied to get only the sun values. Notice how it's occluded by two grass leaves which will play a big part in the next steps.

Now we have this texture, the next step is to do for every pixel a line from it to the center of the sun. We know where the sun is in the viewport since we calculate this position in a script and send it to the shaders who need it. With this line, we sample the values along it and add onto an accumulator the color of the pixels sampled. Finally, by dividing the accumulated value by the number of samples, we get a ratio of how much sun exists in the line between the current pixel and the sun. This value is then attributed to a new texture and added on top of the render.

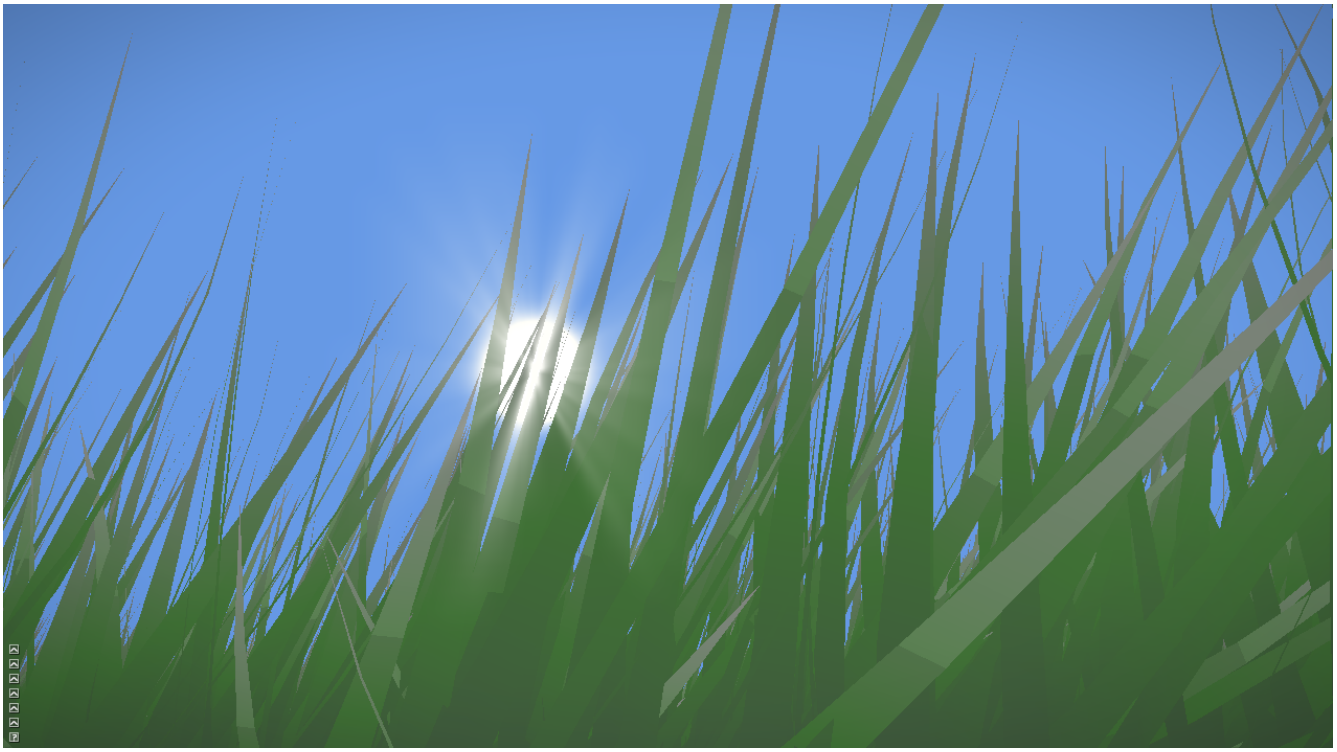


Figure 24: Screen Space God Rays layered on top of the rasterization render. Notice how there's streaks of light where there's less occlusion by the leaves and there's less light where it is most occluded.

This is something that's very expensive to do due to how many memory accesses are done especially without coalesced memory. A very simple optimization we did was to create the threshold texture with an eighth of the full resolution. The quality of the full resolution is not needed since we're simply handling white values that will be later added on to the full resolution render by upscaling it. The difference is definitely noticeable when comparing different resolutions, however, it isn't exactly bad looking.

