Ελληνικό Μεσογειακό Πανεπιστήμιο
Σχολή Μηχανικών

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Πρόγραμμα σπουδών – Μηχανικών Πληροφορικής Τ.Ε.

Πτυχιακή Εργασία

Υλοποίηση ενός FPS 3D Adventure game σε Unreal Engine

Τροχίδης Ευθυμιάδης Βασίλειος - Α.Μ. 4377

Επιβλέπων Καθηγητής: Παχουλάκης Ιωάννης

Contents

Image Table

Abstract

Introducing techniques to create a game environment with AAA graphic techniques for optimal performance and great quality in graphic fidelity is the concept of this thesis. We will first understand the Unreal renderer that we will use and the performance tricks we will need to go through, to get the maximum performance, and we will introduce concepts such as Stochastic Triplanar, Volumetric lighting and Virtual Textures. We then go on how to create a robust pipeline for asset creation, and last, we will explore some of the gameplay logic and code of the AI and the player. The game consists of 3 missions for the Player to explore the map and experience the environment.

Introduction

For the current thesis we are going to develop a game in Unreal Engine 4.25.1 and creating a realistic environment with many different shading techniques and optimal performance. Techniques for shading the terrain, spawning procedural foliage, creating an AI, and developing a mission system will be the essence of this thesis.

Motivation

The motivation for developing this project is to explore some of the techniques used by big game studios in their pipelines that allow them produce stunning results, with great performance. Learning Unreal is a huge asset in the game development world, and it is a great skill for pursuing a job in this field. Along with Unreal and creating this project many other programs were used to create this result. Gaining experience in 3d modeling and texturing was a plus from this thesis, because those tools help us also understand how things are being done, what are the best practices to create assets for games, and how to make things easier for further development to be used from other program down the pipeline. In conclusion learning the Unreal Engine and game development in general was the goal and motivation of this thesis.

The goal is to create a realistic environment for a game, while maintaining good fps results (frames per second), and also creating a small gameplay example like we did with adding missions to the game, to make it more interesting and fun for the player, and give him a motivation to explore our small world.

Thesis Structure

On this report first, we will analyze how we the deferred rendered of Unreal works, what are the limitations of the engine performance and some optimization we can do to help that. Then we are going to see what are the shading techniques we used to render our landscape, how we handled the landscape optimizations, how we populate the world with foliage in a procedural way, and how we render our grass. Then we are going to see how we utilized the Virtual Texturing system, to blend meshes with our landscape. After that we are going to analyze the lighting in the scene, and we choose the dynamic lighting. Next we are going to explain how we created the AI in C++, and how the behavior trees work, we are going to see the pipeline used to import custom assets to the engine as well as how we imported and rigged the player and the functions he can perform in game. Lastly, we are going to touch just a little the topic of particles and the Niagara system.

Implementation methodologies

In this section we are going to discuss about the programs used in this project, like blender and Quixel Mixer, what they do, and how we combined them together and used them in our pipeline.

- *Unreal Engine 4.25.1*
- *Blender 2.83*
- *Zbrush 2020*
- *Quixel Bridge*
- *Quixel Mixer*
- *SpeedTree*
- *xNormal*

Unreal Engine is the game engine we used to create and run our game. Is a very good solution for creating small and big games, has a wide variety of tools to offer to the developers, making it a very appealing option for new developers and indie studios. Has great performance in rendering and we also have the option to modify the engine code and shift the engine to fit our needs. The version used in this project is the 4.25.1.

Blender is great open source 3D modelling tool. It is free with frequent updates to keep up to the industry standards. Unreal often recommends using Maya as the main 3d modeling tool, but recently has developed tools, available on GitHub that make the process of linking blender with Unreal a lot easier and essentially create an official support for blender. Blender is used for creating models, and for animations.

Zbrush is the industry standard sculpting tool. Used by almost every big studio out there. We used it for create rocks, that we scatter across the whole terrain and as well for sculpting small details on surfaces and creating normal maps out of those.

Quixel Bridge comes free for users who are working with Unreal. It offers the whole Megascans library for free. Asset like 3d cliffs, and rocks, foliage, and surface textures are all mainly from the Megascans library and imported with the help of the Bridge tool.

Quixel Mixer also from the Quixel team is a great free solution for texturing our models. We used this tool to texture all our imported models. Has support for Box projection of the textures, masking the model with a variety of options like noise, curvature of the model, normals of the model and many more. Also offers support for material ID masking, custom normal maps import and a very robust texturing process.

SpeedTree was used to create the trees in our scene. Has a great support for create different variation of the same type of tree, very fast and easy to use. SpeedTree used along with blender for modeling the branches and importing them to SpeedTree.

xNormal is one more free software that was used to create high detail normal maps for all our model. By providing a high poly mesh and a low one, we get the normal map of the model.

Unreal Engine Deferred rendering pipeline

On this project we used the deferred renderer that Unreal Engine provides us along with the forward renderer. The forward shading is only used for the translucency effects. The main difference between those two is the way they calculate the light. First, we will look at the forward rendered.

This shading method does all the calculations in one pass, sort of, but the main problem with this shading method is that in that one pass it gathers all the geometry data, runs the vertex shader pass and then for each light that we have in the scene (we have limited number of lights) , at the pixel shader or the fragment shader we calculate for each pixel or fragment the light for that specific pixel, and we do this for all the lights that we have in the scene and for all the vertices and pixel, even for that ones that are occluded even though unreal does a run a cull pass to optimize as much as possible the overdraw that this shading method is introducing because of its nature. Now if we have many lights in the scene these calculations must be done for each light, but in this shading method and with unreal we can only use up to four overlapping shadow casting movable lights in the scene, we can't sample from the GBuffer anymore (will explain the GBuffer later), we also have less texture samplers for our materials, and of course we cannot use Screen Space Techniques such as SSR(screen space reflections), SSAO(screen space ambient occlusion), and MSAA on D-Buffer decals. But it also has some advantages, such as, good anti-aliasing methods like MSAA which can be faster than TAA(temporal anti-aliasing) that unreal offers, it is fast when we don't really want many lights in our scene (even faster the deferred shading), we can render semi translucent objects, something we can't do with deferred shading and we actually use the forward pass to render that while we use the deferred renderer.

Now in our project we used the Deferred Shading Renderer, which is a common shading technique used for many of the modern games. The whole concept of this technique is that we let the lighting pass, which is a very performance intensive task, for a later stage. So, we essentially have 2 passes. The first pass is the Geometry pass where we write all our geometry data to the G-Buffer that we talked about earlier. The G-Buffer consist of six render textures as we can see in the DeferredShadingCommon.ush file that unreal provide as.

The first texture is the normal of all of our object that we see in the screen, and this texture has 4 channels, the first three are for the normals (the r, g, and g channel) and the alpha channel is for per object GBuffer data.

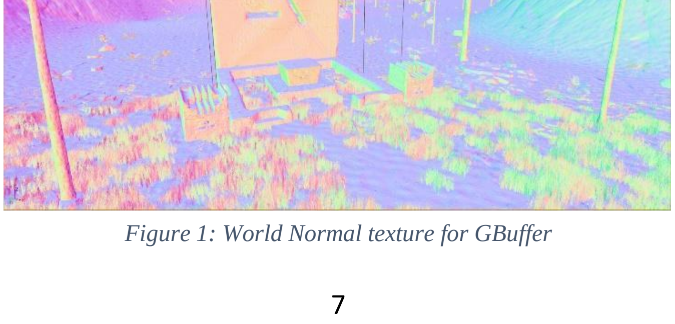


Figure 1: World Normal texture for GBuffer

The second texture consists of four maps packed into one texture. In the R (red) channel of the texture we have the Metallic information, in the G (green) channel we have the specular information, in the B (blue) channel we have the Roughness information and in the A channel we have encoded the ShadingModelID and the SelectiveOutputMask.

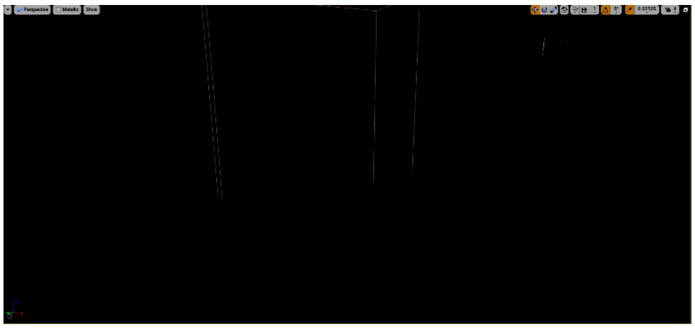


Figure 2: Metallic texture from GBuffer

Above we have the metallic information from the GBuffer, and as you can see it is black because none of the object in this scene are metals or the metallic attribute is set to 0 inside their material. Full white means it is metal and inside the material of the object the metallic attribute is set to 1 and full black means no metal at all.

Bellow we have the roughness texture and the specular texture, and both are represented as the metallic one, with one channel, each with values from 0 to 1 (black and white). In the roughness map 0 means the surface is smooth with no roughness at all and 1 means it is very rough, and the same goes for the specular map, and of course we can see values higher that one and lower that zero. And the shading model ID is to identify what is the shading model of that material ex. Purple means double sided foliage

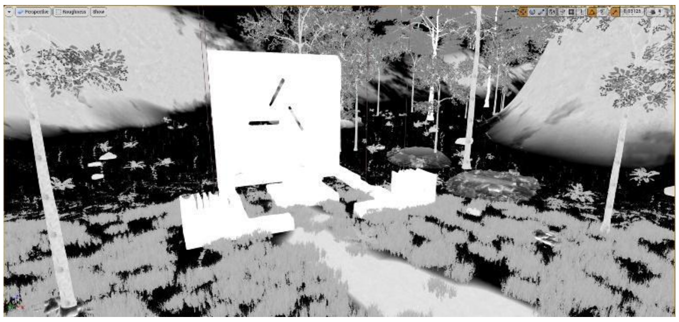


Figure 3: Roughness texture from GBuffer

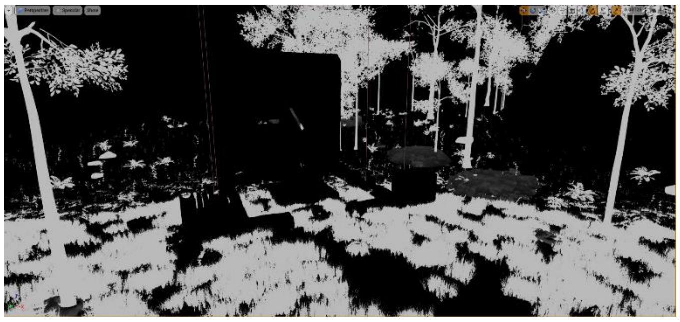


Figure 4: Specular texture from GBuffer

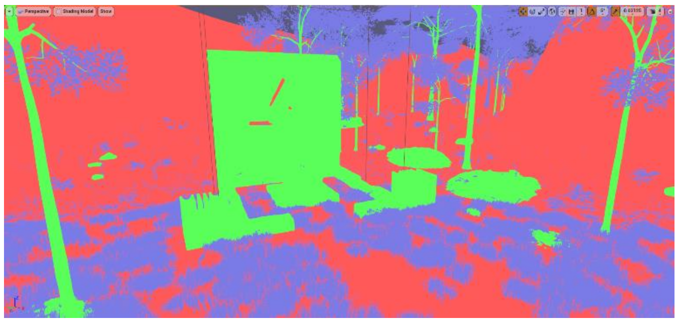


Figure 5: Shading Model ID texture from GBuffer

Third texture on the GBuffer is the base color information and the AO (ambient occlusion). In the first three channels we store the base color of our scene and the preprocessor flag ALLOW_STATIC_LIGHTING is false then we store the AO in the alpha channel, and we also set the indirect irradiance to 1.

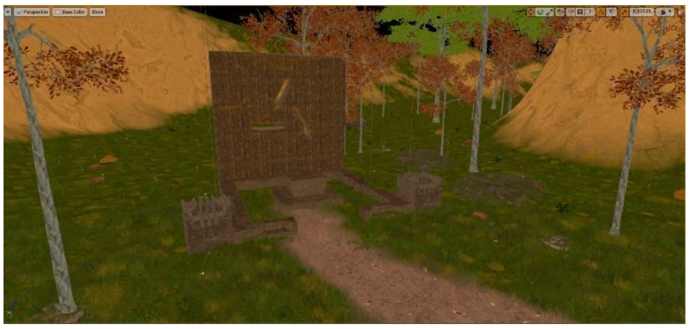


Figure 6: Base Color texture from GBuffer

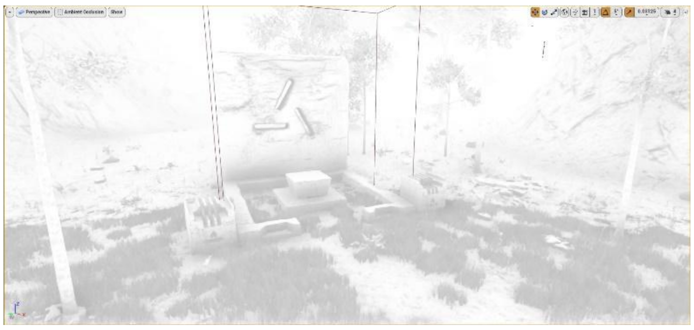


Figure 7: Ambient Occlusion from GBuffer

Next we have a texture for custom data that we can use to write our own data to the GBuffer, also we have one more texture for PrecomputedShadowFactors, and one more for the tangent normals if the flag GBUFFER_HAS_TANGENT is true. We also store in the GBuffer the Subsurface Color and the Depth of the scene in world units.

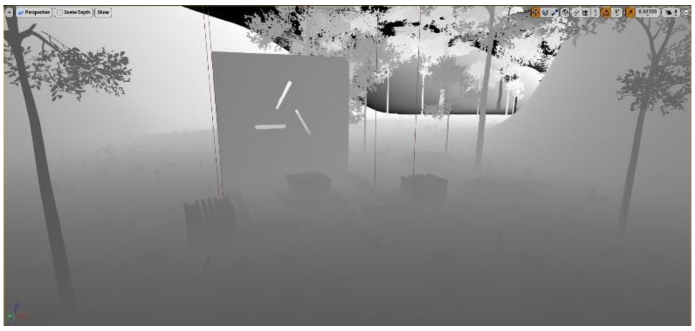


Figure 8: Scene Depth from GBuffer

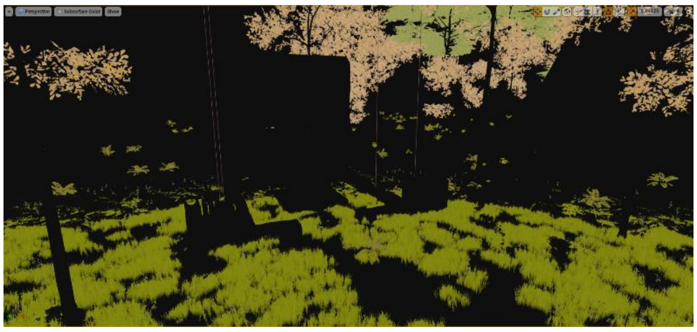


Figure 9: Subsurface color from GBuffer

So now that we have all our render targets store in the GBuffer, we can now go to the second pass, the lighting pass, and with the help of the GBuffer we can calculate the light. The difference now with this method is that having all the data of each fragment (pixel) we can feed them (normals, base color etc.) to the light algorithms and calculate for each pixel the final lit color. But we already have the final fragments that will be shown in the picture (from the GBuffer) so that means we only have to calculate the light for each fragment once, so there is no wasted calculations done on pixel that will be not shown on the screen. And with this method we can have as many lights as we want in the scene with a minimum impact on performance. Plus, we can sample the GBuffer render targets to use the in our materials.

There are some disadvantages also with this method, as with every method. Firstly, we cannot draw Translucency, that is why we use the forward shading method to render that because we cannot correctly blend the translucency with another translucency. We also cannot have a good anti-aliasing solution such as MSAA, so we fallback to the TAA which is a post process effect on the final image, and we also have a bigger memory overhead as this method requires more memory than the forward shading.

But there are many advantages also, like we can have post process effects in a lower cost such as the Bloom effect which is an effect when we look at the light directly we see a blur, we can also have screen space effects like SSAO (screen space ambient occlusion) based on the depth buffer, translucency can cast shadows to itself and other lit translucency. In general, when we have very simple scene with a few lights it is not that optimal to use the deferred rendering but when we have a complex scene with many lights the advantages are obvious.

So here is the final scene rendered with the deferred shading, which is rendered with combing all the above render textures.

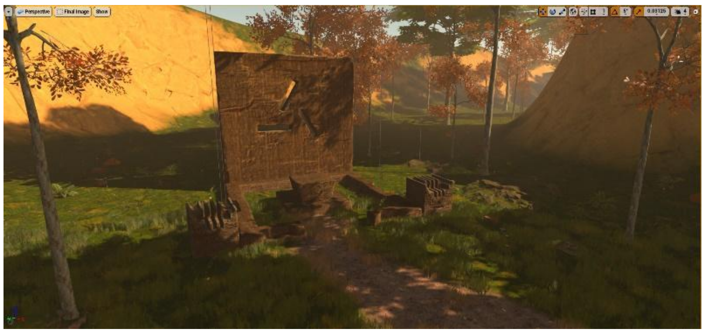


Figure 10: Final Image

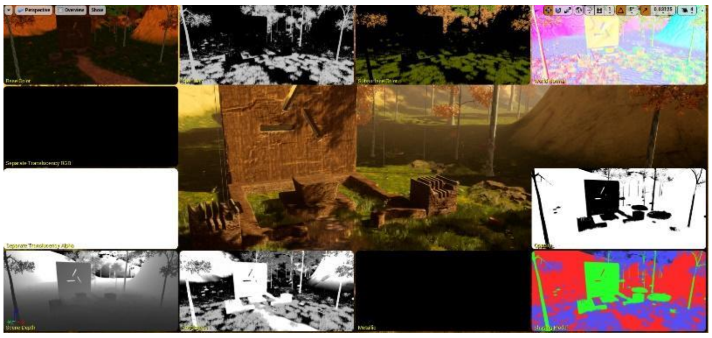


Figure 11: Final Image with all the buffer RTs

Optimizations and Limitations to think about

While working on this project we must take in consideration a few limitations and possible optimization we can try to avoid performance hits. The first “limitation” all though Unreal does not limit us, is the triangle count of the scene. Every model we import into the engine (props, static meshes, foliage, characters, etc.), we must think about the triangle count of that object, generally speaking for a AAA game the polycount we can target is about 4-7 million triangles (most of the high graphic games that run on consoles such as PS4, Xbox One are limited to 30fps), so we have to consider very carefully the amount of triangles each object has, and for that reason we create LODs (level of detail) for all of our meshes in the scene. So, what is an LOD?

All our objects are being apart of many triangles, for example a typical tree in our scene has about 18,000 triangles, that is the highest LOD of that mesh and we call it LOD 0. But if we want to render 100 of those trees, we will need to render 1.8 million triangles! That is a lot and that is only our trees. But when an object is close to us we want to render the highest mesh available for the best quality, but in the distance we can’t really see all of that detail and we don’t pay that much attention on the object, so what we can do is to render the same object but with less triangles, so we try to re-create the mesh as close to the original but with some sacrifices on the triangle count and the mesh detail. In our scene the same tree has 3 LOD’s, the highest has 18k triangles and the lowest on has about 6.5k triangles. Unreal handles the reducing of the triangles automatically by specifying how much of the percentages we want to keep. And the way we define which LOD we will render on the screen is by evaluating the screen percentage the object occupies in our screen, the lower that value is the lower the LOD we render on the screen.

One more technique we use to reduce the poly count is by creating normal maps. Normal maps are just 2D textures that we create in order to tell the engine how the light will interact with the mesh’s surface, in a way we are faking the light. When we have an object, like a rock , and we sculpt that mesh to create the desired look we want, that instance of one mesh may have more than 2 million of triangles, just a rock alone, and we can’t import that mesh into the engine and use it, because 2 rocks will take all of our budget alone to render them. So what we do is to remesh the mesh, that means we create a much lower in triangles mesh (about 5-15k triangles for a rock) that is similar to the overall shape of the rock and then we take the high poly rock and the low poly and we bake the normal of the high poly onto the low ones uv’s space normals. With that normal texture now, we can tell the engine in which way the surface is point to (normals of a surface) so the lighting will be calculated according to that information. Below is an example of normal bake of a high poly mesh.

So, let us take a default cube with 14 triangles. This mesh has already the UV prepared for texture projection that we are going to need to project the normal map bellow.

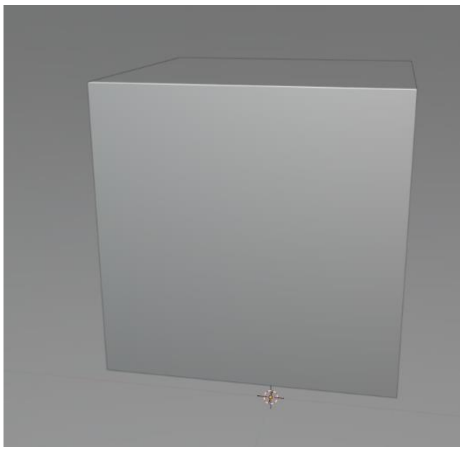


Figure 12: Default Cube with 14 tris

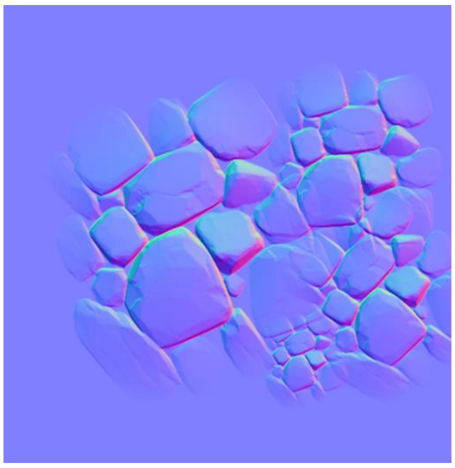


Figure 13: Normal Map of a rock shape baked to the UV's of the Default cube using xNormal

And above we have a picture of a normal map that we took by baking a high poly mesh into the low poly mesh above, the 14-triangle cube. First let us see how we got that normal map for the high poly mesh. Bellow we have the same cube from above but we remesh it to about 3.5 million triangles. We want that many triangles to sculpt the rock formation we see bellow, but from the other hand 3.5 million tris are that feasible to use in our scene. Using xNormal we can specify the low poly mesh and the high poly one, and it will bake into the low poly's UV's the details that we get from the high poly. And then we can use this map to tell the light how it will interact with the mesh.

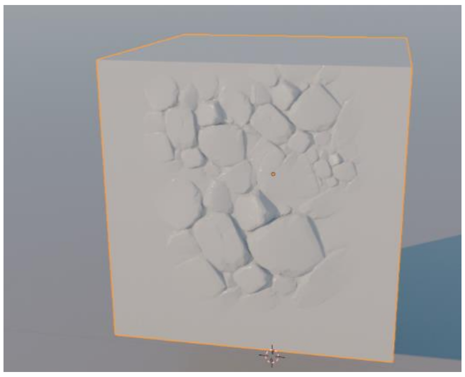


Figure 14: Sculpted Cube with 3.5 million triangles (rendered in blender 2.9 and Octane)

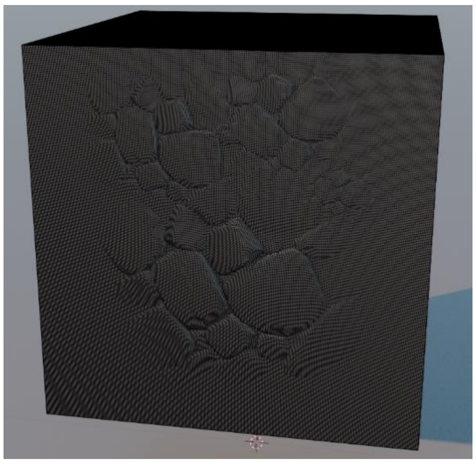


Figure 15: The wireframe of the mesh shown at the Figure 14

The problem with this much dense geometry is not only the count of the triangles that will be heavy to render for the engine, because this is a simple rock we may use it multiply times in the scene so the polycount may skyrocket, we also introduce another problem that we will take about later, and that is the triangles overdraw. Because of the size of the triangles is too small this problem comes to life, and the GPU has to work harder to render that object correctly.

So, if we get the low poly mesh now and apply the normal map, we got from xNormal we can see that the results are pleasing. The mesh from certain angles looks the same. The mesh still has only 14 triangles as you can see bellow from the pictures, this illusion breaks only if we look the mesh perpendicular of the normals direction, because this is only a lighting effect and there is no displacement of the mesh like the original one. In this simple step we got rid of the 99% of the original triangles, and we gain a huge performance boost, and we use this technique for almost all of the objects in our scene.

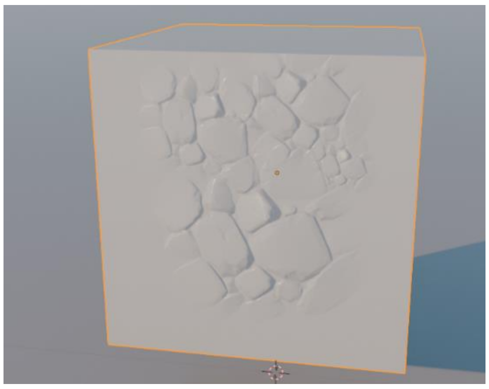


Figure 16: Cube with 14 triangles rendered with the normal map from figure 13

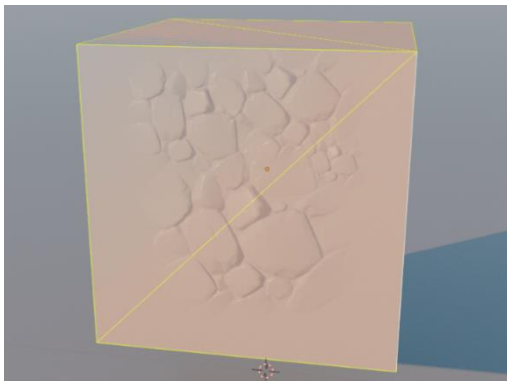


Figure 17: Cube's wireframe from figure 16

Overdraw

Overdraw is one thing to think about when it comes to performance. This is a common problem for the GPU, because the calculation are being done by quads (2x2 pixels), when we have very small triangles in the scene, or the triangles are that far away from the camera became very small proportionally to the screen size, so the GPU essentially is wasting those calculated pixels. Smalls triangles should be avoided but quad over draw can be produced in another way. Transparent meshes and overlapping meshes produce this problem as well as the GPU must render the vertices of mesh that is behind another mesh and then it needs to waste them and re-render the ones in front.

But there is a way to overcome both problems. Do deal with the small triangles in distance we use the LOD's. When a mesh is far away the triangles are still the same and that means they now occupy less screen space causing the overdraw to begin, because they are too small. So, when we LOD a mesh we basically lower the triangle count and that means now the triangles occupy more space because the overall shape of the mesh is the same, we just lose some details that we can't notice far away. But now with bigger triangles there is no overdraw. Now with the overlapping meshes what we can do is to enable an option in the engine settings the "early Z-Pass test", which is used as a culling method to eliminate what it shouldn't render in the screen and only render the final pixels, which it can create some overhead in the performance but if we have a lot of overlapping like foliage it is a huge gain in performance. Bellow we can see the overdraw in our scene.

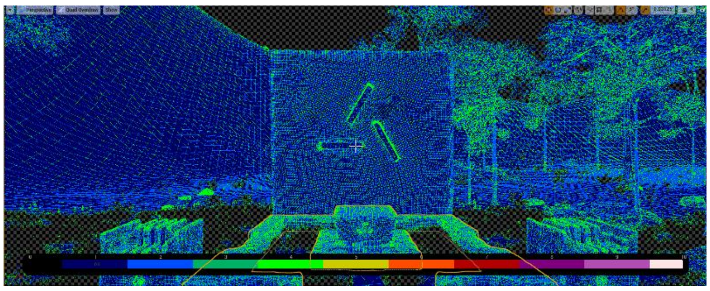


Figure 18: The mesh in the middle is at LOD 0 (highest detail) more and smaller triangles

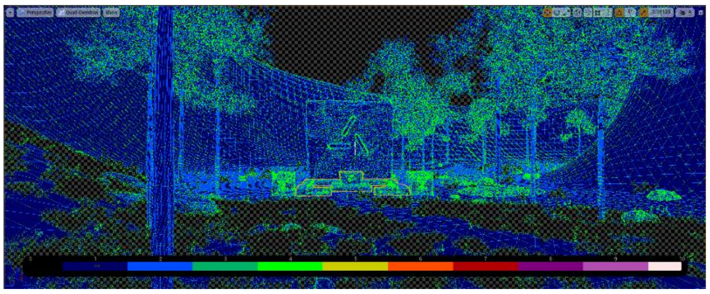


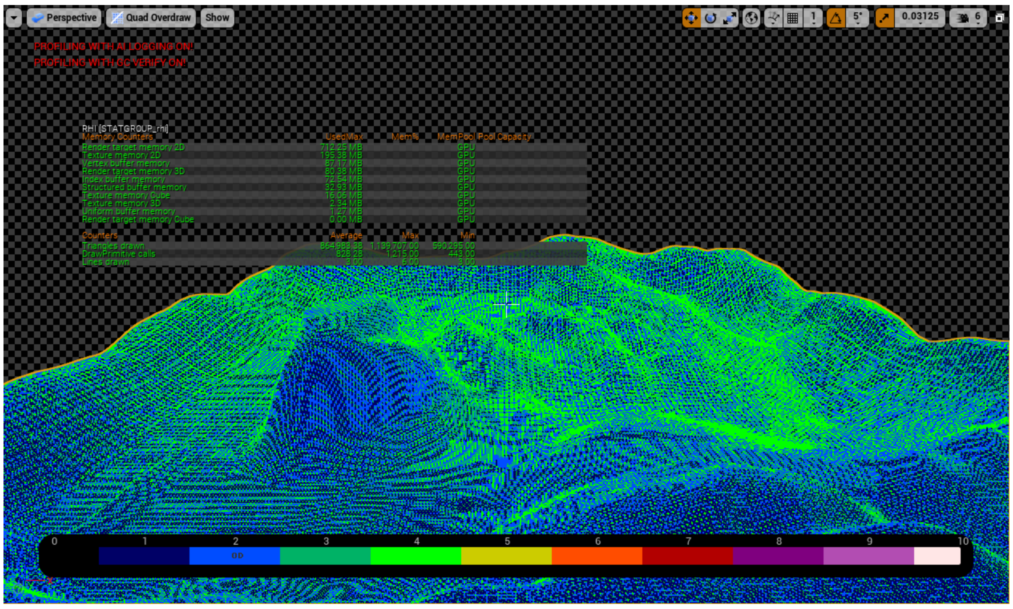
Figure 19: The mesh at the middle is at LOD 3 (lower detail) triangles are less and bigger

Landscape

The focus of this project is definitively the look of the game, o the graphics we can say. So, let us start with the landscape of the scene, the actual map that the player can walk, explore, and complete all the missions of the game.

First, we need to know that our landscape is made from many components, and in our case our landscape is made from 156 components. Now each component has 391x361 vertices that make up our triangles of the component. We need to keep the component count as low as possible because every component is a draw call to our GPU, and more draw calls to our GPU means poor performance, we should aim to about 3000-5000 draw calls for the whole scene (a draw call is a call that the CPU makes to GPU about the object the later one needs to render to the screen with the proper material, and that's why sometimes we combine the static meshes into one to reduce the draw calls as each draw call has its own overhead and if we multiply that overhead many times we get poor performance). Now our landscape has many triangles too, and as we talked before many triangles is not that good but the worst part is the overdraw that will occur here as the triangles that are far away will get too small and will cause the overdraw. But we can fix that too by specifying what LOD we want to render.

As you could see below here is the landscape if we rendered with the highest LOD all the way to the back. We render about 900k to 1 million triangles and we get a lot of overdraw for detail we do not really need.



```
PROFILING WITH AI LOGGING ON!
PROFILING WITH GC VERIFY ON!

RHI [STATGROUP_rhi]
Memory Counters                 UsedMax     Mem%   MemPool  Pool Capacity
Render target memory 2D        712.25 MB              GPU
Texture memory 2D              195.38 MB              GPU
Vertex buffer memory            87.17 MB              GPU
Render target memory 3D         80.38 MB              GPU
Index buffer memory             72.54 MB              GPU
Structured buffer memory        32.93 MB              GPU
Texture memory Cube             16.06 MB              GPU
Texture memory 3D                2.34 MB              GPU
Uniform buffer memory            1.27 MB              GPU
Render target memory Cube        0.00 MB              GPU

Counters                        Average         Max          Min
Triangles drawn              864,983.38  1,139,707.00  590,295.00
DrawPrimitive calls              828.28      1,215.00      443.00
Lines drawn                        3.00          6.00        3.00
```

Figure 20: Landscape wireframe and Quad Overdraw visualization

Now we will specify that we want to draw a high LOD (more detail) close and far we want to use a lower one. That will improve the tri count (down to 180k from 900k) and will fix the overdraw problem at the same time. Here is a picture with the tweaked settings.

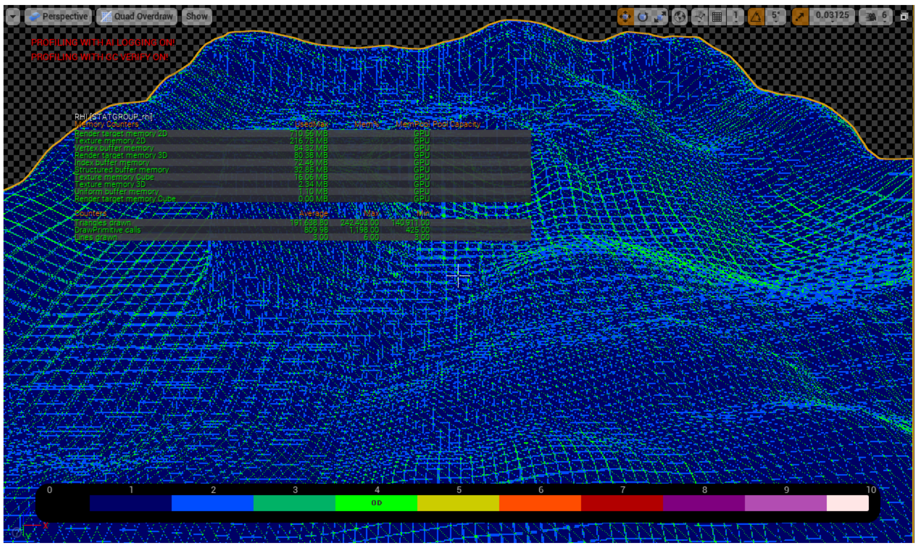


Figure 21: Quad overdraw with LOD's

And to visualize better here we have an image only with the wireframe of the landscape to illustrate the difference in the LOD's according to the distance from the camera.

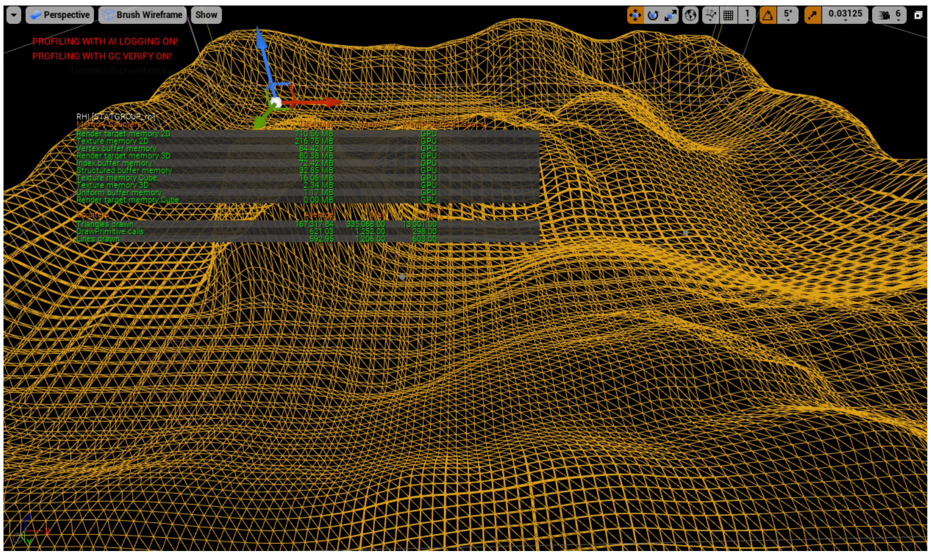


Figure 22: Landscape wireframe LOD's

Landscape Material & Stochastic Triplanar Projection

The next big part of this project, and one of the most crucial, is the material we used for the landscape. With the current material we have the ability to auto detect the slope of the terrain and shade it properly (detect and shade the cliffs), we can paint each section of the landscape with whatever material we want (grass, soil, gravel, snow) (Layers), and create fast different worlds that fit our needs, we can instantiate procedural foliage and specify in which layers we want to populate with what type of foliage (short grass, flowers and every other static mesh we want), we can map the textures in a Triplanar way but cheaper for better performance and at last we can use the virtual texturing system to store all of the landscape data to a texture that we can sample later on from different materials for other meshes in order to blend them with the landscape smoothly.

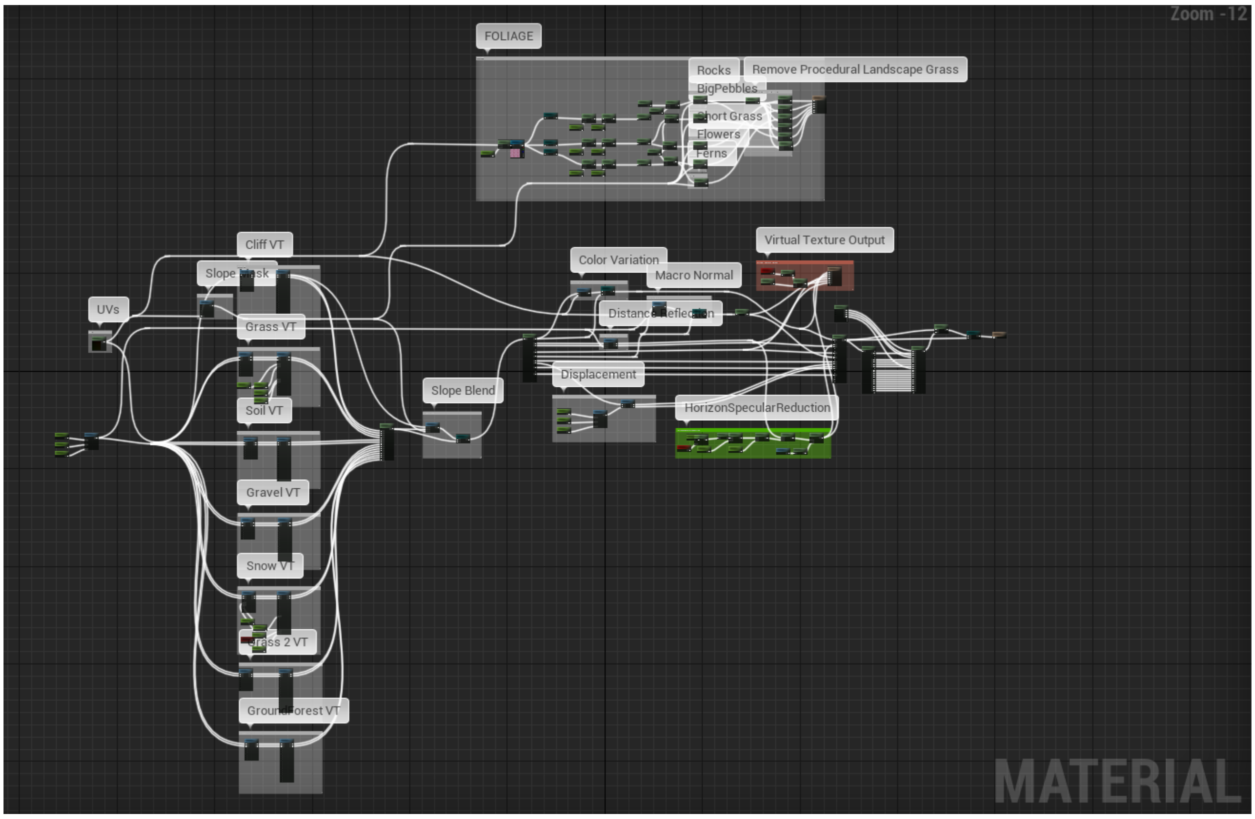


Figure 23: The landscape material

Layers

In the landscape material we use the concept of the layers. Each layer is basically a different material that we can then later use in the editor to paint our landscape with whatever material we want. In this material we used 7 different layers (cliff, grass1, grass2, soil, gravel, snow, ground forest).

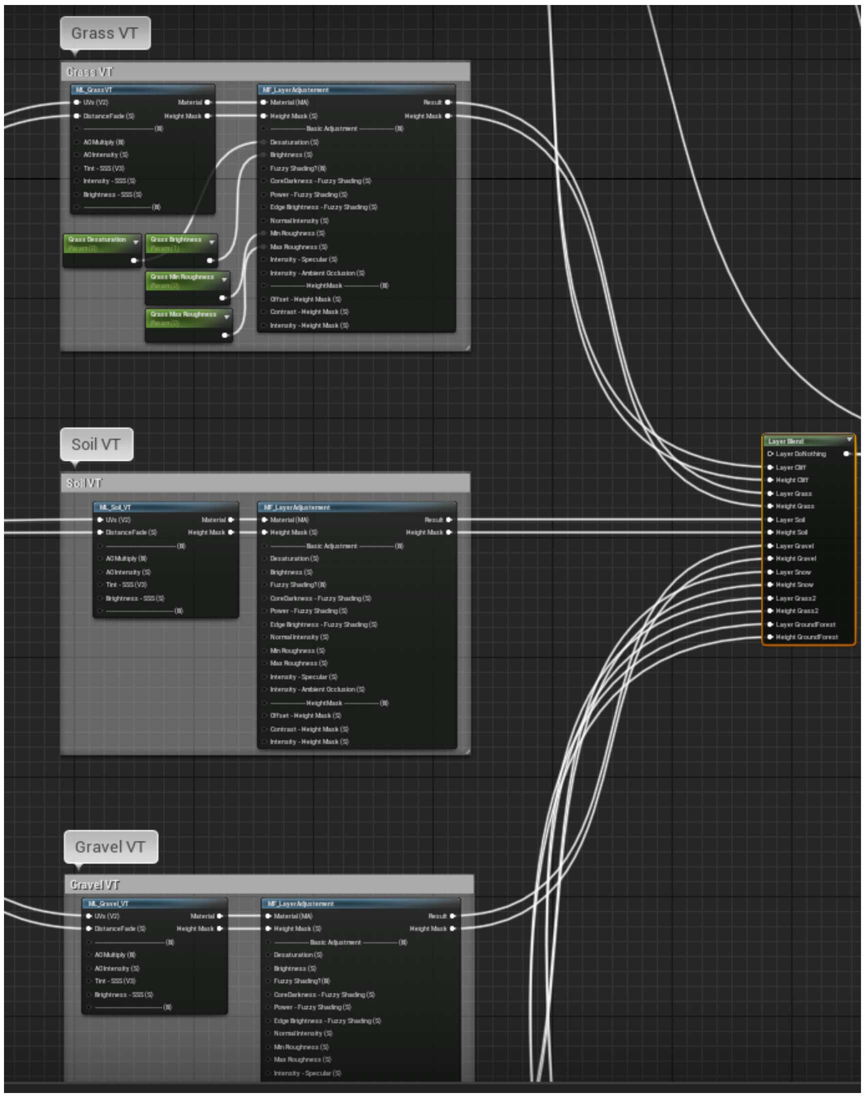


Figure 24: Different Layers for the Landscape material

At the right we see the node that takes as input all the layers we want to have, and at the left we see each layer like the grass and the soil layer. Each layer we see in this picture (figure 24) is a function that we must specify how each material is described.

Bellow we will see the grass layer, which is similar in with all the other layers because is a function, and the only difference with all of the other layers is the textures. So for each layer we have two functions (from left to right) the first one samples all of the textures of the layer like the soil albedo texture and the soil normal texture and the second function takes the output and make all the adjustments like the brightness of the final color of the layer, the roughness and many more.

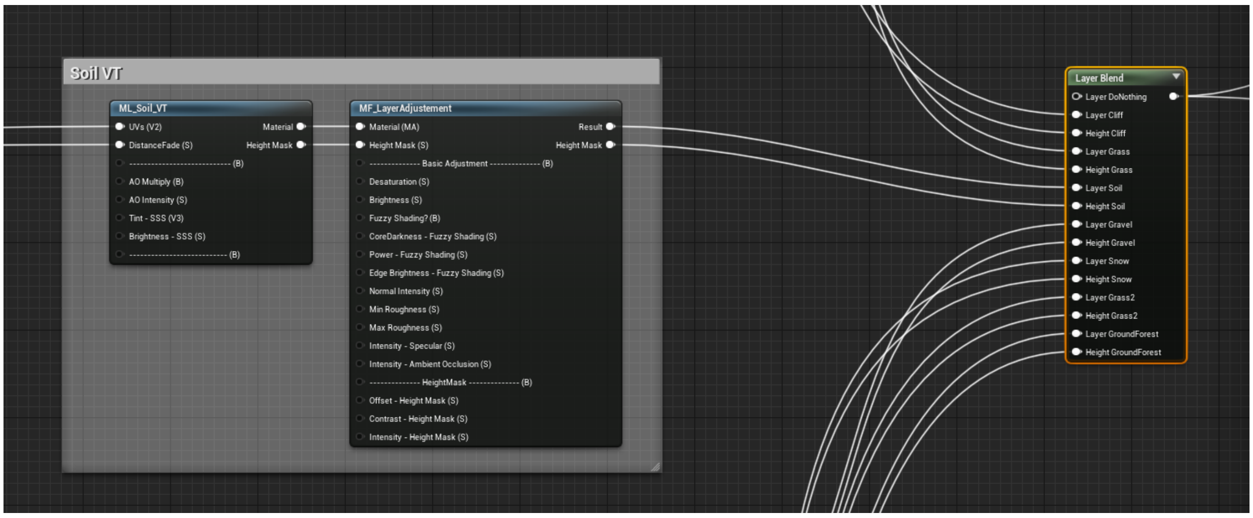


Figure 25: Example of one of the layers setup

If we dive in the MI_Soil_VT function, we are going to see all the texture samplers and more.