5.2. Lens flares

To go with the post processing of light, we chose to do simple lens flares that would fill the scene more when the viewer is looking towards the sun.

This uses the same sun threshold texture as used for the god rays and the method isn't too different. It relies on flipping the texture coordinates every time it's going to process a new flare and depending on the number of flares, it adds distance away from the centre of the texture for sampling as an offset. As more flares are processed, the further away they get from the centre but also the bigger and fainter they get.

To improve the look, because it's simulating an effect of real camera lenses, light is often split into its many frequencies due to the curvatures of these lens. That's where chromatic aberration enters. By splitting the components onto its red, green and blue components and giving them an offset when accessing the texture, we are able to split the values onto the screen.

Figure 25: Lens flares layered on top of the rasterization render with godrays. In this image, there are 3 lens flares. The first one to be rendered was the smallest one in the middle. It's easy to see how if the texture coordinates were to be flipped in both axis, the sun would land on it, which is what is done. The second flare is the left most one and the final one is the right most one in the corner.

One thing that looks weird with this is when the sun is occluded by objects in the scene. We thought of hiding this by blurring the texture before adding it on top of the render, however, due to some problems with implementing the shader we were not able to.

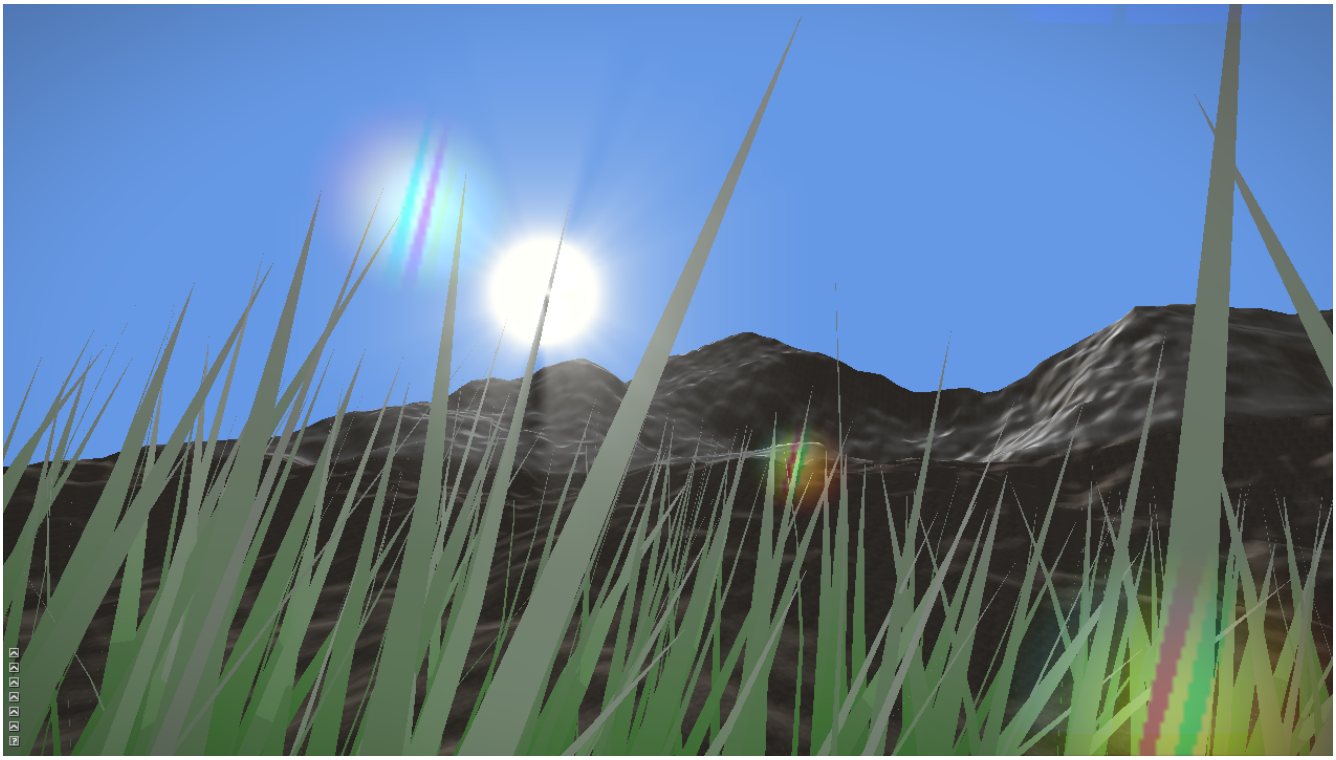


Figure 26: The chromatic aberration here due to how it's splitting the color components, with areas that are occluded, produces odd and noticeable colors. Pixelization also becomes very obvious since we're using a smaller texture for performance purposes.