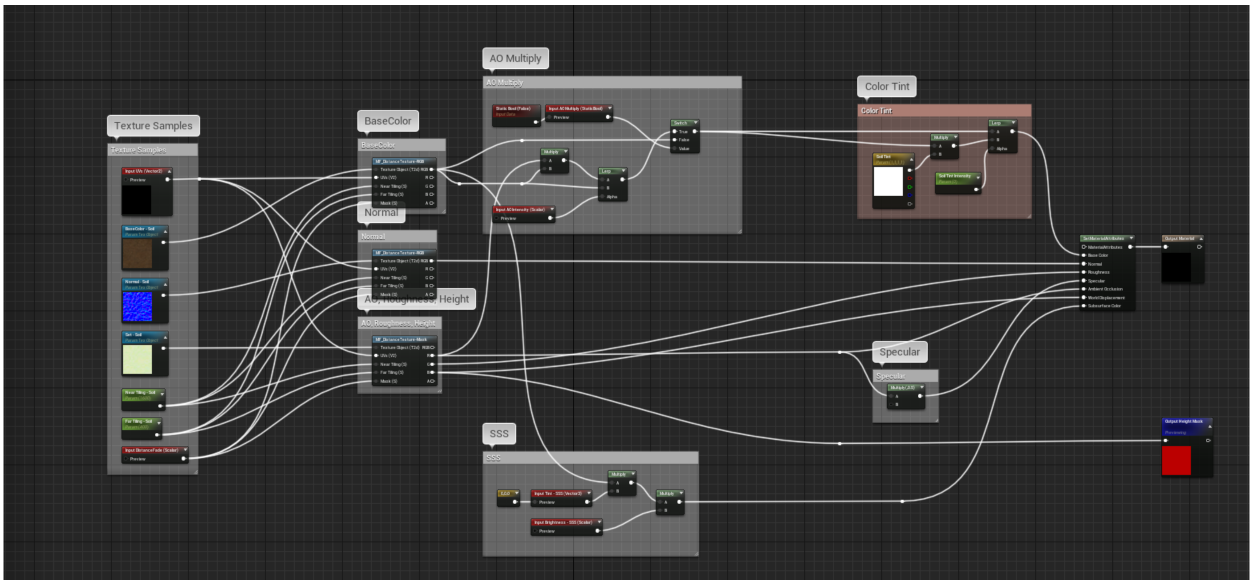


Figure 26: MI_Soil_VT function

At the left we see the 3 textures we sample for each layer, the first one being the albedo (color) texture, the second one is the normal texture and the third one is a packed texture containing 3 different information one in each channel. In the R channel we have the Ambient Occlusion, in the G channel we have the Roughness, and in the B channel we have the Metallic. At the bottom left we have 3 parameters the near tilling of the texture, the far tilling of the texture and the distance fade.

Now let us explain the need for those 3 parameters. To start, with a single texture for the whole landscape would be too small to fit across the whole landscape, and it would look very stretched. So, what we do is to tile the texture across the landscape, in other words repeat the texture repeatedly until we cover the whole space. Now we have two parameters for that tiling, one for near tilling which we generally want to be a bigger number than the far tilling because we want to keep the texture small enough to avoid making that big that all of the detail is gone because it is the same if we were zoom at the texture, it would look blurry from close. But when we are close, we cannot really notice the tilling, but far would be very noticeable effect. And that is why we have the far tilling which is basically a smaller number meaning less repetitions and bigger textures. That way we avoid noticing tilling far away and because its far away we can get away with the ‘zoomed’ bigger texture. The distance fade now it is a value which we use to calculate from which point and on we are going to use the far tilling while in the rest we render the new tilling texture. That technique is a little more expensive than using one tilling from the whole landscape because we essentially sampling the same texture 2 times, one for each tilling method.

For the rest of the material we extract the sub surface scattering from the textures, and we can use it if we want in materials like snow which the light penetrates the snowflakes and create the sub scattering effect. We also have the option to multiply the ambient occlusion and at the final stage we can change the whole tint of the texture to make it fit in our current situation. Bellow we see the tint effect on the grass layer.

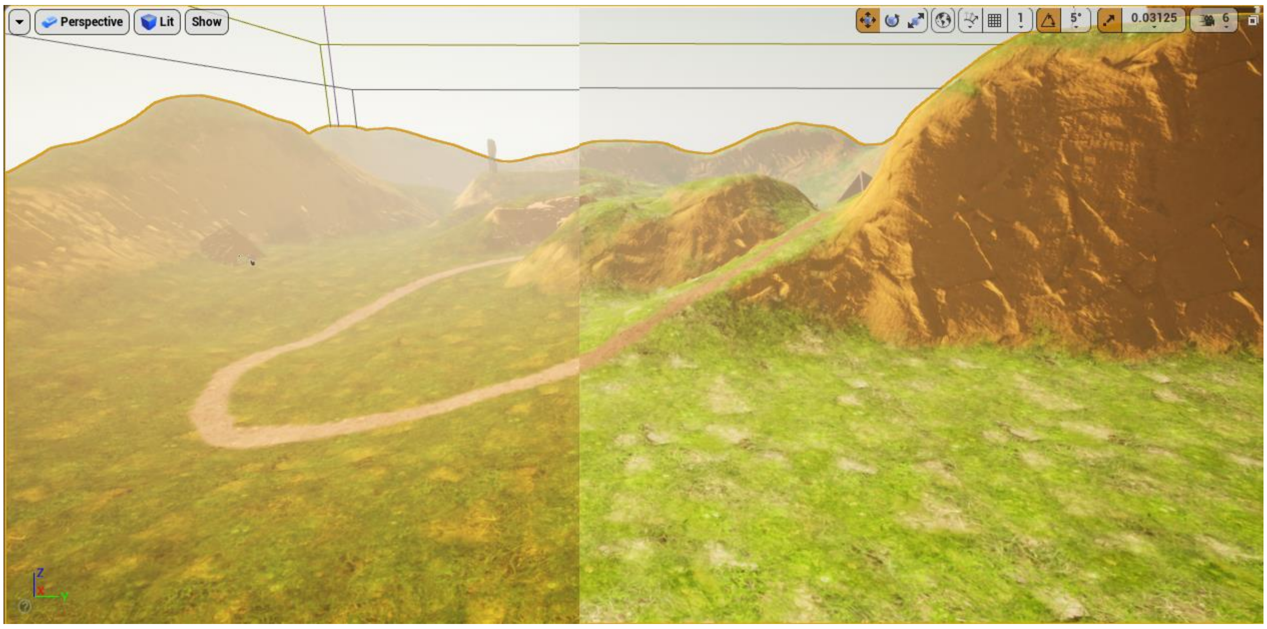


Figure 27: Tint visualization (left with tint correction - right original tint)

Next, we have the auto detect slope mask. With this function we can detect based on the angle of the terrain which surface has to be shaded differently based on the slope.

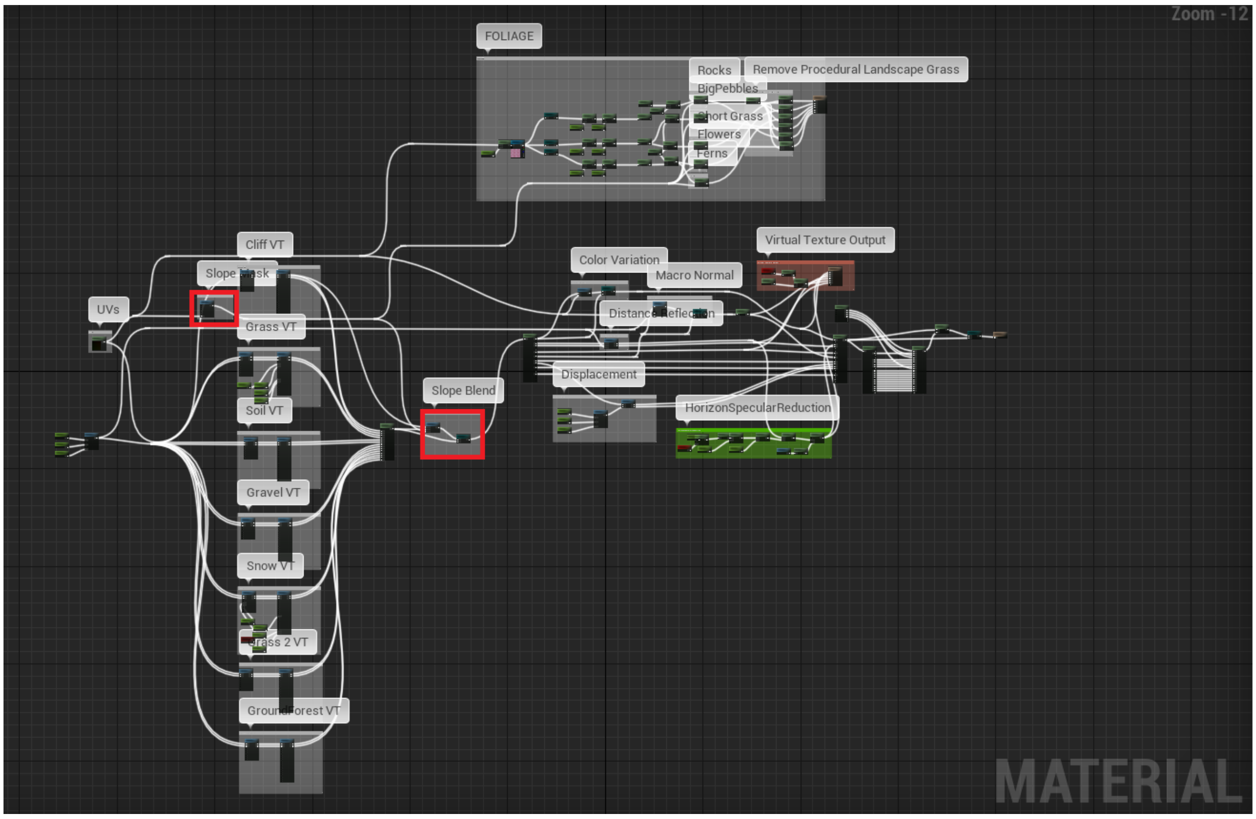


Figure 28: The 2 points at which we calculate the slope shading

At the left of the above image (Figure 28) we have the slope function detection MF_SlopeMask and at the right we get that result and we blend it with the rest of the layers (grass, soil, etc.). Now let us see how this function works.

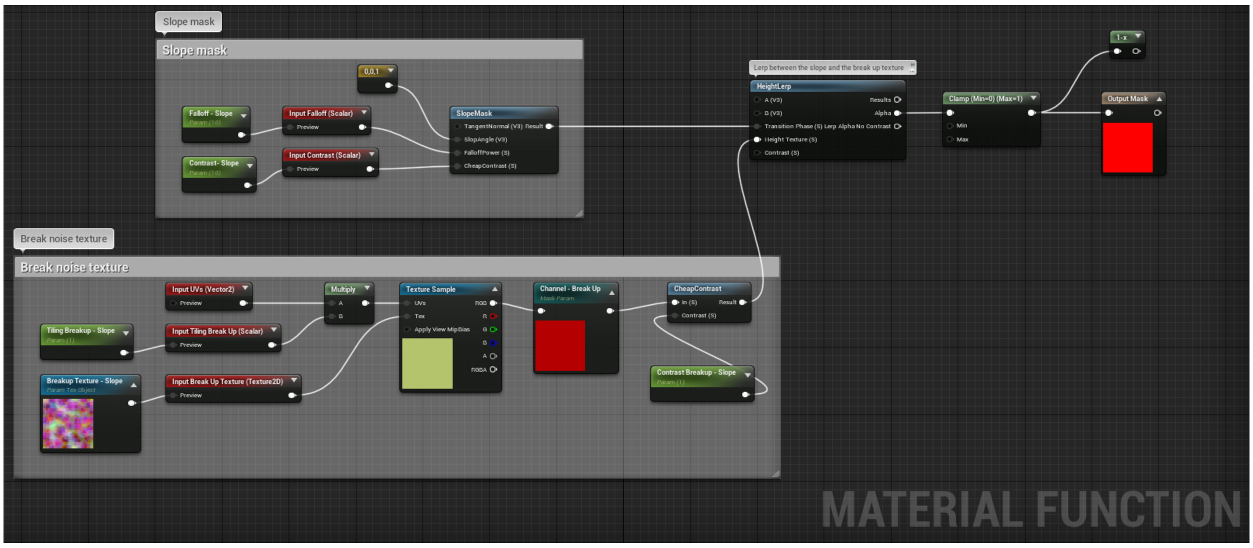


Figure 29: MF_SlopeMask material function

At the top we have the commented section "Slope mask" with two scalar parameters (scalar parameters can be changed dynamically without recompiling the shaders in runtime when we convert the material to material instance). This is using the Unreal's slope mask function and with the parameters we can configure the slope detection to our needs. At the lower part of the picture we see the commented section " Break noise texture" and we blend basically a noise texture at the end of the slope where the transition is being made between the slope shaded material and the other material in order to randomize the transition and not just be a hard line like transition which is won't feel natural. We have parameters to control the tilling of the breakup texture and a parameter to tweak the contrast of the texture (soft noise transition and hard noise transition based on the contrast value).

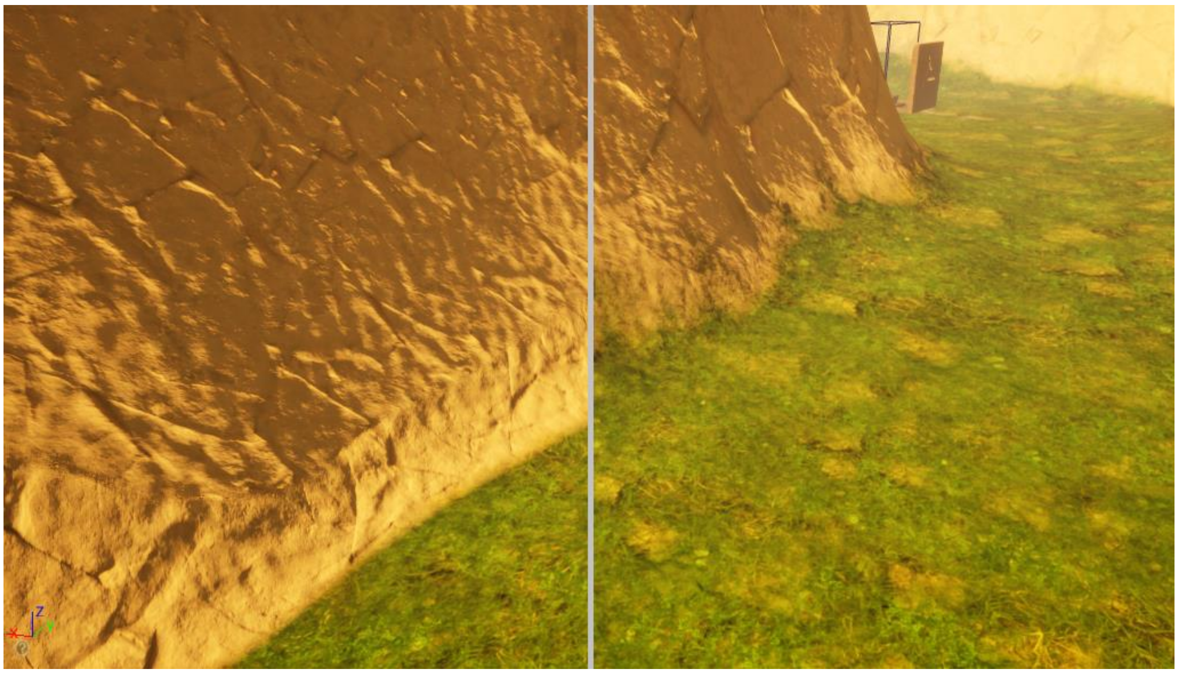


Figure 30: Slope break up (left no texture break up - right with break up texture)

Procedural Instancing

Now that we have our textures mapped to our landscape in different layers, we can use that data to spawn static meshes on the terrain. Let us start with the layers. Each time we paint a layer (material) to the terrain we basically create a mask (a black/white texture) for our terrain where we specify at which spot our current layer will be rendered. If there is white in the mask the layer will be rendered, if its black it will not be rendered, and all the in-between values will make it blend. But we can sample from these masks that exists for each layer, and we can use these values to control the spawning of the static meshes.

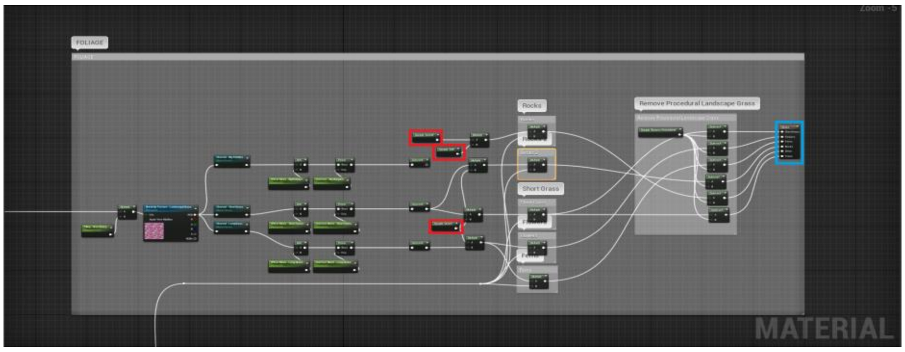


Figure 31: Part of the material shown in figure 23 responsible for spawning procedural meshes

The nodes in red are the ones that sample our layers, like in this example we sample from the grass layer and soil layer. Basically, what we want to achieve is to spawn grass, flowers and more where there is a grass texture at the landscape and spawn other meshes like rocks where there is soil or grass too. We also have a node that samples from a layer called "remove procedural" which basically is a layer like the others but instead of paint an textures at the terrain it's only being used as a mask that tell us in which parts we don't want to spawn anything even if we are on a layer that we should spawn something. That gives us more artistic control over this method. The node with the light blue is the one who is responsible for spawning all the meshes. There we specify which meshes we want this shader to spawn.

So, when we have our static meshes, and that can be whatever static mesh we want, we then convert it to a "grass type" object, where we specify more parameters for the instancing.

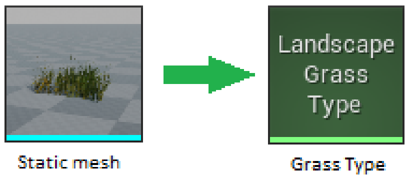
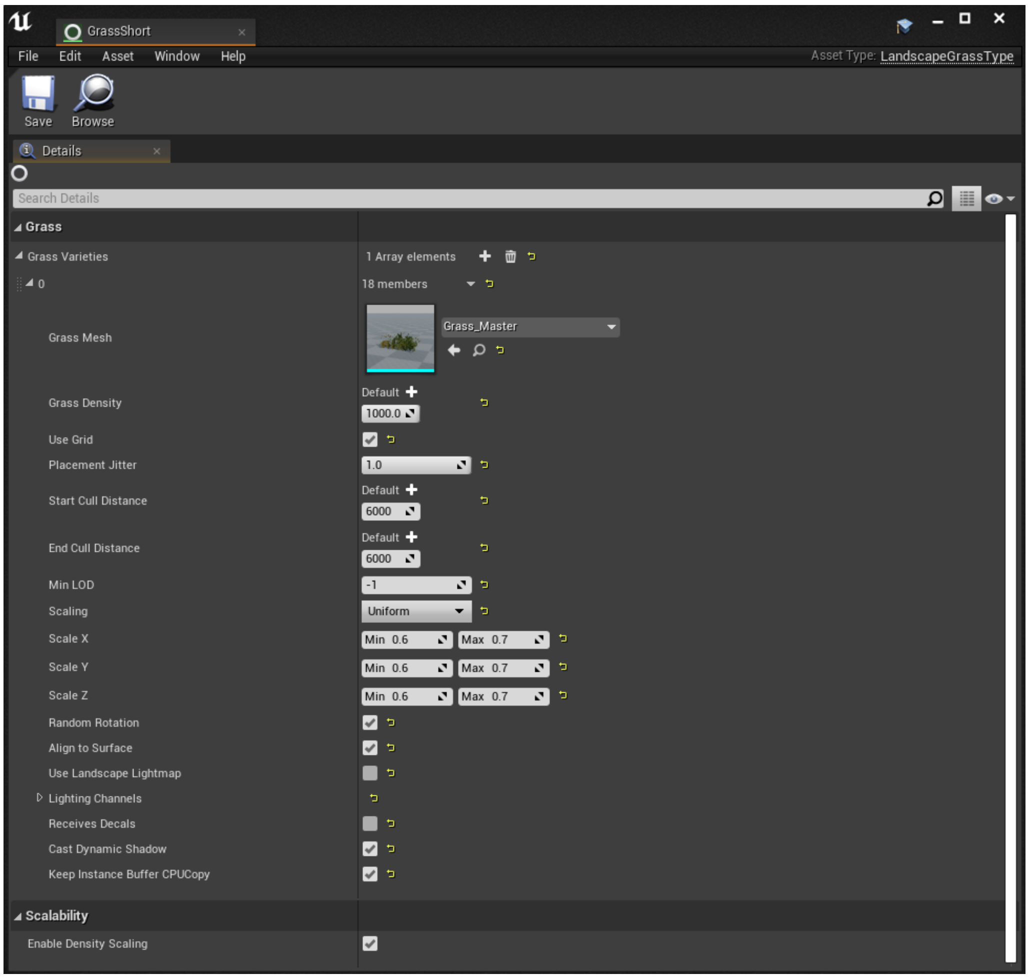


Figure 32: Static mesh to Landscape grass type

Now that we have the Landscape Grass Type object we can specify how dense the population of this mesh will be, how far we want to render which in this example after 6000 units we stop rendering the grass, we can specify the scale of the object if we want to align to surface and at the bottom we have the option to make it follow the density scaling which is basically a parameter that the engine provides us in order to tweak performance quicker, this parameters based on its value will adjust the density of all of the meshes that have this option enabled in order to reduce the count of the meshes quicker.



Asset Type: LandscapeGrassType

Grass

Grass Varieties	1 Array elements
0	18 members
Grass Mesh	Grass_Master
Grass Density	Default 1000.0
Use Grid	✓
Placement Jitter	1.0
Start Cull Distance	Default 6000
End Cull Distance	Default 6000
Min LOD	-1
Scaling	Uniform
Scale X	Min 0.6 Max 0.7
Scale Y	Min 0.6 Max 0.7
Scale Z	Min 0.6 Max 0.7
Random Rotation	✓
Align to Surface	✓
Use Landscape Lightmap	
Lighting Channels	
Receives Decals	
Cast Dynamic Shadow	✓
Keep Instance Buffer CPUCopy	✓

Scalability

Enable Density Scaling	✓

Figure 33: Landscape Grass Type for Grass in the scene

With this method, spawning assets on the GPU side we can populate whatever landscape we have no matter how big that is very fast, without having to place be hand anything, and we have the ability to change all of the parameters really quick like what type of grass we want to spawn how dense, we can spawn flowers and many more. Here is the scene with this shader enabled.

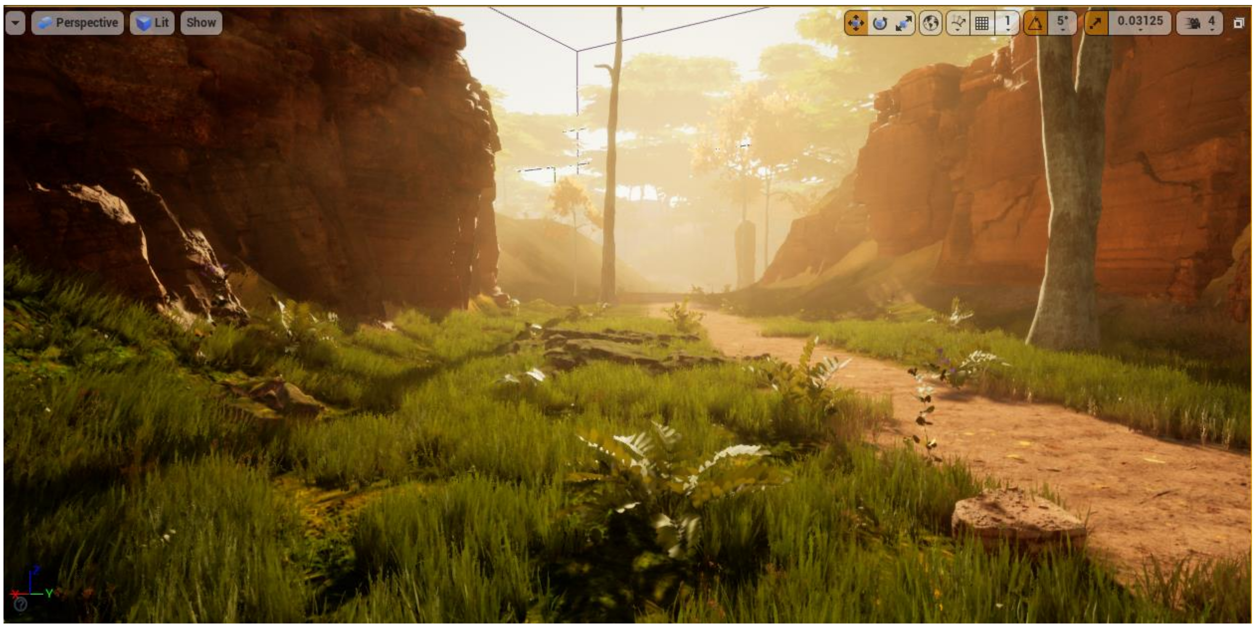


Figure 34: Procedural Foliage Shader overview in Scene

In our scene this shader has the following assets being spawned procedurally

- Grass
- Ferns
- Flowers
- Rocks
- Trees (although we place most of the be hand for more control over the scene)

Stochastic Triplanar Projection on Cliffs

The idea for this shading method on cliffs comes from a paper for a very well-known AAA game FarCry 5 produced by Ubisoft. In this paper among, with many other insights on how the game handles the terrain rendering, we can see how the team managed to render the cliffs. They used the method of stochastic Triplanar projection.

First let us take about the Triplanar projection of the textures on the mesh. The Triplanar projection is a world space projection of the texture onto the mesh based on the normals of the mesh or the camera vector. And what does that mean is that we do not follow the UV layout of the mesh to project the texture, and we use the x, y, and z axis to project the texture.

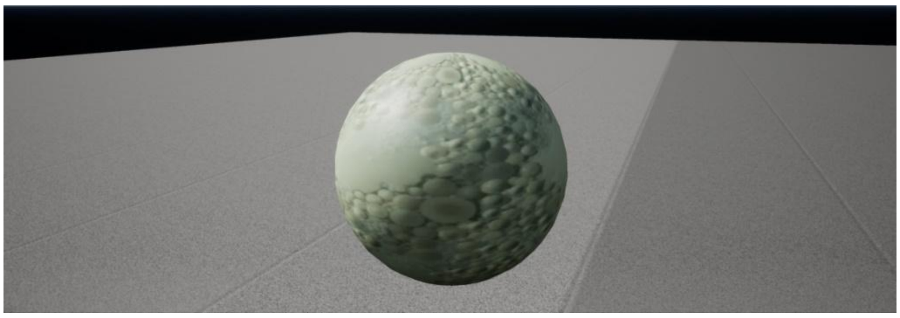


Figure 35: Sphere with normal UV texture mapping

Above we see the projection of the texture onto the UV's of the sphere. If we move the sphere, the texture will remain in the same place as a regular object you would expect to behave like so. But in the second and third picture (Figure 36 & 37) you can see now with the Triplanar projection the texture and the object are not connected in the same way because we project the texture from world space to the mesh based on the 3 planes, one for each axis (x, y, z). And each time we move the object, in this case the sphere, a different part of the texture will be projected onto the mesh.

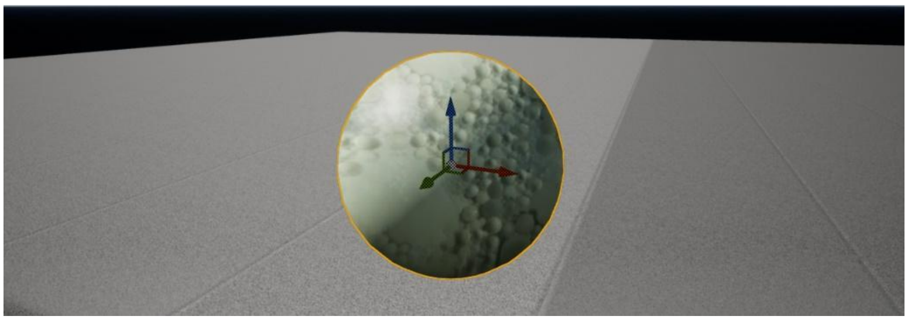


Figure 36: Sphere with basic Triplanar projection 1

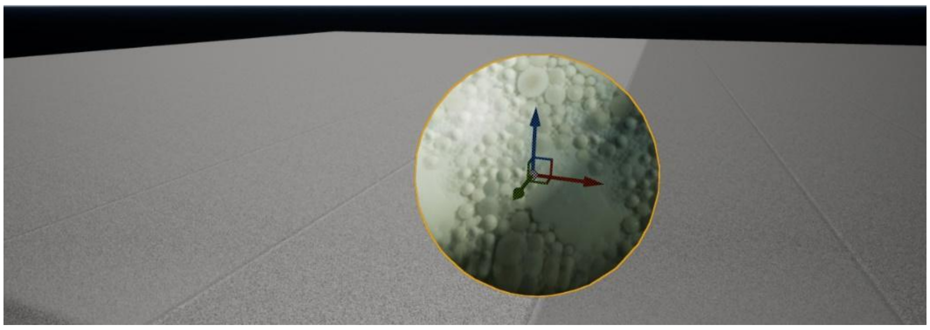


Figure 37: Sphere with basic Triplanar projection
(but moved to the right from what we see at figure 36) to show the effect.

As you can see above when we move the sphere to the right the texture changes to a different projection. We use this Triplanar technique because with the landscape and the cliffs having a big slope the normal projection would cause the texture to be stretched and be blurry and that is why we use the Triplanar technique. But this has its downsides as well and comes with a cost. Normal Triplanar projection will cause the material to sample 3 textures each one for every axis (we mirror the results for the -x and +x and so on, otherwise we would need 6 projections). Now sampling 3 times a texture than 1 is not that big of a difference but we do not have only one texture. We have one texture for base color, one for the normal map and one for the ambient occlusion, roughness and metallic. Plus, as we talked before to avoid the tilling issue of the textures far away, we sample all the textures one more time and use them with a different tilling parameter. So, before Triplanar we had (1 + 1 + 1) x 2 samples, to a total of 6 textures and that is for each component of the terrain. And with normal Triplanar we now have (3 + 3 + 3) x 2 samples which equals to 18 samples. But with the stochastic technique we only sample the texture once. But it also comes with a cost.

To start with, we have the same material setup for the cliff as the one from before we used for the soil or the grass (figure 26), but instead on normal sampling we used the stochastic Triplanar one. Let us see again the sphere we saw earlier but the stochastic technique to see the results.

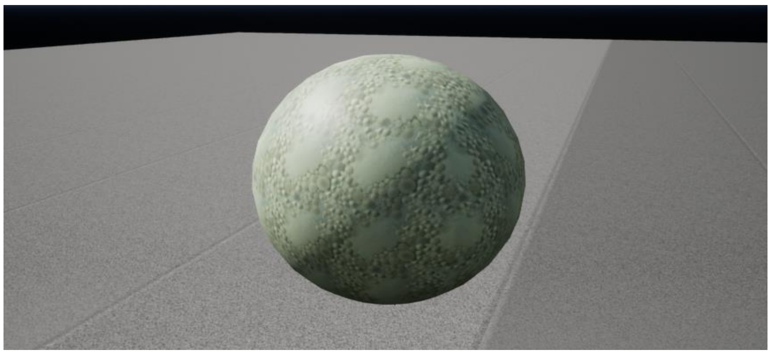


Figure 38: Sphere with Stochastic Triplanar Projection

For far we do not really see the problem that much, but there is one. To blend each projection with each other instead of doing that gradient linearly, we use a noise for the gradient transition. To see the noise, we can visualize each axis projected to the sphere represented with a different color.

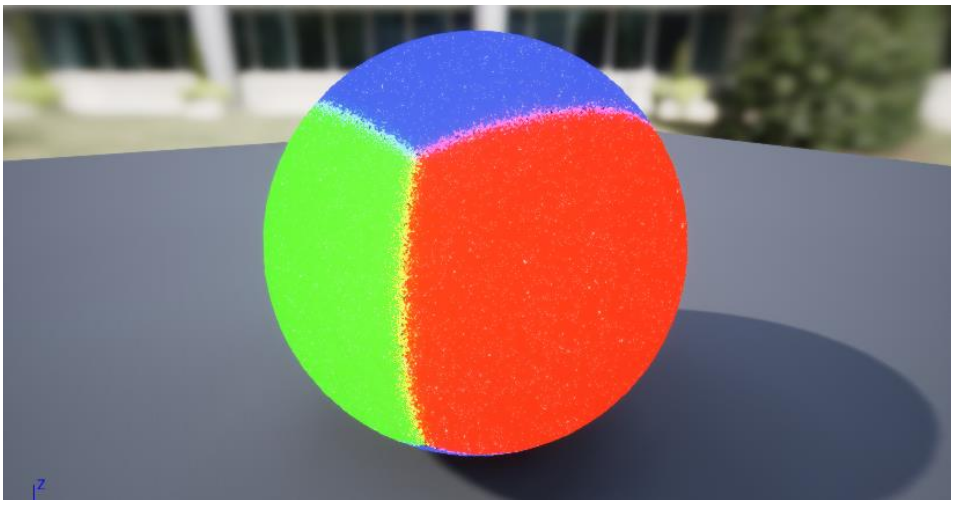


Figure 39: Stochastic Triplanar projection but with each axis represented with a different color

As you can see between each transition of each axis there is noise that simulates the linear gradient but its random noise. The actual noise itself is shown in the image bellow.

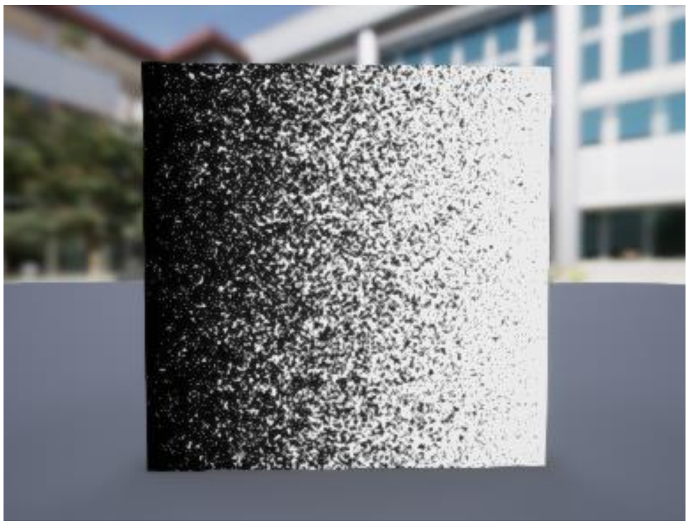


Figure 40: Noise gradient

And the linear gradient that replaces look like this.

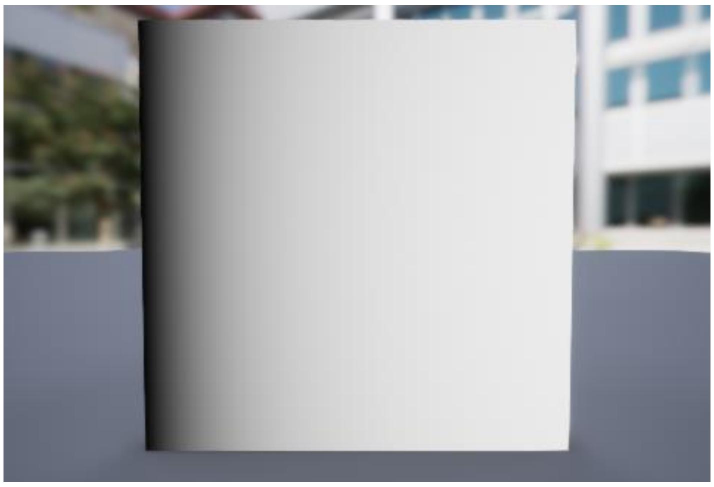


Figure 41: Linear Gradient

So, the noise is not perfect in every point but its good enough on average. The noise comes through in each different axis projection in the normal material but only if we really pay attention to the detail and look close to the mesh. Bellow we see that effect on the cliffs shading.

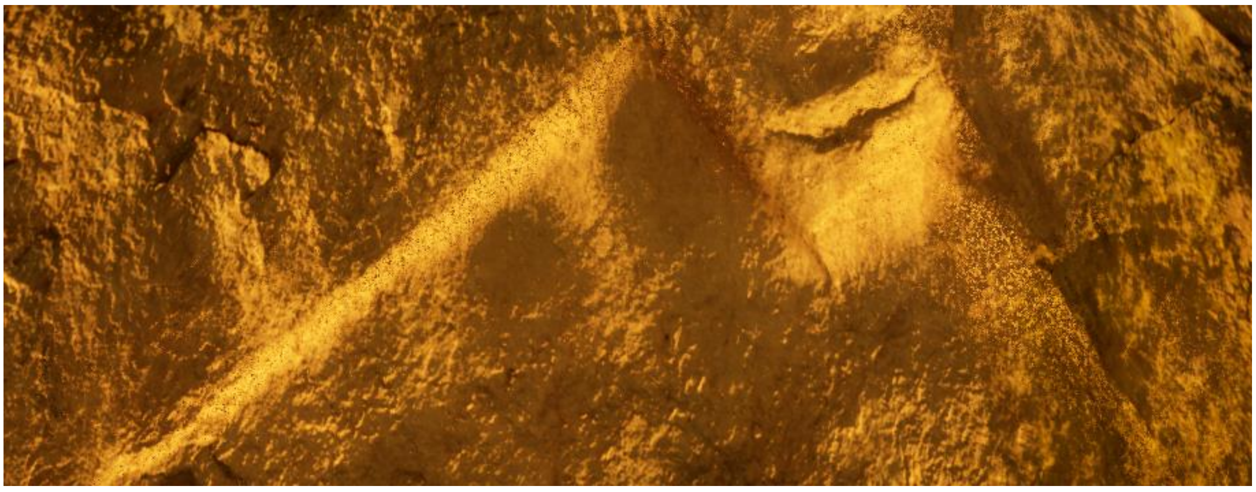


Figure 42: Noise gradient on cliffs

And again this effect only takes places where we need to blend 2 different axis, which most of the time we are not doing that because the cliffs in this scene are very sharp and flat, because we don't use any sort of displacement on the terrain so this effect is minimized. In conclusion, random noise is cheaper than the linear gradient and we only have to sample the textures once and not three times like we did with the normal Triplanar projection, but instead we get this weird artifact with the noise and the wrong anti-aliasing but this is fixed by doing the mip map level calculations manually with the DDX and DDY functions that we can use inside the material.

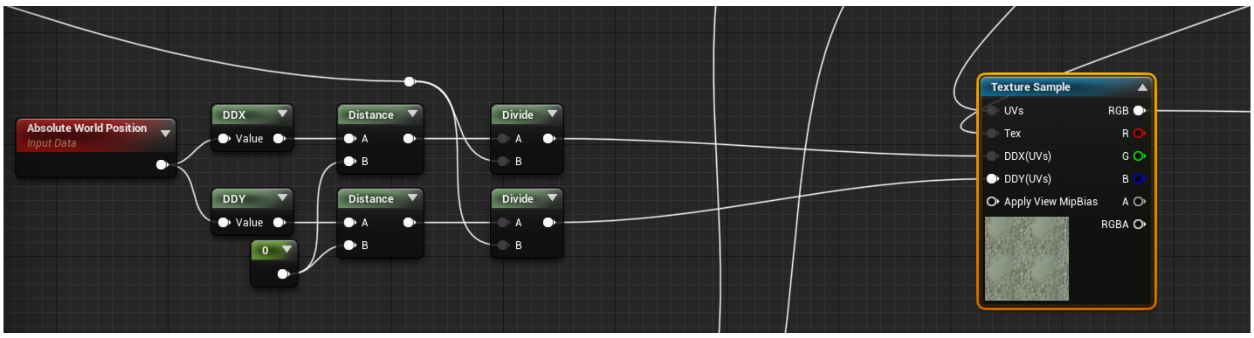


Figure 43: DDX and DDY functions inside the stochastic Triplanar function

Virtual Texturing Landscape

Unreal provide us with two different methods of virtual texturing. The first one is the Streaming Virtual Texturing, which is a different method of stream our textures. Basically, this allows us to handle very big textures with lower memory cost, streaming them from the disk, as to the traditional way of mip map streaming. We can utilize this method by enabling the option in each texture we desire to be streamed like a virtual texture. With this type of streaming the further we are from a mesh the lower the resolution of the texture will be, and as we get closer the texture resolution is growing. We only stream the texture part (tile) that is visible in the camera, which the GPU decides based on the depth buffers. This method does require more performance power because of more texture lookups and some math calculation but it reduces the GPU memory cost. Bellow we have an example in our scene of a mesh that uses streaming virtual texturing for its textures.

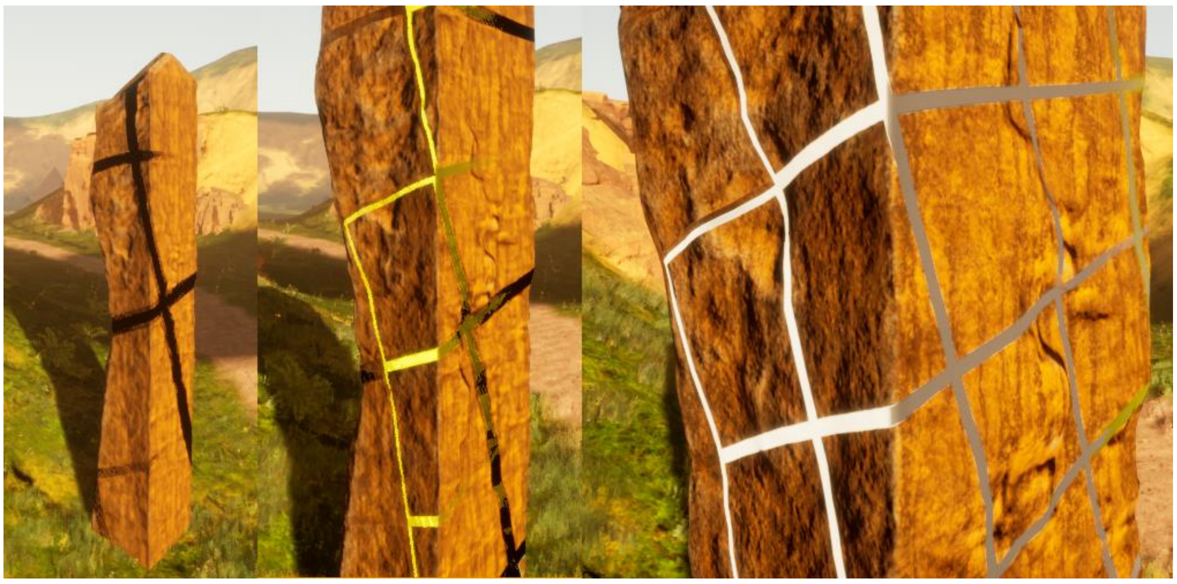


Figure 44: Streaming Virtual Texturing overview

What we can see here is that the further we are from the mesh, the bigger are the tiles that we stream just like the traditional mip map techniques, it’s a lower resolution, and the closer we get to that mesh we stream more and smaller tiles that contain more detail. Not many objects in our scene use that technique, because we only use up to 2K textures but that would be useful if we wanted to contain our texture memory budget in case we wanted to stream 4K or even 8K textures.

What we use a lot in our scene is the second method of virtual texturing, which is called Runtime Virtual Texturing. This method allows us to render the output of a material that an object has to virtual texture. First, we take a Virtual Texturing Volume that we place in our scene and make the bounds big enough to fit the mesh we want to capture, in our case the landscape.

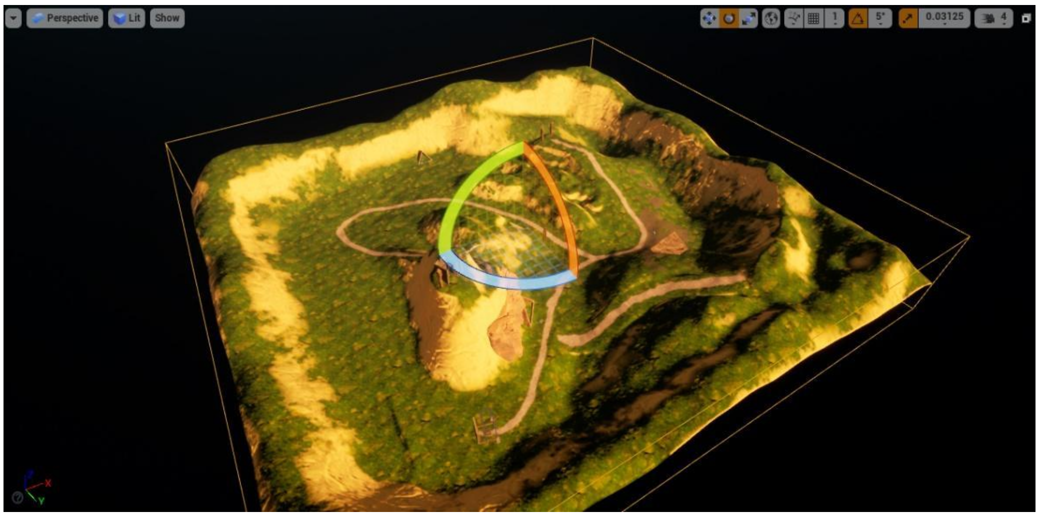


Figure 45: RVT volume bounds

After that we need to create a virtual texture asset from the editor where we are going to store the results of this process. We can specify there how big our RVT will be and how many mip map levels we want to have. The more mip map levels we have the bigger the memory cost will be. Now that we have that we need to adjust our landscape material to write its output to that RVT.

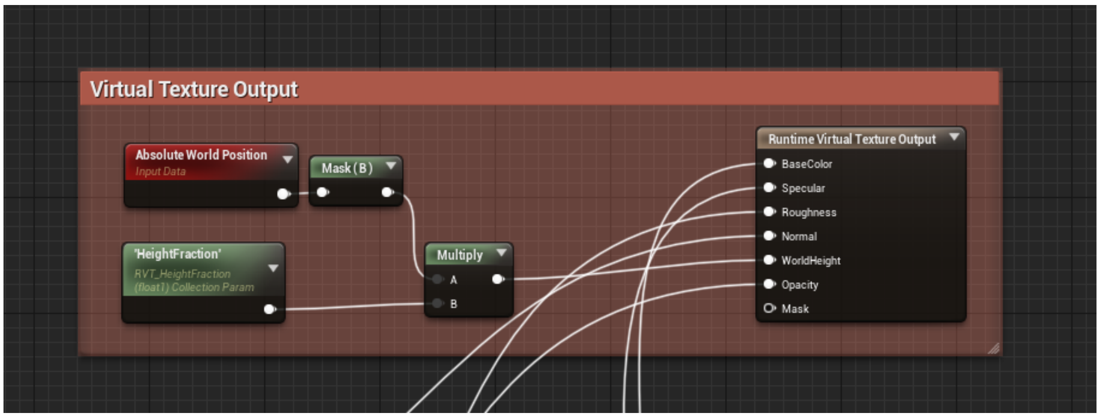


Figure 46: Landscape material writes to RVT Output

As you can see, we write the base color of the terrain, the specular, the roughness, the normals, the opacity and the world height. The result is the following two textures, one containing the world height and the other one for the rest of the attributes.

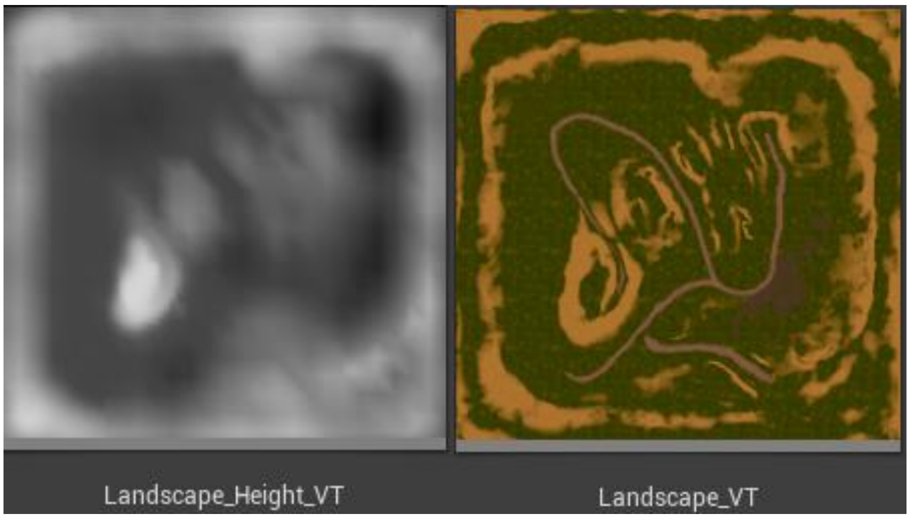


Figure 47: RVT's sampled from the terrain.