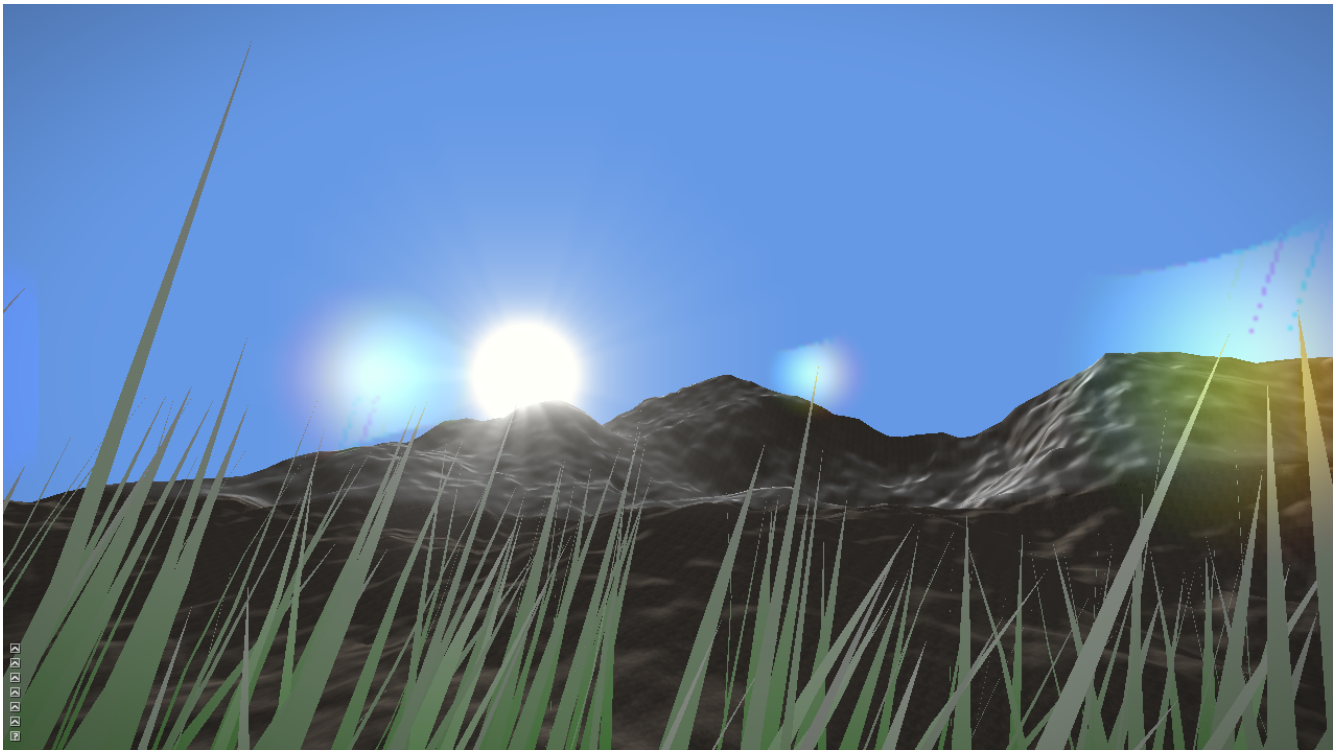


Figure 27: Same problems as before are visible here with the addition of a sudden cutoff from the sun being occluded by the mountain. This also causes pixelization of the lens flare.

5.3. Vignette

Something very small and probably unnoticeable but which, in our opinions does quite a lot for the render quality, is the vignette effect. However, despite being unnoticeable most of the times, when it's turned off after a while the difference is incredible and very noticeable.