Now we can sample from those 2 textures from other materials, and we can also apply this texture to the terrain but we need to make more changes to our material like the distance based shading because the camera distance concept is not available when we apply this to our material but we can use the mip map level to approximate the distance. But we mostly going to use this for other materials. The image bellow shows a mesh from MEGASCANS BRIDGE with the default material how it would look like if we just used the original textures.

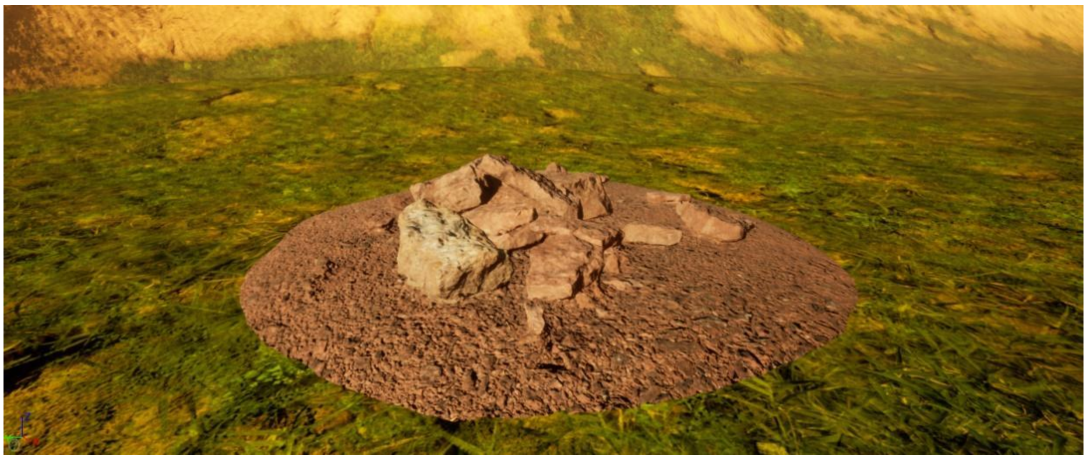


Figure 48: MEGASCANS mesh with default material

Now, if we dive into our material for this mesh (Figure 48) we can modify it like so we can sample from the RVT's and specifically from the point this mesh is placed according to the landscape. With this modification we can combine the original color texture and the color of the landscape and blend between those two, to create a more seamless transition between mesh and landscape. Below is the material.

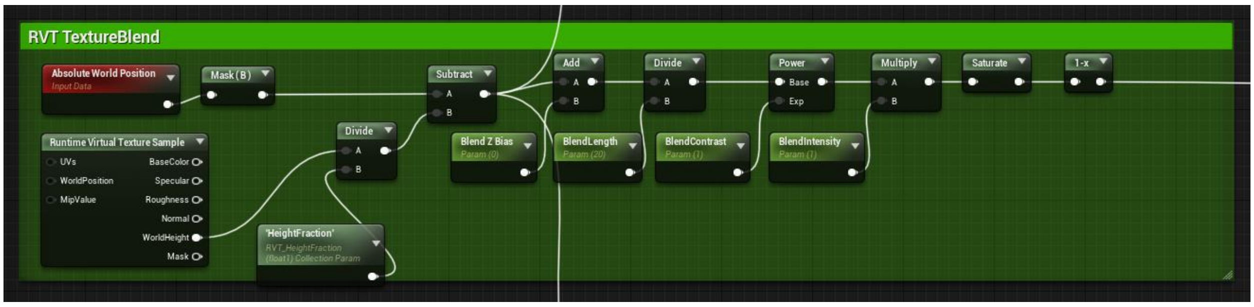


Figure 49: Material that samples from the RVT's

On the left, we can see the RVT sampler. We then use some noise textures and some math to apply the noise as a mask for the blend to create a more random transition and just a flat line, like we did with the landscape at Figure 30. We have more option like how far the transition will go, the noise tilling and more. Bellow you can see the difference with the from the Figure 48, and the position of the mesh is the same.

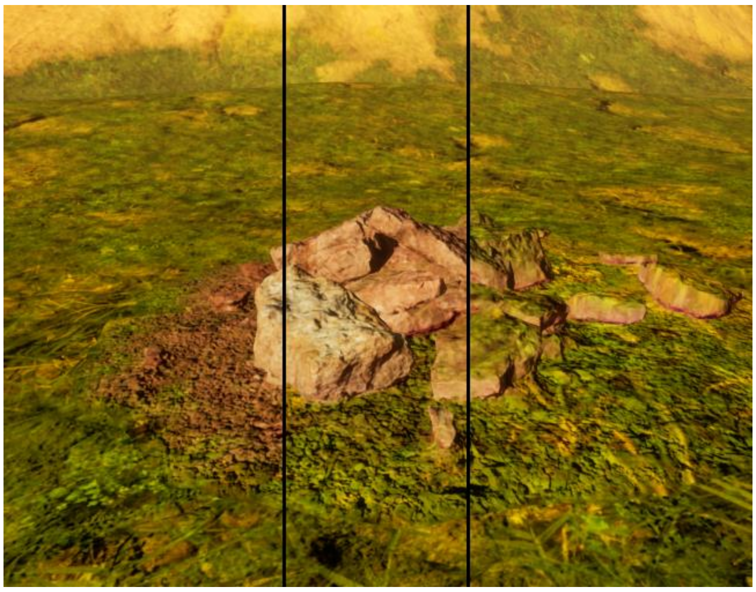


Figure 50: Blend Length adjustments when using the RVT's to blend the mesh.

Foliage & Grass

For the foliage we used in the scene most of them are imported through the MEGASCANS BRIDGE. Things like ferns and flowers are straight from there with small adjustments to the colors, and the subsurface scattering.

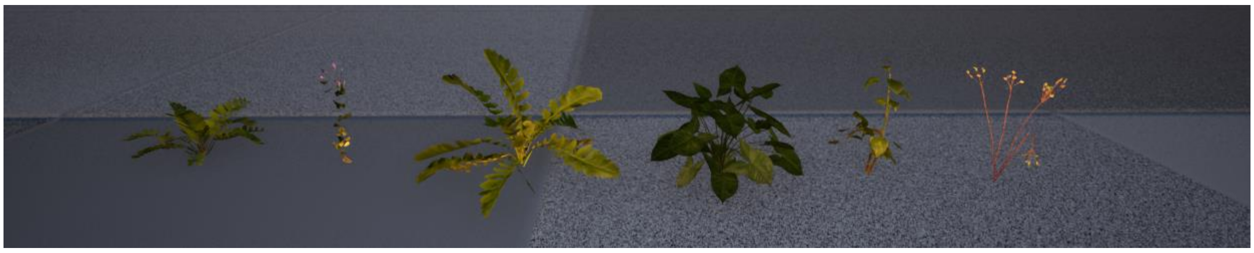


Figure 51: Collection of Ferns and Flowers used in the scene

Each mesh of the above ranges from 1k tris to 2-3k triangles. The first concept of grass was an import from the Megascans Bridge plugin, but again as above each clump of grass is about 10k triangles up close and the concept is to spread the grass all over the landscape meaning a lot of instances of grass spawned.

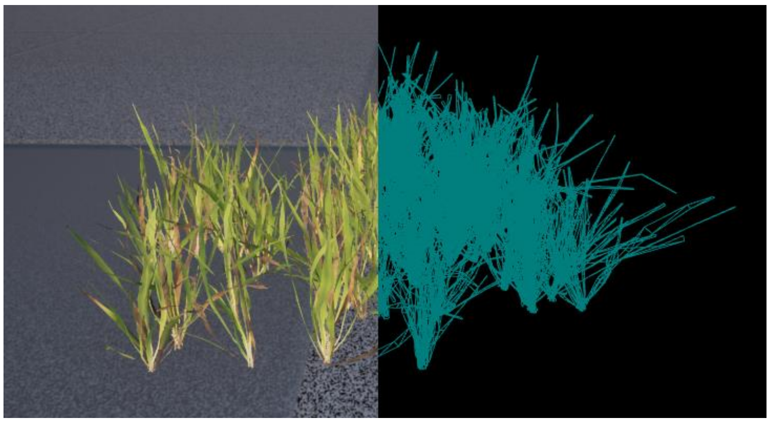


Figure 52: A clump of grass imported directly to the engine from Megascans Bridge, with each blade modeled separately with 10k triangles.

If we try to populate a big area with dense grass the triangle count will skyrocket, and we will end up taking a big performance hit. So, what we can do instead of modeling each blade individually we can create planes that will use opacity masks to cut out the grass shape on the plane.

Having the opacity masks from the Megascans and all of the other textures like color and normals that we get from the plugin in order to use them as billboards when we are far away from grass, we instead are going to use them to shade our planes.

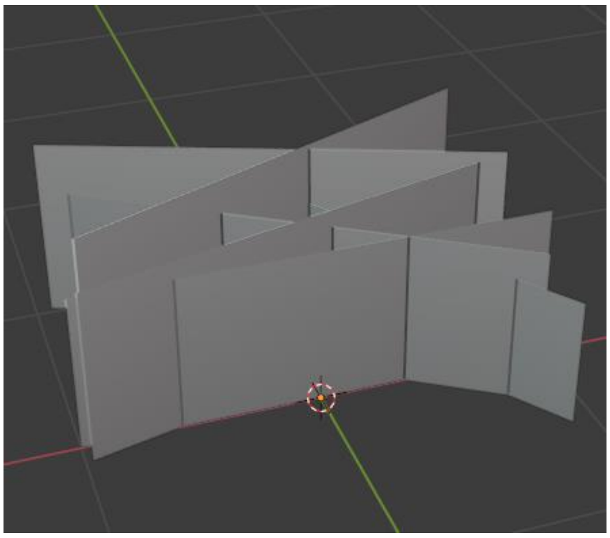


Figure 53: Grass planes modeled in blender 2.8.3

These planes are exported as one mesh to help the instancing get better performance while being only 12 triangles instead of 8000 triangles we had before. Below is the opacity mask we used for the planes.

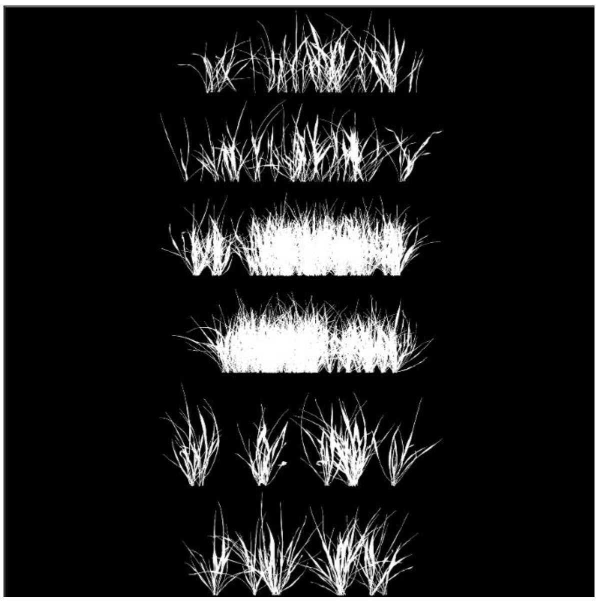


Figure 54: Opacity texture mask for the planes

Now we can map the UV's of each plane to the opacity texture. And with the opacity mask texture when rendering in Engine the black parts of the texture will be rendered fully transparent, and the white parts will render normally.

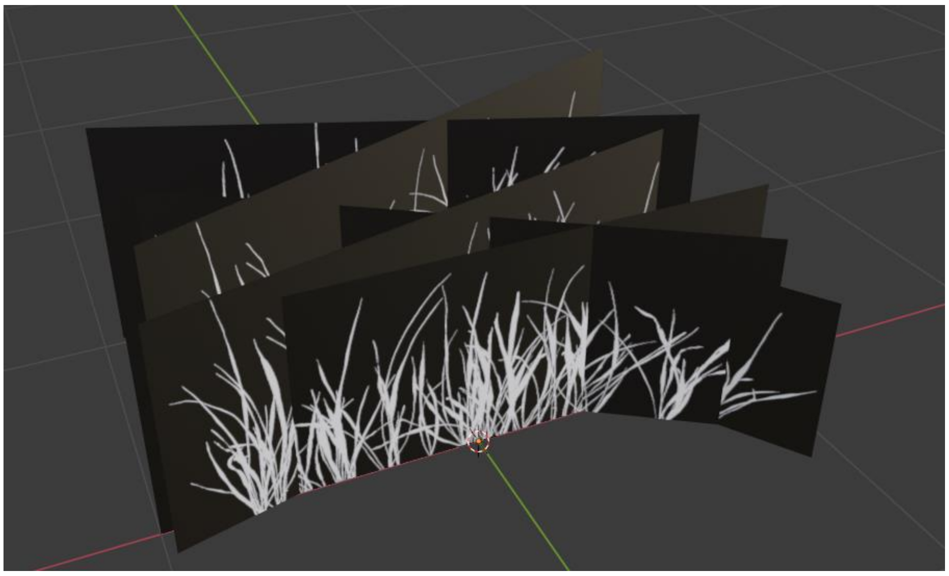


Figure 55: Opacity Applied to Mesh

But with this approach we have a problem when we are looking at the grass from above mostly. We can clearly see the planes of the grass and that does not look so good. But there is a solution to that problem. We can follow a similar approach that used in the game Horizon Zero Dawn that the developers talked about in their GDC talk. What they did is to tilt the planes according to the player camera position.

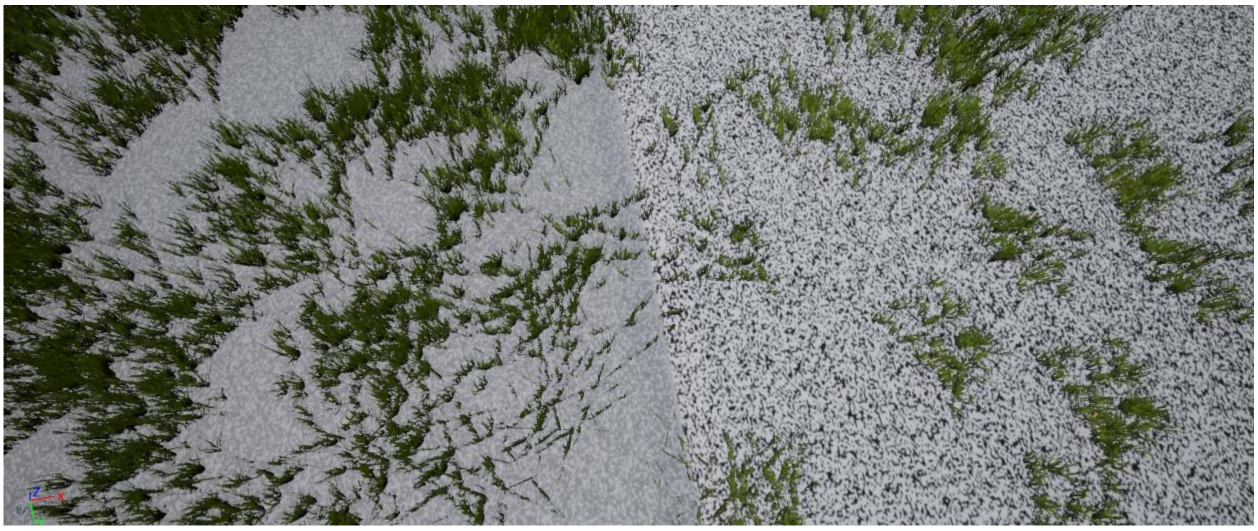


Figure 56: Grass planes when looking from above

We need to do two things to make this work in unreal also. First, we need to paint the vertices of the planes with weights, and with this way we can tell which part of the mesh we are going to tilt and which not. We can do that in blender by going to the vertex paint menu.

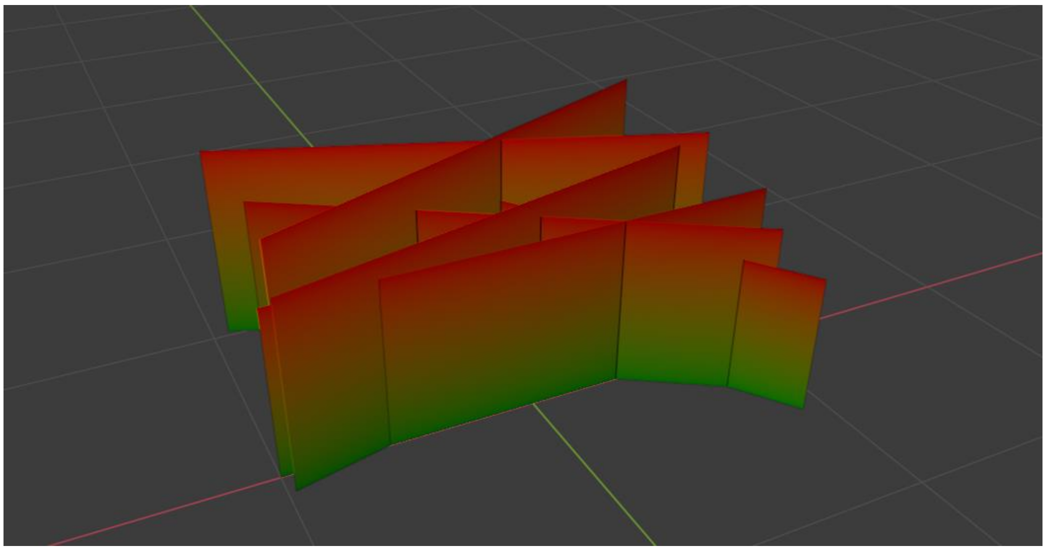


Figure 57: Grass Vertex Paint

Now that we have that we need to write the custom shader code to tilt the grass. We can do that by getting the distance of the camera and the distance of the mesh, and based on the distance of both the closer we are to the mesh, the more we are going to tilt the planes based on the vertex colors.

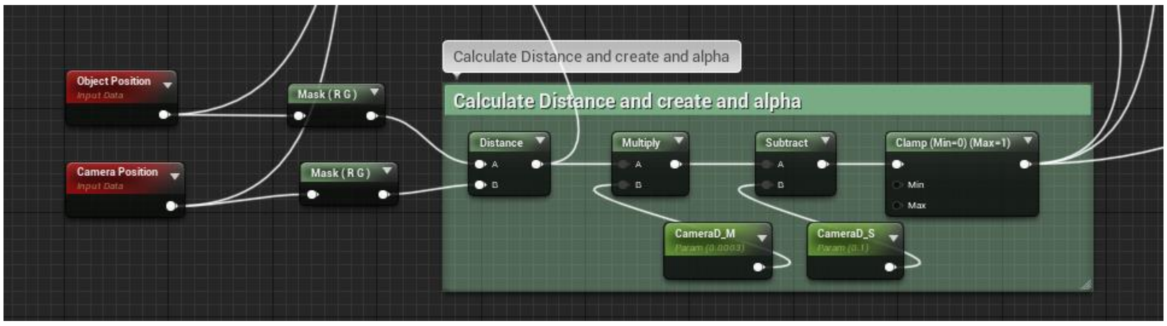


Figure 58: With these nodes we calculate the distance and we are creating and alpha gradient we are going to use later

Now we can create a custom node inside the grass material to write our custom HLSL code to tilt the grass. Below you can see how we take in consideration the vertex color and we want to displace only the red painted parts of the mesh.

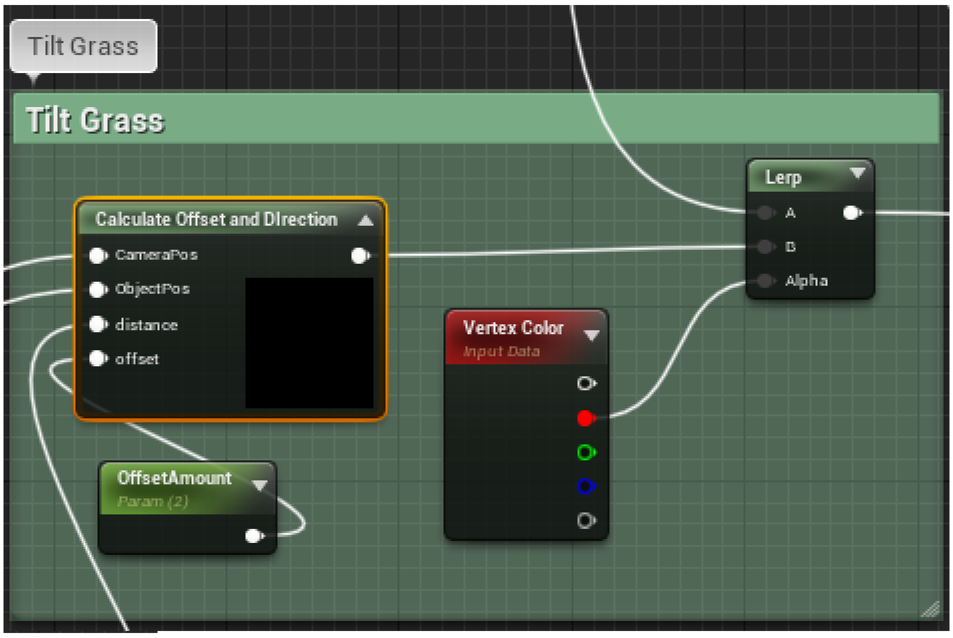
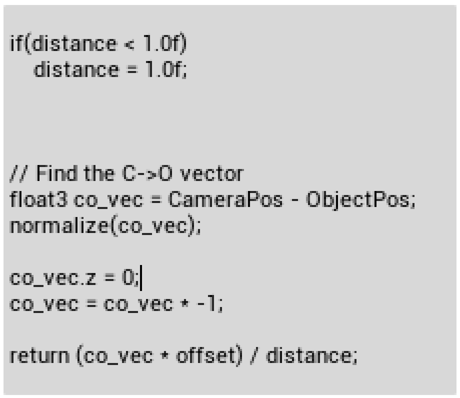


Figure 59: Custom Node inside material

The custom code looks like this. We first do a distance condition. Then we want to find the vector that goes from the camera to the object (grass mesh), and then we normalize that vector because we want only the direction of that vector. After we make the Z coordinate equal to 0 for the camera vector position, because we do not want that to affect the tilt amount, we then invert the direction of that vector to tilt the grass to the way the camera is facing and we multiply that value by a variable we can later adjust.



```
if(distance < 1.0f)
   distance = 1.0f;



// Find the C->O vector
float3 co_vec = CameraPos - ObjectPos;
normalize(co_vec);

co_vec.z = 0;
co_vec = co_vec * -1;

return (co_vec * offset) / distance;
```

Figure 60: Custom HLSL code to tilt the mesh

Now if we look at the grass from above at an extreme angle again, we will not see the planes like we did before. Of course, this is a bit too much tilt but also the angle is very extreme and normally we do not notice this effect, but it helps a lot to sell the realism needed to hide the planes at other angles.

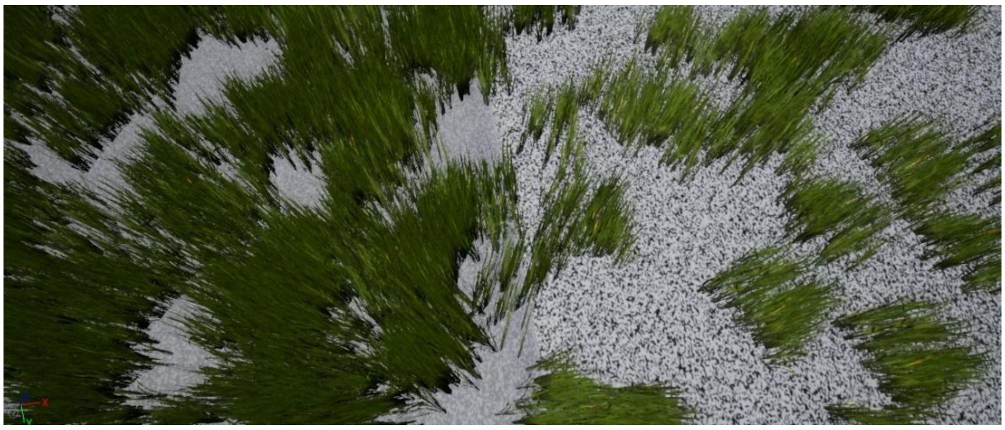


Figure 61: Tilted grass

To optimize further the grass because of its density we use a culling effect to stop rendered the grass at far distances. This also causes a noticeable pop effect every time we move forward because the GPU will try to batch big patches of grass together in order to optimize the draw call count, and that means when we move forward and we need to render the new grass it draws the whole patch at once and causing a pop of the grass to appear on the screen making the culling effect noticeable. But if we use the Distance mask we talked about later (Figure 58) we can make the grass that lives at the edge of the culling distance be under the ground by displacing the vertices of the mesh. And based on the distance the grass will grow from the ground progressively. With this way the pop of the grass happens under the ground and when we go close enough the newly grass grows from the ground smoothly.

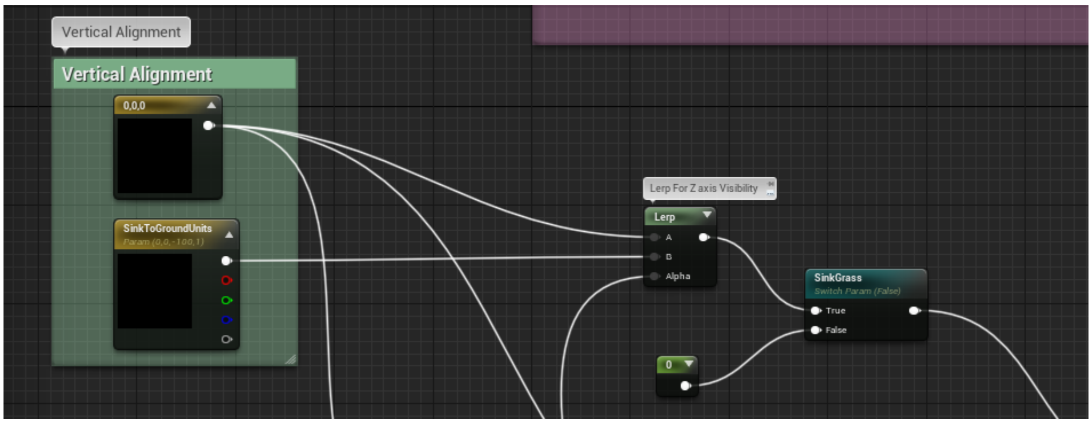


Figure 62: Sink Grass to the ground material

At last we use the traditional wind node inside the grass material to simulate the wind, which is just a noise displacement of the mesh based on the vertex colors. And we also have the option to shade the grass based on the landscape position, by sampling from the RTV's of the landscape.

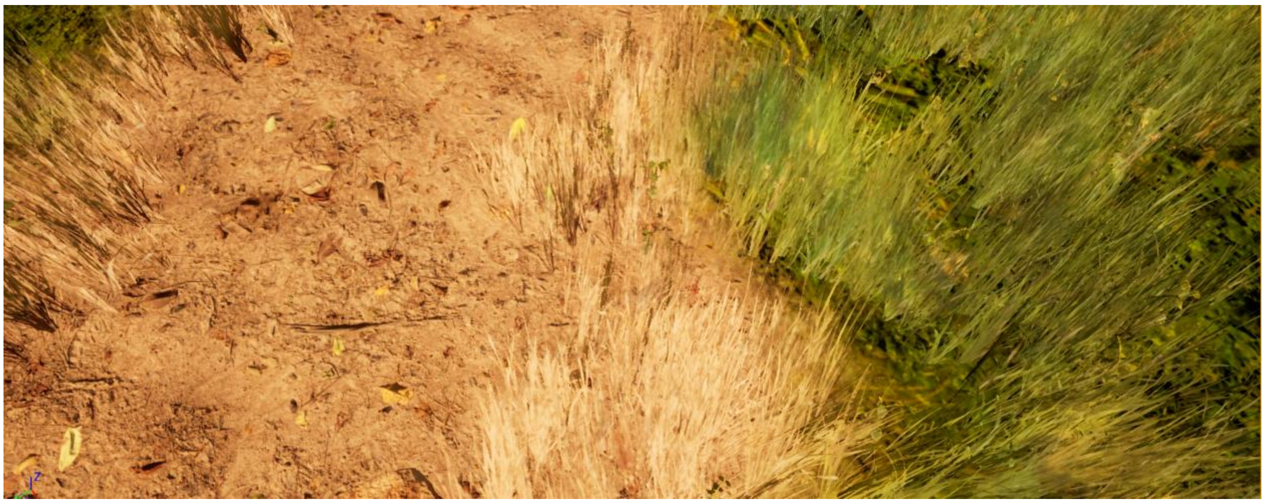


Figure 63: Grass color by sampling the RVT of the landscape

The above example shows the grass sampling from the RVT of the landscape, and only using the color data provided by that texture and none of the original grass texture just to illustrate the effect better. But we can blend between those two textures in a more subtle way to make the grass more diverse and make it fit in the different parts of our landscape.

Lighting

To start with, the goal was to make the lighting in the scene dynamic, which means the light source can be both movable and change its intensity and color, to simulate a day/night cycle. The techniques used in this project are inspired again from "Horizon Zero Dawn", based on an article from the lead lighting artist of this game. What they did is to keep the Directional Light dynamic, allowing it to retain all of the properties discussed before, using a volumetric approach with spherical harmonics, and also including a Sky light which lit the scene based on the sky color from all angles. The Indirect lighting is baked into irradiance volumes, like 3D textures to light all the static and dynamic objects.

The same approach is used in our project. We have a Direction Light that mimics the sun and it is also dynamic. Using only that without baking anything we only get bright spots and very dark spots as this Light does not simulate the light bounces. Next to that we used a Sky Light, which is stationary light. Now what stationary means is that we cannot move the light but what we can do is change its intensity and color. After that we also baked the indirect lighting only, using volumetric lightmaps.

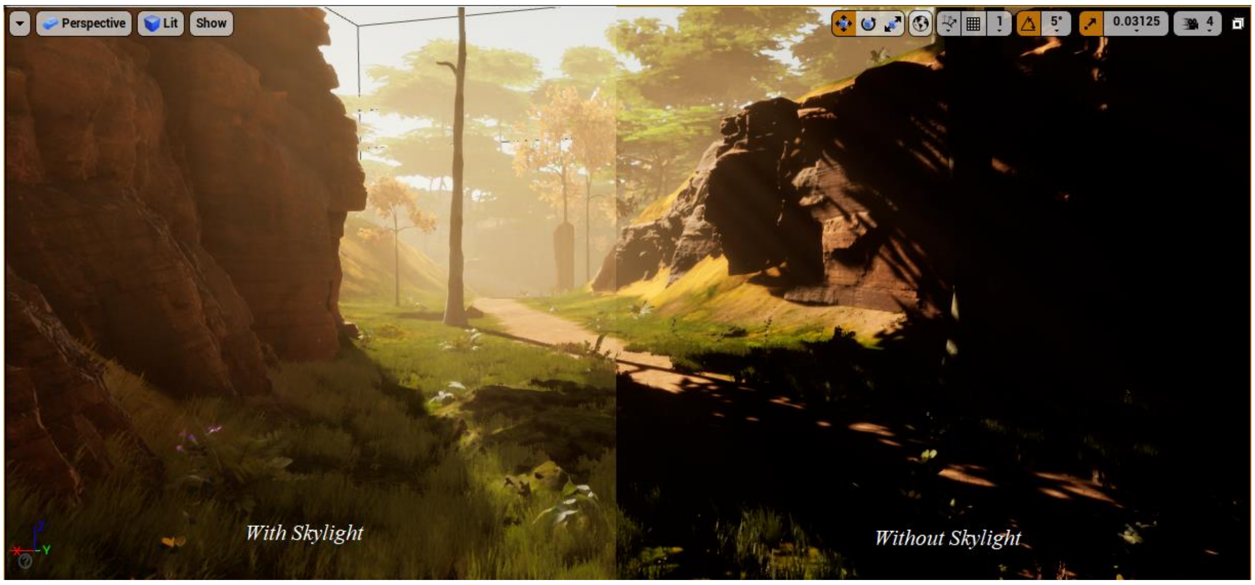


Figure 64: Skylight Difference (With Skylight left / Without Skylight right)

With this solution what the engine does is to spread samples across the whole scene, especially nearby objects, that will capture the indirect lighting that we produce when we build the lighting of the scene, using third order spherical harmonics to capture the lighting from all of the directions. Now a dynamic object moving in the scene, like the character, will be lit according to its surroundings, the same if that object was static and had its light built offline.

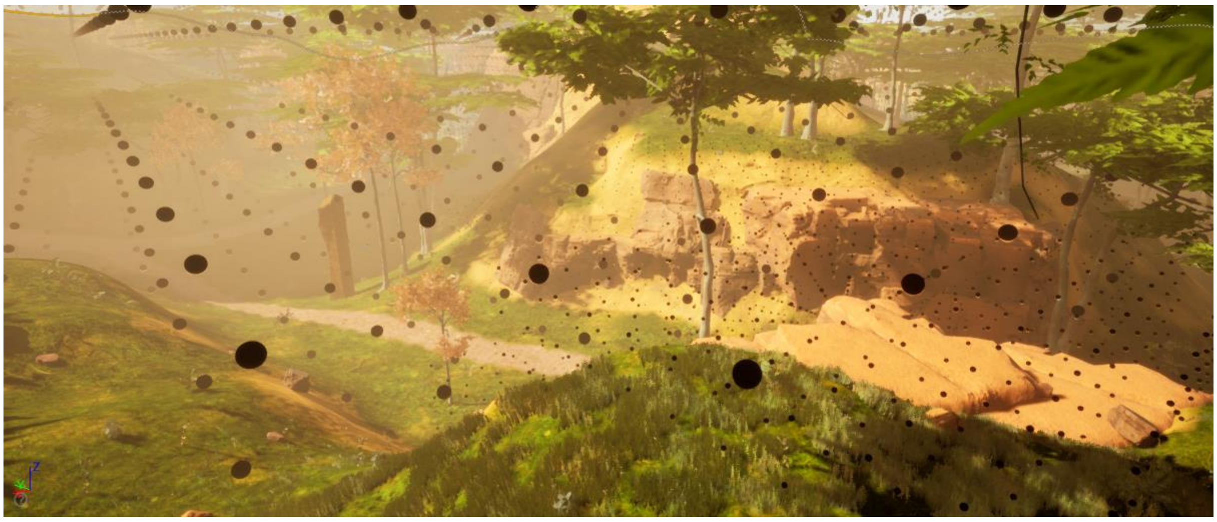


Figure 65: Volumetric Light Samples (They are denser close to static meshes like at bottom right)

Now that we have a Directional Light, a Sky Light which is used as an ambient light, and our indirect light, we also need a sky, and for that we used the Unreal’s Sky Atmosphere component. We can assign the Direction Light as the light source for this component and it will solve the sky color for us automatically simulating the sunrises and all of the colors of the sky based on the sun position, while we can still tweak that component to adjust our sky color to our needs. The Sky Atmosphere is a physical correct implementation of the real sky on earth which gives great results out of the box. At the image below we can see the different colors of the sky in the early morning first, and then when the sun rises (still morning) and at noon.

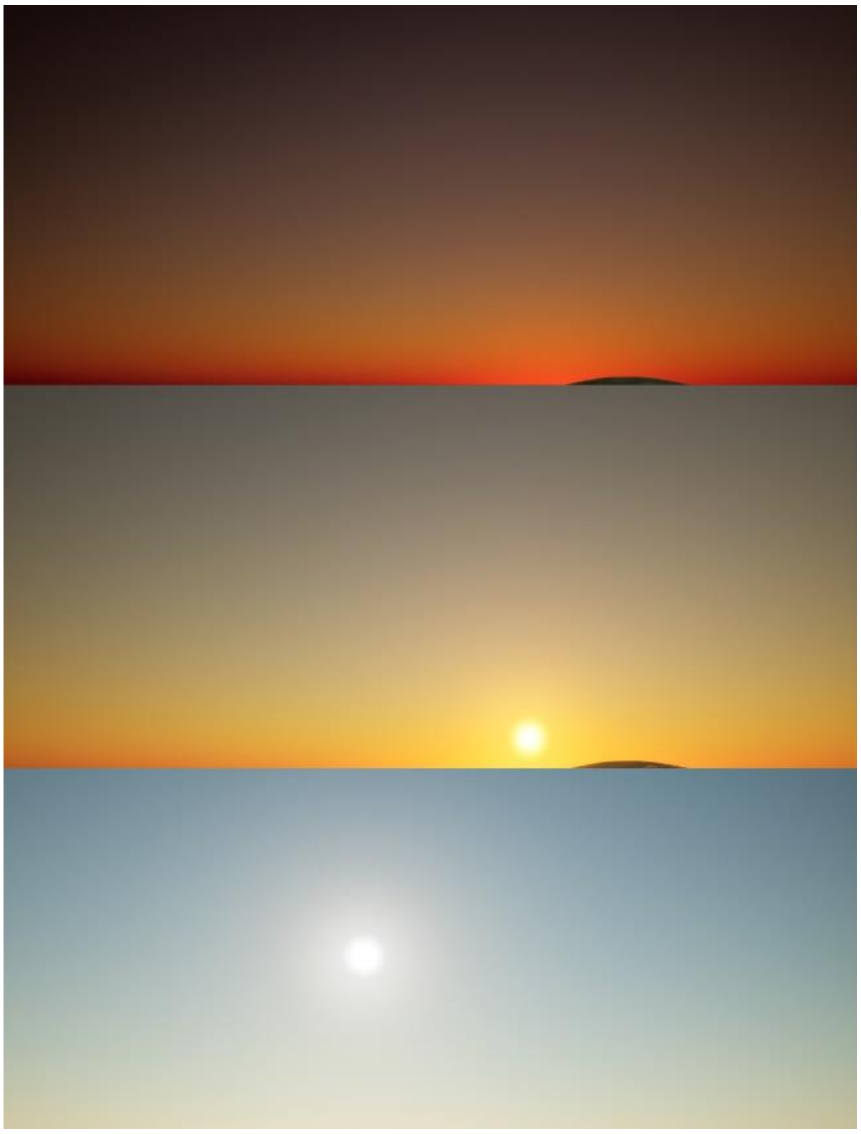


Figure 66: Sky atmosphere in different times

For the fog and the volumetric effects, like volumetric fog and the "god rays" from the light we used the Exponential Height Fog component. The volumetric fog does cost a little more that the regular for, but it does help a lot the scene come through. And with the volumetric fog we can recreate the rays of light that scatter through the fog creating the "god rays" effect. The fog is also used to hide the distance meshes and their flaws and blend together the whole scene.

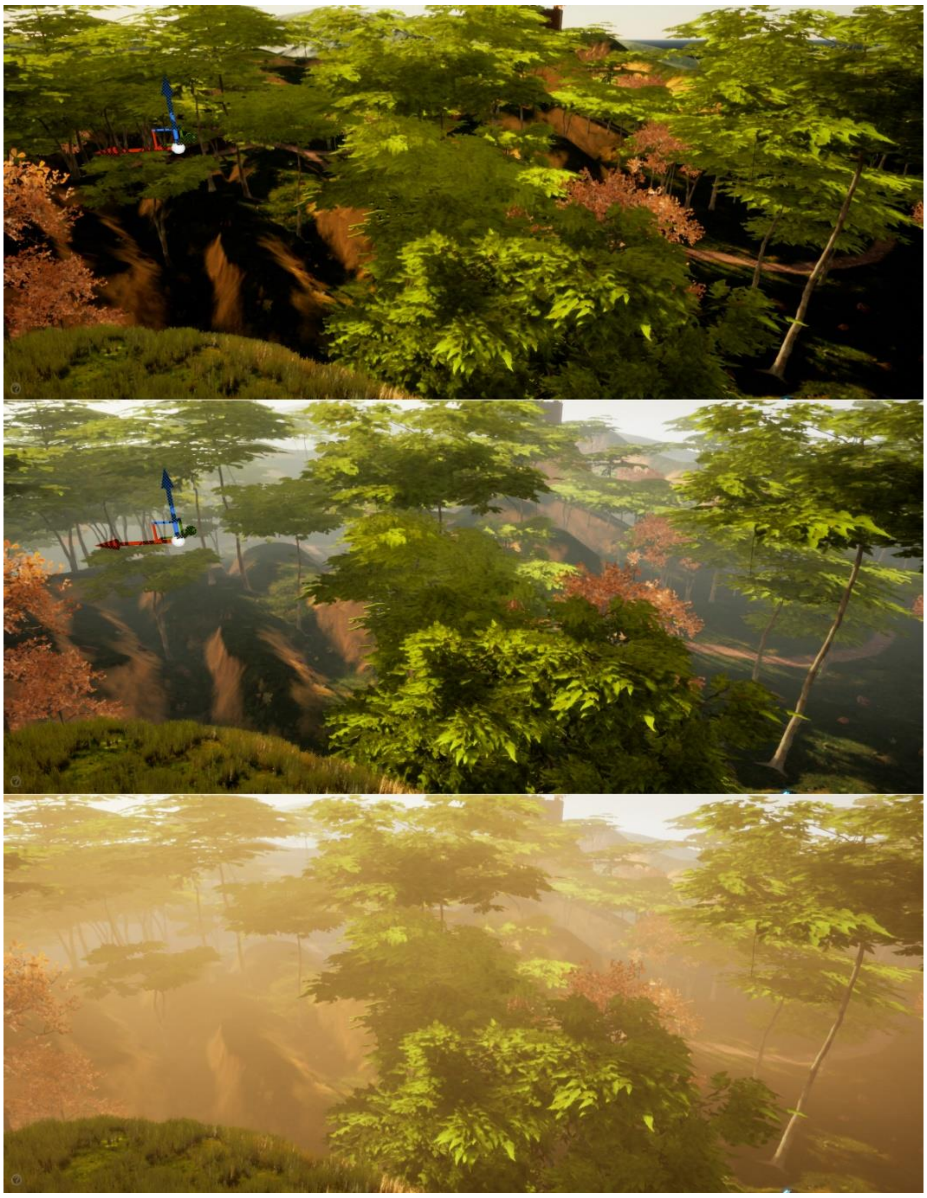


Figure 67: No fog - Exponential Height Fog - Volumetric Fog with "God's rays"

Lastly, for the shadows we used 2 different techniques. Shadows take a very big portion of the render time (from 2ms – 6ms) and are very heavy to render. For the close shadows we use the Cascade Shadow Maps system, and for the far shadows we use the Distance Fields of the mesh to render their shadows. With the CSM, we basically LOD the shadow quality into steps. In our scene we have 3 steps. The dynamic shadows that are closest to the player will be rendered with the best quality and from there and on the quality will decrease. At the picture below we see only the dynamic shadows for the scene. The render distance for these shadows is 4000 units, and we can clearly see the line where the shadows stop rendering.

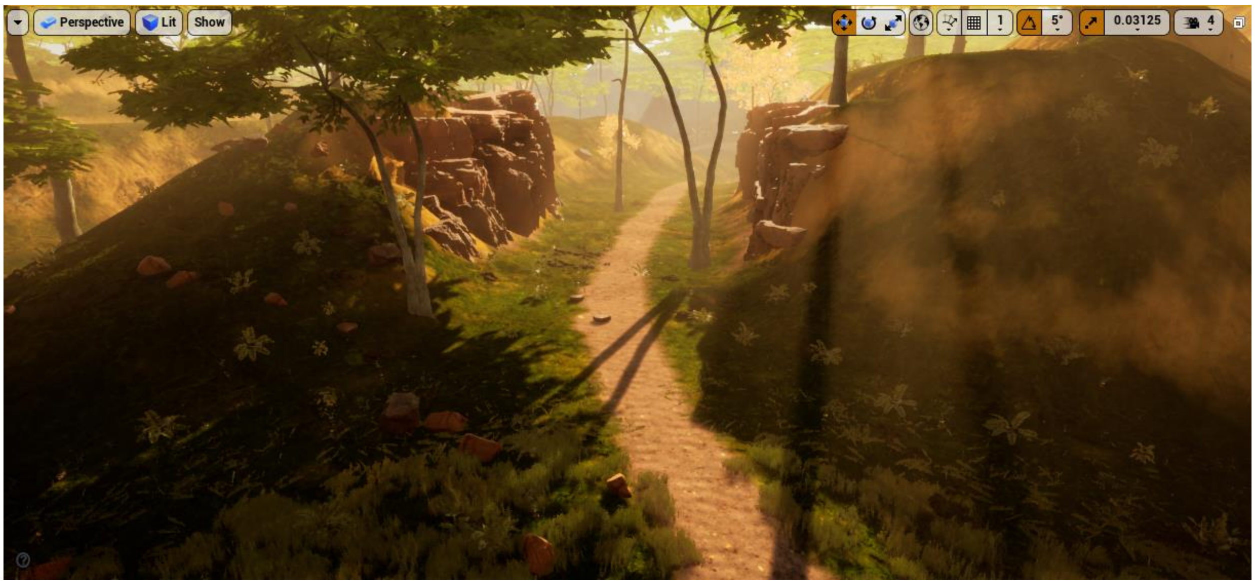


Figure 68: CSM with 4000 units render distance

We can see that the trees on the background do not cast any shadows neither to themselves nor the terrain. Before the rocks we can see where the shadows stop rendering. We do that because it is very expensive to render dynamic shadows that far away. But there is a solution to the problem, and it is called Distance Fields. We need to enable that feature inside the engine settings. Basically, what the engine does is to store into a three-dimension texture (a volume texture) the distance to the nearest surface of every mesh. It is an approximation of the mesh, stored in volume textures, and we can use this data for ambient occlusion, shadowing, and for GPU particle collision.

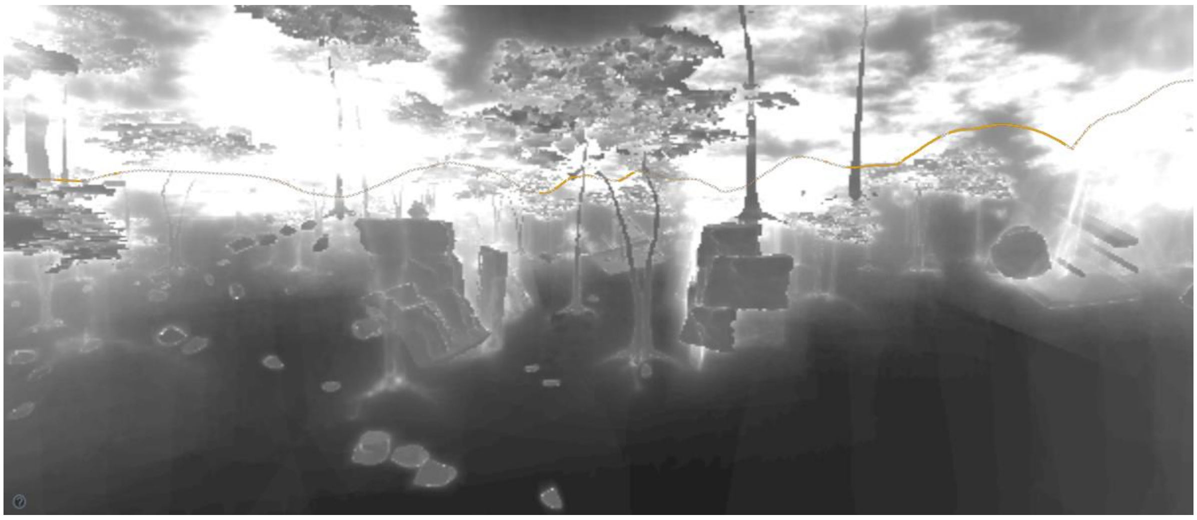


Figure 69: Mesh Distance Fields

Back to shadows, after the 4000units we use the distance fields to produce the static shadows for all the way to the end of the terrain. These shadows are very cheap and fast to calculate over the dynamic shadows, and they are always in the background so we cant really notice the change between the dynamic ones and the static, because we blend between those two.