As it's possible to see from the few last figures, the corners are slightly darkened. This effect is very simple to do because it's a simple distance calculation from the center and darkening based on that. It's an effect we applied in the same pass as the combining of all the different textures created by the previous post processing effects, so it creates very little overhead.

5.4. Grass Subsurface Scattering

Utilizing the sun screen position again, we thought it would look nice to make grass have a different shade of green when the sun was behind it. We do the same gaussian distribution on the grass but with much higher radius and falloff. This would cause an effect we refer to subsurface scattering only because that's what we were going for despite not being in any way how subsurface scattering actually works.

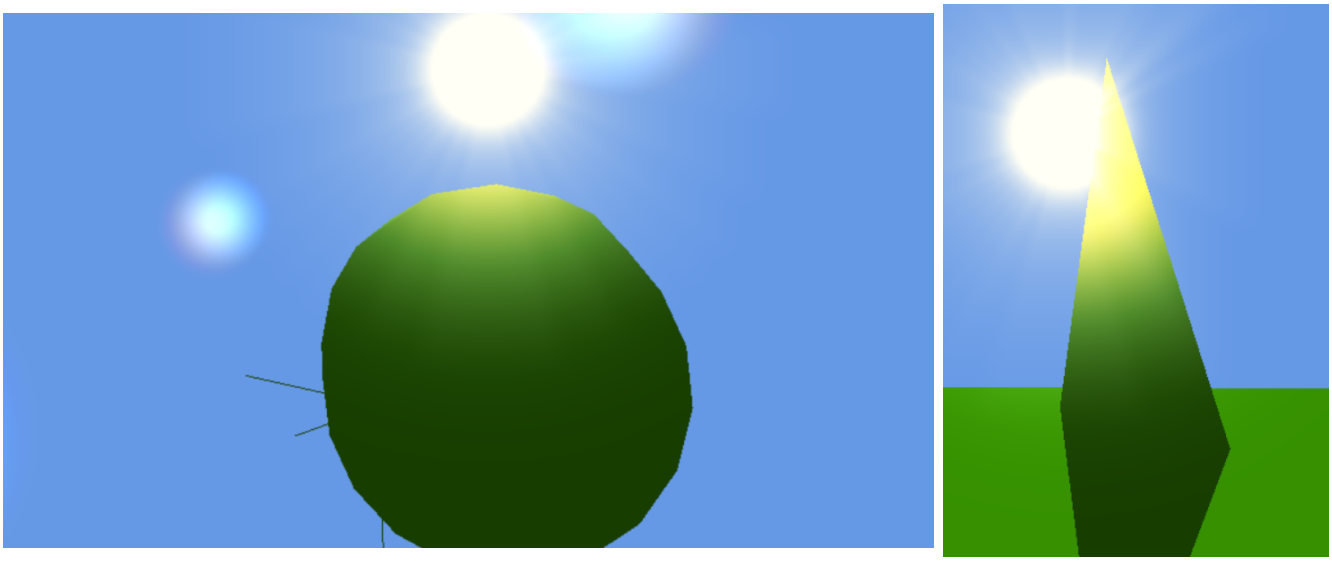


Figure 28: Old renders from the beginning of the project where light from the sun is scattering in the what at the time was going to become the fragment shader for the grass, creating a different shade of green.

Unfortunately, due to problems in combining the different stuff each of us made, we weren't able to add this in the final project, but it's still something we thought of, implemented and shouldn't be hard to add it.

6. Conclusions and Future Work

While we did manage to take some quite good looking screenshots and we were quite happy with the way the terrain generation turned out there are still some areas we could improve on if we had more time.

The scale of our terrain didn't quite fit the scale of our grass, should we have more time we would have added more grass at a different scale and we would change the way the grass is placed.

It's frustrating that we weren't able to put the sun onto the screen correctly with a script despite having been able to do it in the vertex shader. However, we were still able to show our post processing in action and that was only possible with the sun screen position being calculated and shared by the script.

And finally our terrain generation's performance could be heavily improved by pre-calculating the terrain into textures in big chunks and calculating more when the camera gets near it. This would greatly improve the performance since we wouldn't need to calculate normal vectors in the fragment shader and could, instead, just load them from memory.