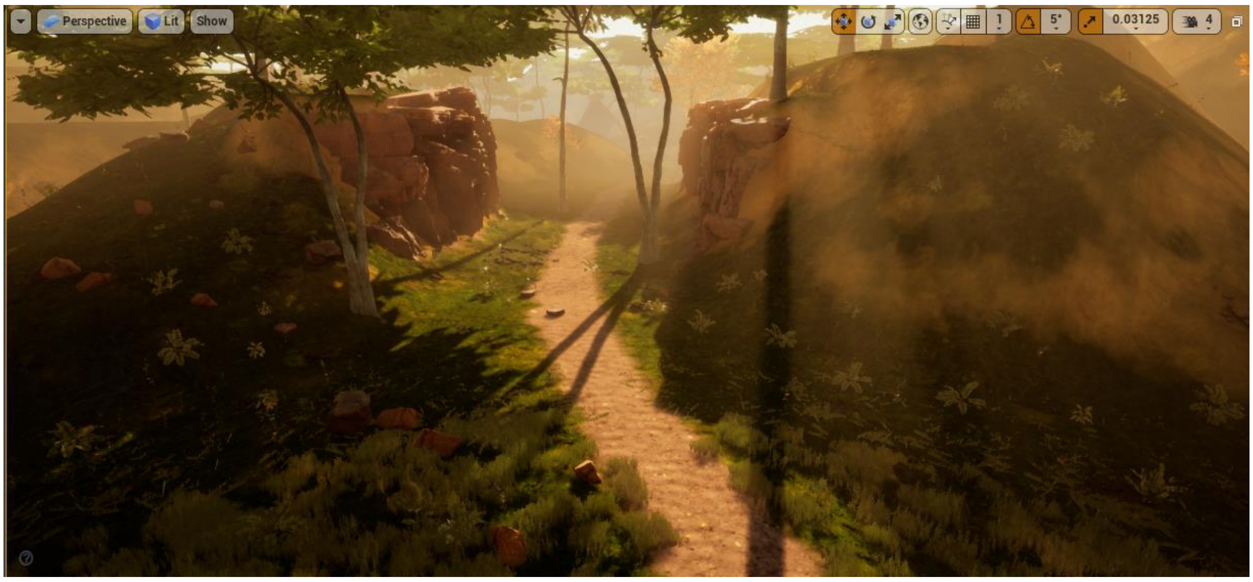


Figure 70: Distance field shadows on

Asset Creation Pipeline

Now we are going to talk a little bit about the asset creation pipeline used on this project. The main programs used are, Quixel Bridge for the surfaces (textures), Quixel Mixer for texturing the meshes, Blender for creating the meshes and creating the UV's of the meshes, Zbrush for sculpting the models, and xNormal for baking the normals of the meshes.

For sculpting some of the rocks we start with the Zbrush, create the high poly model with all of the details, then we remesh the object to have a low poly model too, that we will use in our game. With xNormal we will bake the high poly details to the low one. With blender we create the UV's of the low poly rock. After having the normal map, we move to the Mixer, where we create our textures for the rock. With mixer we can create various textures with techniques like masking with noise, taking the curvature maps and many more. Now that we have the textures of the model, we then import the low poly model to the engine along with its textures and create the material that we will use inside the engine. This technique is used with every model created, the arms of the player, the gun of the player, rocks, trees. Below is the process for creating the enemy drone.

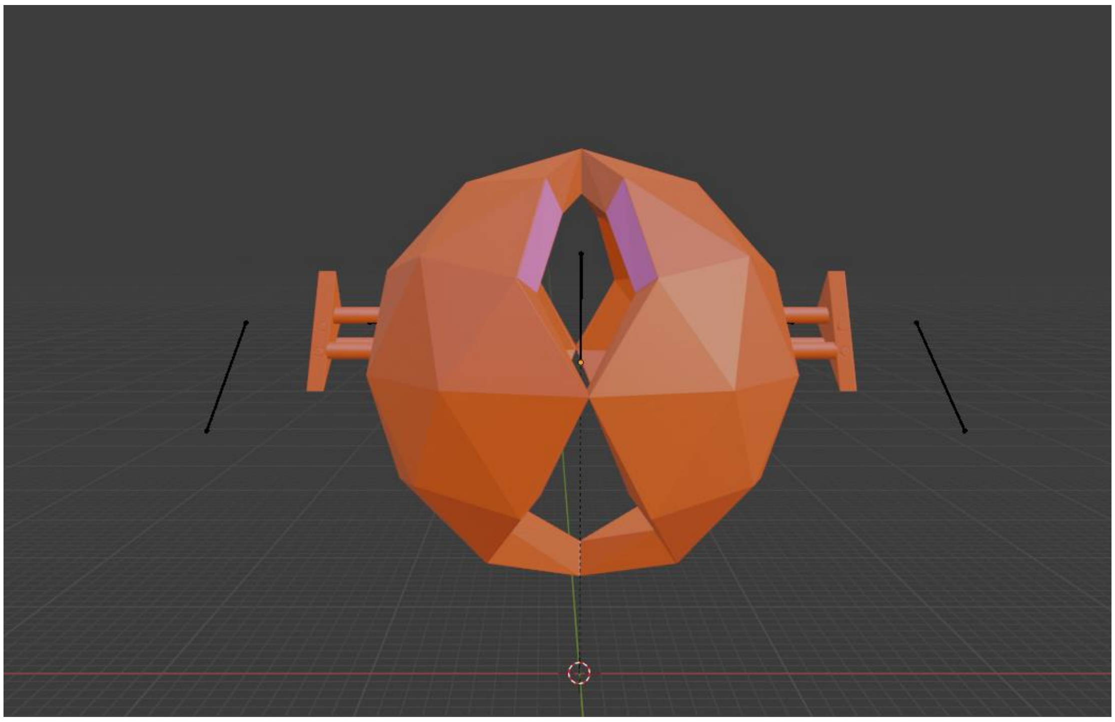


Figure 71: Blender model creation & rigging

We then take the model to Zbrush to sculpt some fine detail, like the rust and the metal displacement to bake that detail with xNormal to a normal map.

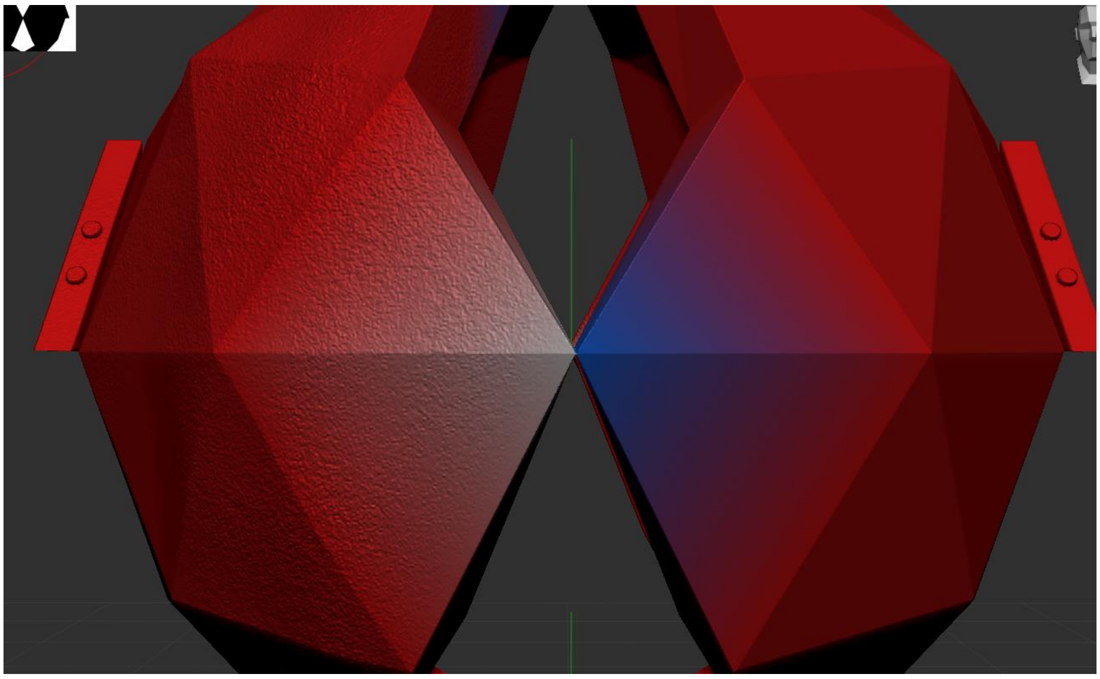


Figure 72: Model in Zbrush (left sculpted detail - right original)

And then as a final step, after we have the UV's ready and the low poly model, we take the final mesh to Quixel Mixer to texture it.

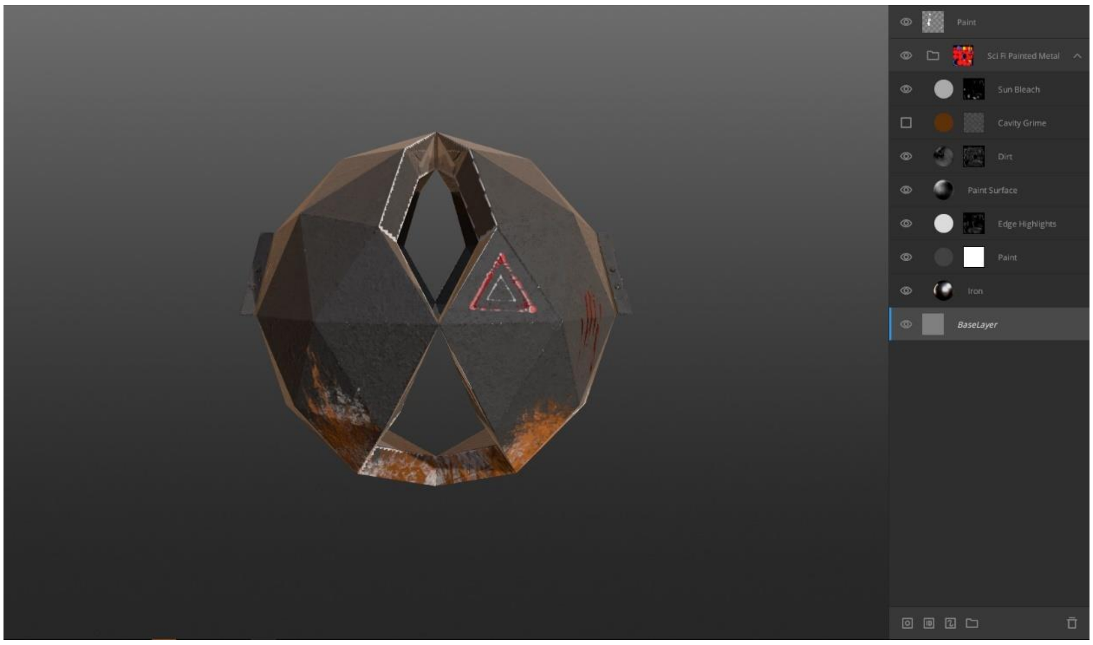


Figure 73: Quixel Mixer texturing

Most of the foliage as we saw before came from Quixel Megascans Bridge, with some changes to the original and mostly to the grass, except from the trees. The trees where made with SpeedTree and Blender. First, we need to model several branches of the tree in SpeedTree to bake their textures to planes. And then use those planes to populate all the branches on the tree.

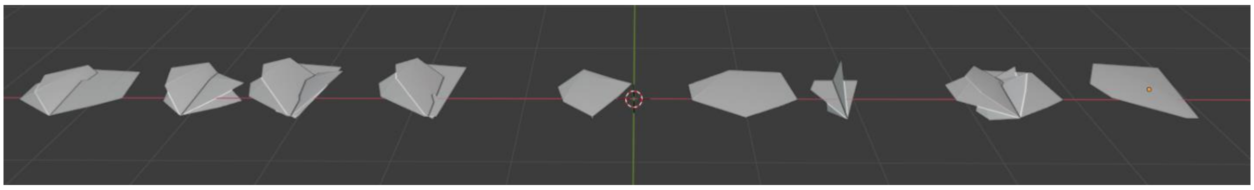


Figure 74: Branches Meshes for the trees

We project the branch textures we create at SpeedTree into those mesh that will act as the branches. This way we save a lot of triangles to render, and we do not have to model each leaf separately.

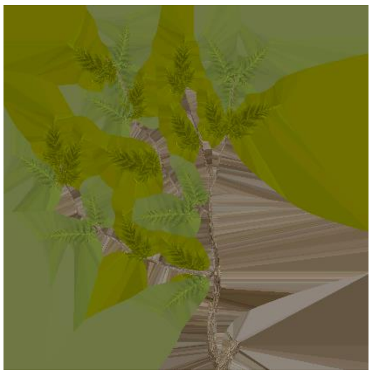


Figure 75: Albedo texture of a branch

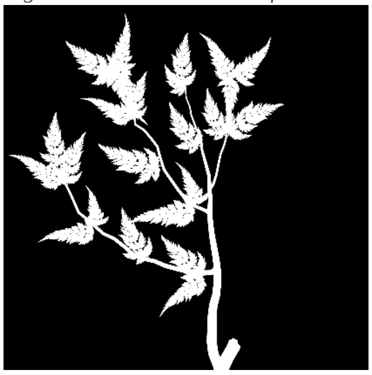


Figure 76: Opacity mask of a branch

And now we can start generating these meshes onto the tree bark to finalize it. As you can see all the leaves and branches are just planes that use a texture opacity to cut out the shape of the branch.

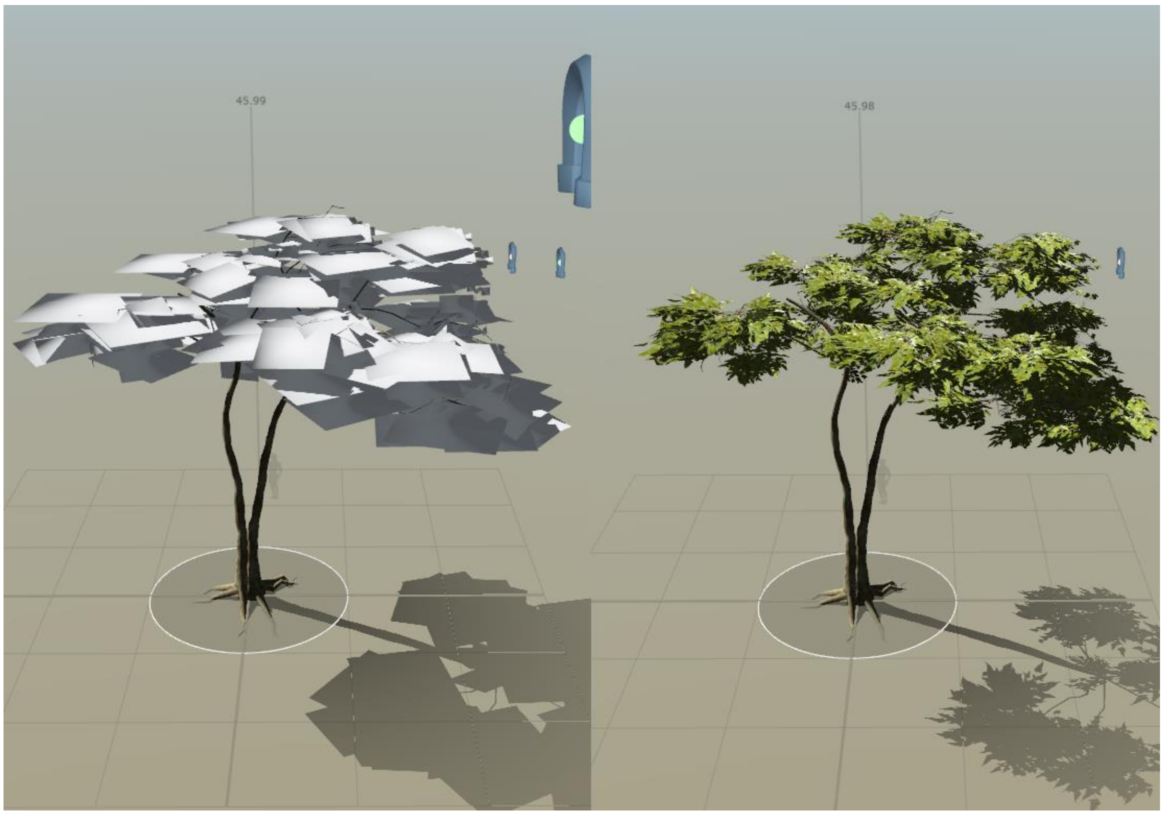


Figure 77: Tree creation in SpeedTree

We created a total of 4 variations of branch meshes and those mesh were used to create 5 different variations of the same type of tree used in the scene. And of course, all the trees have LOD's created by SpeedTree for better performance.

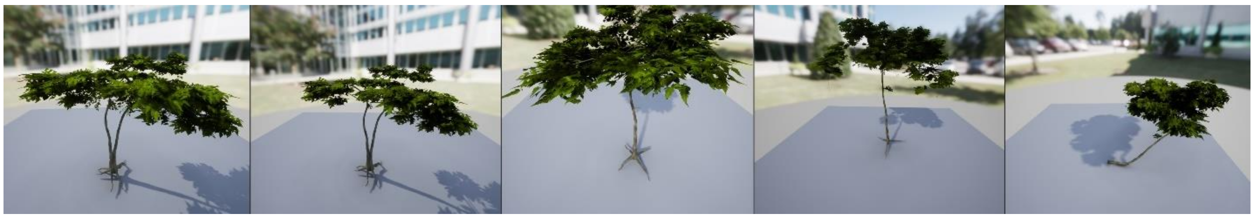


Figure 78: Trees Variations

AI

We have covered all the graphical techniques for the game, and it is a game, so we must create the gameplay. One big part of the gameplay is the enemy. They are Drones that protect the Relics we are trying to gather, to finish the game. So, the AI system used for the enemies lies down to the Tree Behavior system that Unreal provides along with custom C++ code.

First, we need to make clear what are the possible action the AI will take during the gameplay. We need for example the AI to patrol an area and looking out for enemies (the player) and then attacking that enemies. So, the final actions of our AI are the following:

- Patrol a marked area
- Make the AI able to sense with the following senses:
    - Sight
    - Hearing
    - Damage
- Make the AI able to follow the player while remaining a distance between AI and Player
- Attack the Player
- Dodge the Player Hits

The Behavior Tree we made for the AI looks like this:

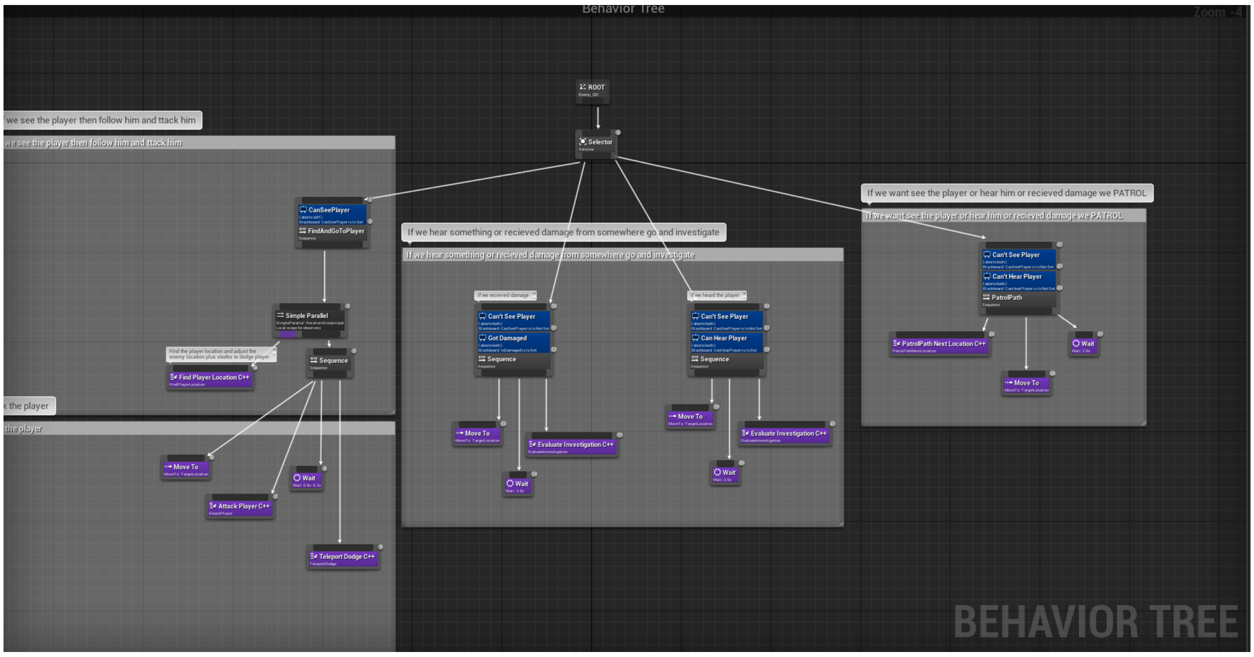


Figure 79: Behavior Tree

Each purple box is a task that the AI will execute, written in C++ for the best performance. On the right side we have the patrol actions. What those do is to first validate that the AI cannot see or hear the player, and then will reference the patrol path (C++ class) that holds the vectors or the points of the path and will designate the next point that the AI will have to move to.

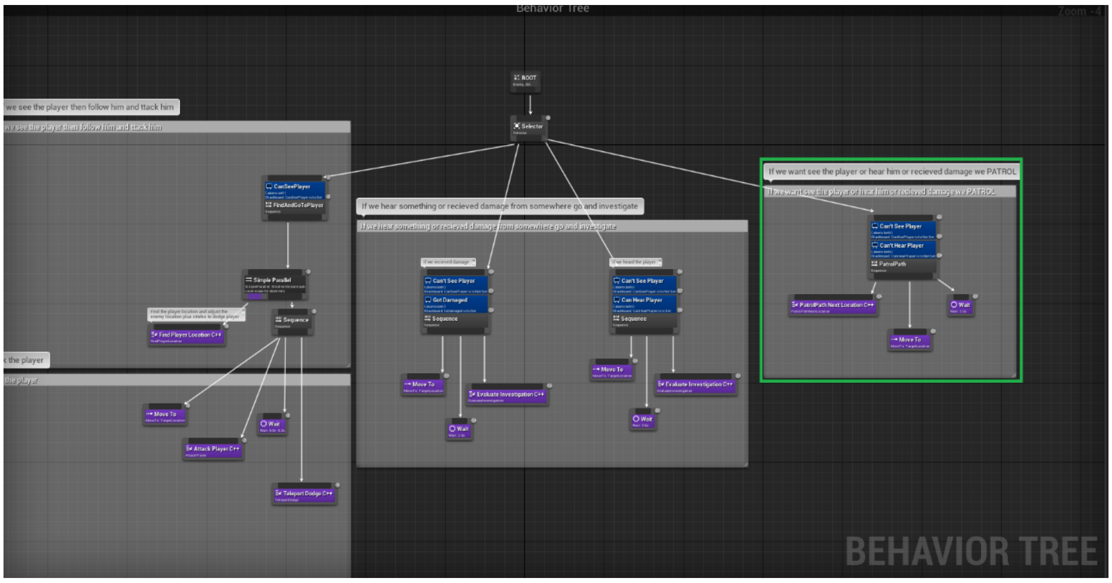

Behavior Tree
ROOT
Selector
we see the player then follow him and ttack him
CanSeePlayer
FindAndGoToPlayer
Simple Parallel
Sequence
Find Player Location C++
Move To
Attack Player C++
Wait
Teleport Dodge C++
If we hear something or recieved damage from somewhere go and investigate
Can't See Player
Got Damaged
Can Hear Player
Evaluate Investigation C++
If we want see the player or hear him or recieved damage we PATROL
Can't Hear Player
PatrolPath
PatrolPath Next Location C++
BEHAVIOR TREE

Figure 80: Patrol path on BT

Next, we have the Investigation part of the BT. What is this essentially is a way to make the AI investigate certain events. Each time the player makes a noise or hits the AI, the later one will move to the point where the event took place.

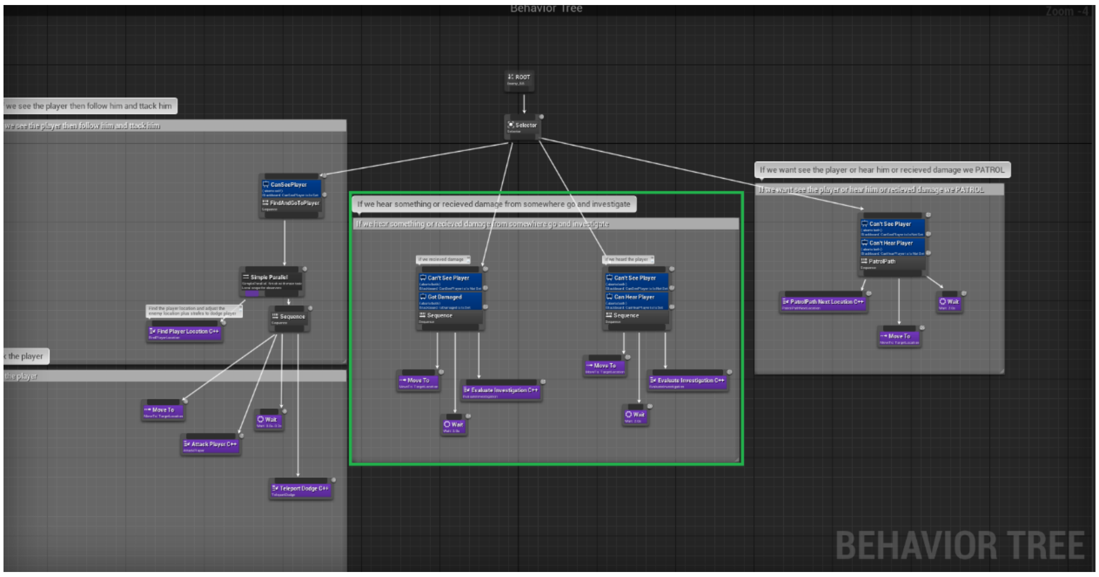

Behavior Tree
ROOT
Selector
we see the player then follow him and ttack him
CanSeePlayer
FindAndGoToPlayer
Simple Parallel
Sequence
Find Player Location C++
Move To
Attack Player C++
Wait
Teleport Dodge C++
If we hear something or recieved damage from somewhere go and investigate
Can't See Player
Got Damaged
Can Hear Player
Evaluate Investigation C++
If we want see the player or hear him or recieved damage we PATROL
Can't Hear Player
PatrolPath
PatrolPath Next Location C++
BEHAVIOR TREE

Figure 81: Investigation on BT

And lastly, we have the attack actions. When the AI has vision on the player this part of the tree will be activated. Here the AI will try first and go the right place to attack the player. Because the AI is a ranged type attacker, we need to make sure the AI will not go too close to the Player and it will always respect a distance. We can make that happen for sure with this equation.

```
float distance_ratio = enemy_ref->preferredDistanceFromPlayer / distance_between;
float x = playerlocation.X + distance_ratio * (enemylocation.X - playerlocation.X);
float y = playerlocation.Y + distance_ratio * (enemylocation.Y - playerlocation.Y);
```

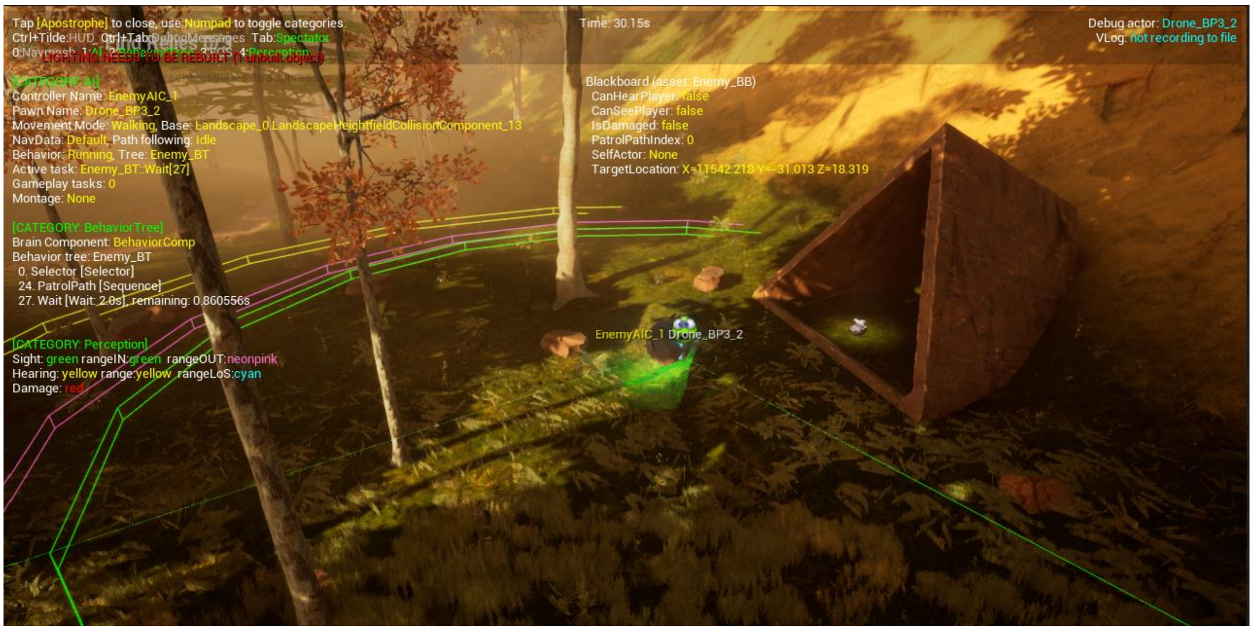
“code from the C++ class UBT_FindPlayerLocation.h”

Where the enemy_ref->preferredDistanceFromPlayer is a variable that we can define and change during the runtime, and it specifies how far we want the AI to be from the player. The distance_between is the distance between Player and AI. And then we find the X and Y coordinates for the point we want to move to, and we will plug those coordinates to the Navigation System to output the nearest available location in the navigable area.

When the AI has found the appropriate position to move, then it performs that move with the MoveTo task. We need to adjust the position of the AI constantly because the fire range of the AI is predefined by the User and the AI cannot fire from everywhere, so it needs to be in its fire distance. When that is achieved then the Tree will run the Attack Task, which will trigger an event on the AI class that will cause a projectile to be fired from the define sockets of the mesh of the enemy. After the Fire Event there is a Wait task to make the AI stop firing for 0.8 – 1 seconds until firing again.

At last the AI after that has fired once will try to perform a dodge move. This is to make it more difficult for the Player to target the AI. The enemy essentially will teleport 100-200 units to the right or left based on the Navigation System finding a proper location that it is navigable for the AI. That move also will trigger other event such as particle spawns to make the teleport move more pronounced and appealed to the User. The actions for finding the proper location for AI while Firing projectiles and dodging the hits, are all being done simultaneously, thus the node “Simple Parallel” that connects all these tasks.

Below we can see the radiuses that define the senses of the AI along with other debugging info. With the green radius representing the sight radius, the pink is the sight radius where the AI can lose sight, and the yellow radius is the hearing radius where the AI can pick up noise for it to investigate.



```
Tap [Apostrophe] to close, use Numpad to toggle categories.
Ctrl+Tilde:HUD  Ctrl+Tab:DebugMessages  Tab:Spectator
0:Navmesh  1:AI  2:BehaviorTree  3:EQS  4:Perception
LIGHTING NEEDS TO BE REBUILT (1 unbuilt object)

[CATEGORY: AI]
Controller Name: EnemyAIC_1
Pawn Name: Drone_BP3_2
Movement Mode: Walking, Base: Landscape_0.LandscapeHeightfieldCollisionComponent_13
NavData: Default, Path following: Idle
Behavior: Running, Tree: Enemy_BT
Active task: Enemy_BT::Wait[27]
Gameplay tasks: 0
Montage: None

[CATEGORY: BehaviorTree]
Brain Component: BehaviorComp
Behavior tree: Enemy_BT
 0. Selector [Selector]
 24. PatrolPath [Sequence]
 27. Wait [Wait: 2.0s], remaining: 0.860556s

[CATEGORY: Perception]
Sight: green rangeIN:green  rangeOUT:neonpink
Hearing: yellow range:yellow  rangeLoS:cyan
Damage: red
```

```
Time: 30.15s

Blackboard (asset: Enemy_BB)
 CanHearPlayer: false
 CanSeePlayer: false
 IsDamaged: false
 PatrolPathIndex: 0
 SelfActor: None
 TargetLocation: X=11542.218 Y=-31.013 Z=18.319
```

```
Debug actor: Drone_BP3_2
VLog: not recording to file
```

Figure 82: AI Debug info

We can see from the picture above which task is the AI executing at the exact time, and what are the values of the conditional variables (CanHearPlayer, CanSeePlayer, etc.).

To summarize we have an AI Controller that controls the pawn we assign it to, in our case the Drone_BP, which has all the logic inside of the senses (Sight, Hearing, etc.), and then we have the Behavior Tree that the AI Controller references to handle all the logic of the enemy, and lastly we have the Blackboard component where we store all of the conditional values from the tasks.

Missions

The goal here was to have a robust, and scalable missions' system that will handle all the missions that we create in the editor. First, to demonstrate the mission's system and make a small gameplay story for tis project, we conceptualize 3 different missions.

The first mission is to gather the missing relics scattered around the map, which are 3 in number. After we complete that mission, the next one is to craft the special item, and with the completion of that we have to take the crafted item to the "final tower", to deposit it there and release a beam of light. That is as far for the story of the game. In this project we recognize a pattern of sequential missions as we call it. That means each mission to be available needs another mission to be completed first, and this is one type of mission. Then there are the non-sequential missions. These missions do not require a previous mission to be completed, instead they are always active and ready to be complete.

The concept is to have a mission controller (MissionController.h) that will handle all the missions we create, if the are registered with this controller. After we create the mission controller with all of its functions such as storing all the missions, register them, and notifying various parts of the game like the UI when a mission is updated (ex. When we gather one of the missing relic). And then we expose this class to a blueprint for an easier control over this component.

Next, we have the missions. All missions inherit from the MissionInterface.h which is responsible for registering, updating, and finishing a mission. And then we have created a class called Mission.h which inherits from the interface we talked earlier and use that class to expose as our Blueprint mission class. So, all our missions now are blueprints inheriting from the Mission.h, and that way every mission that we create is controllable from our mission controller which only recognizes MissionInterface.h classes.

The logic of each mission goes into the blueprint we create every time, where we specify by which controller this mission will be handled by, what type of mission it is, and in which order this mission will be executed by declaring which missions this mission need to be completed first.

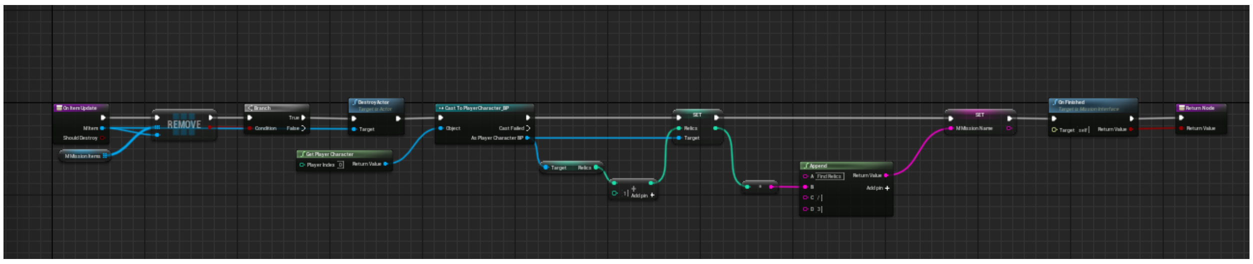


Figure 83: OnUpdate of the Find Relics mission inside the blueprint

Now we will see how the Find Relics mission is created. First, we need to have a class for our mission items that we gather. Every mission blueprint we talked about that we inherit from comes with a TArray that stores all the MissionItem.h references we have. Every mission item mesh comes with a collision profile, and it is where we assign the delegate event that will notify the mission when we overlap with the Player. We also expose the MissionItem.h to a blueprint class to be able to change the mesh of the collectable item fast without recompiling the code and add logic to the item. The item in our project is the following:

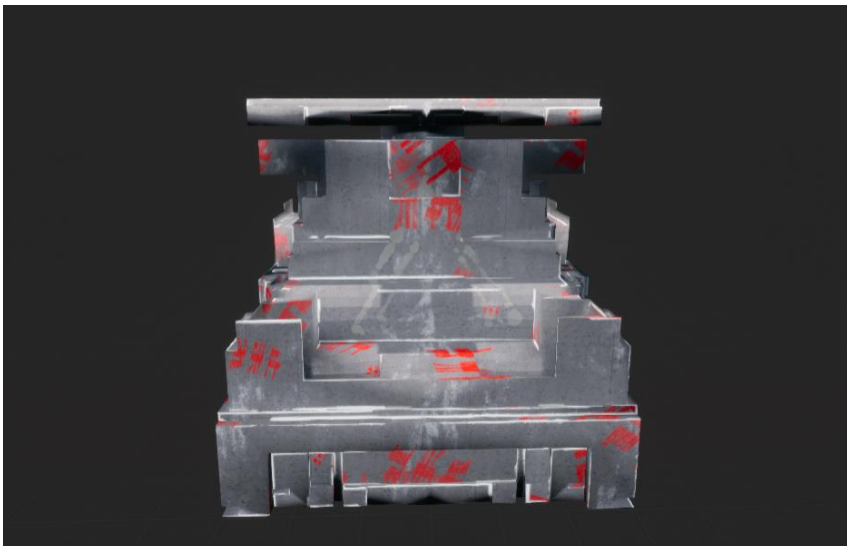
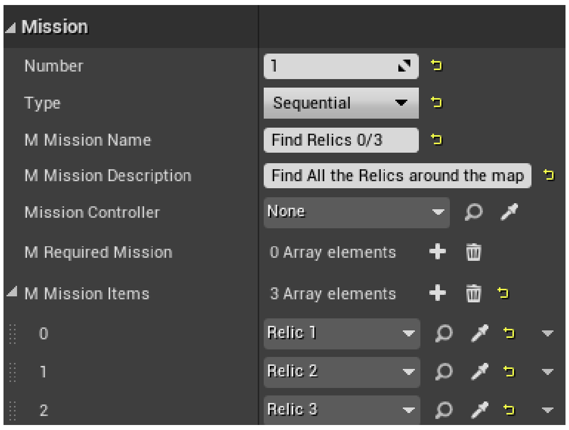


Figure 84: Relic (MissionItem.h)

We have 3 of those scattered on the map and need to be collected. Each one has a particle system inside that is activated when we gather the item, a pickup sound that player when we interact with it and notifies the mission which it belongs to. The mission itself store the references of these object in the scene along with other information we assign to it like the controller and the required mission (in this case none since it is the first mission).



Mission	
Number	1
Type	Sequential
M Mission Name	Find Relics 0/3
M Mission Description	Find All the Relics around the map
Mission Controller	None
M Required Mission	0 Array elements
M Mission Items	3 Array elements
0	Relic 1
1	Relic 2
2	Relic 3

Figure 85: Find Relics Mission details

The blueprint of this mission holds the logic of its completion. It handles what happens when we gather one item and when we have gathered all the items.

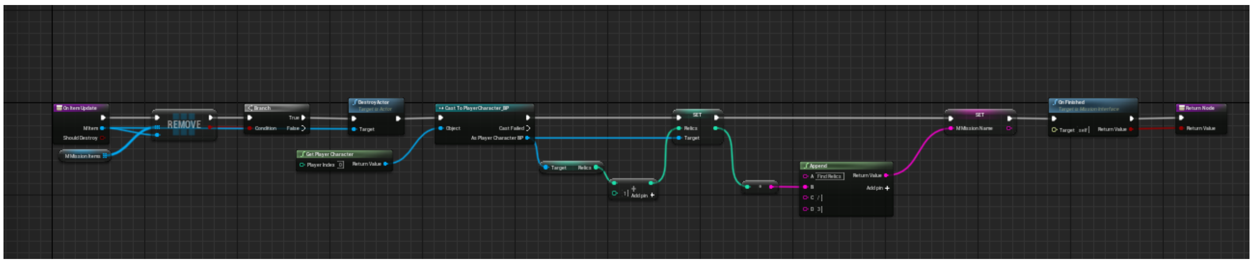


Figure 86: Find Relics OnUpdate (when we pick up one MissionItem and calls OnFinished)

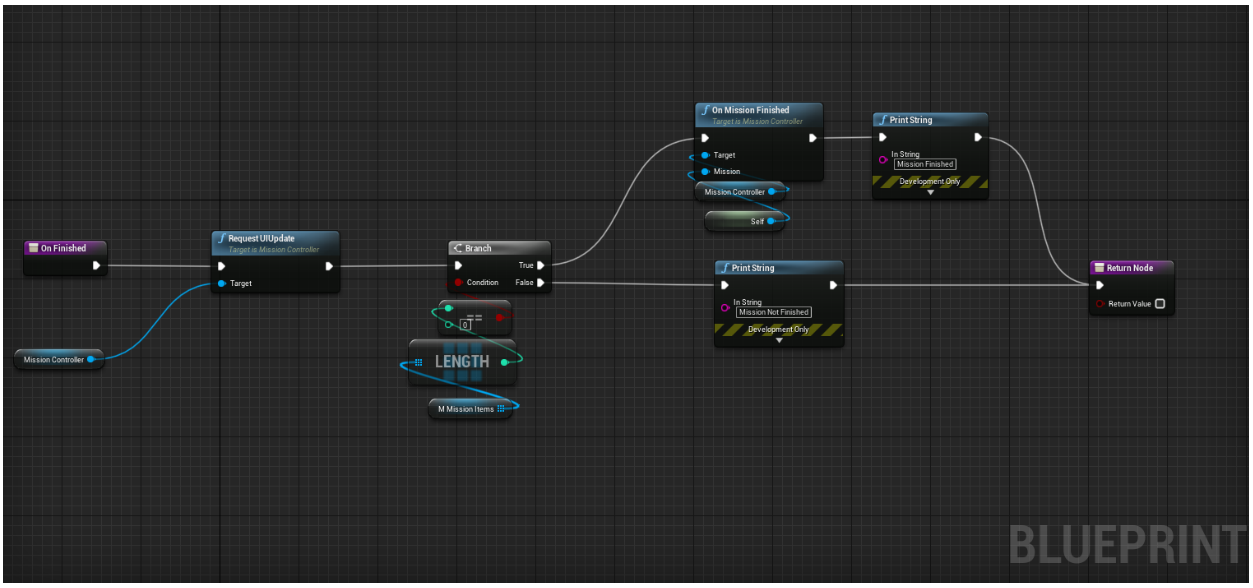


Figure 87: OnFinished, checks if we have completed the mission

The Player

For the player character now, we have a mesh (arms only since it is an FPS game) for the character, and a weapon all created and animated through blender using the addon created by Epic Games "Send to Unreal" which help to export the animation and the meshes to unreal faster and with the correct values adjust for the fbx export (such as scale of the model). The player can Walk, Run, Crouch, Jump and Mantle on static meshes. The mantle system is tweaked version of the ALS v4 Locomotion System on the Marketplace, to fit our needs with our character. The character can also shoot at the enemy.

Let us see first how we created and rigged the mesh of the character and the weapon in blender. As we talked about earlier the character has only the arms modeled, which are just the standard arms extracted from a basic human mesh.

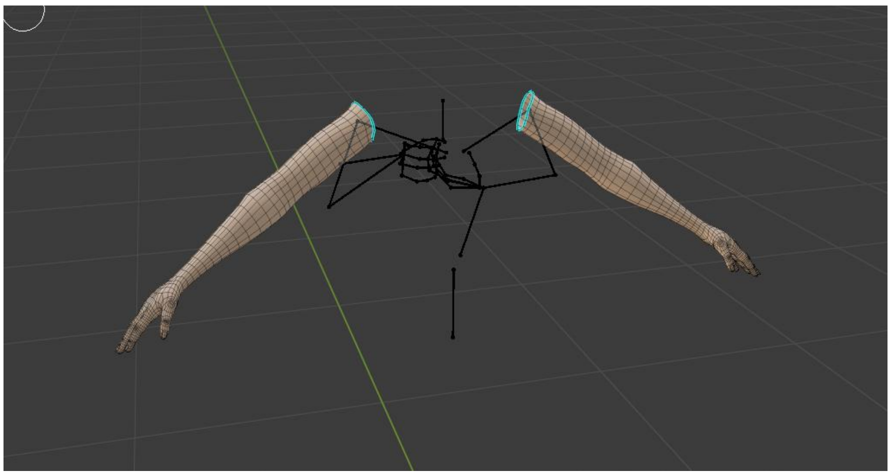


Figure 88: FPS Arms modeled in blender

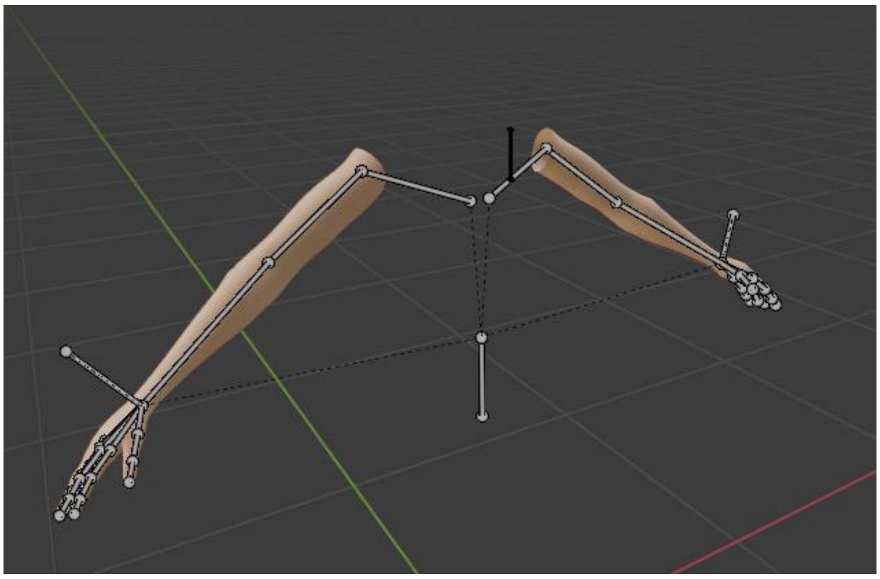


Figure 89: FPS Arms Rigged

And we also created the weapon mesh that we attach to the arms later inside the engine using a bone socket to follow all the animations created inside blender. For the modeling of the weapon the process of the hard surface modeling using Booleans used a lot in this project.

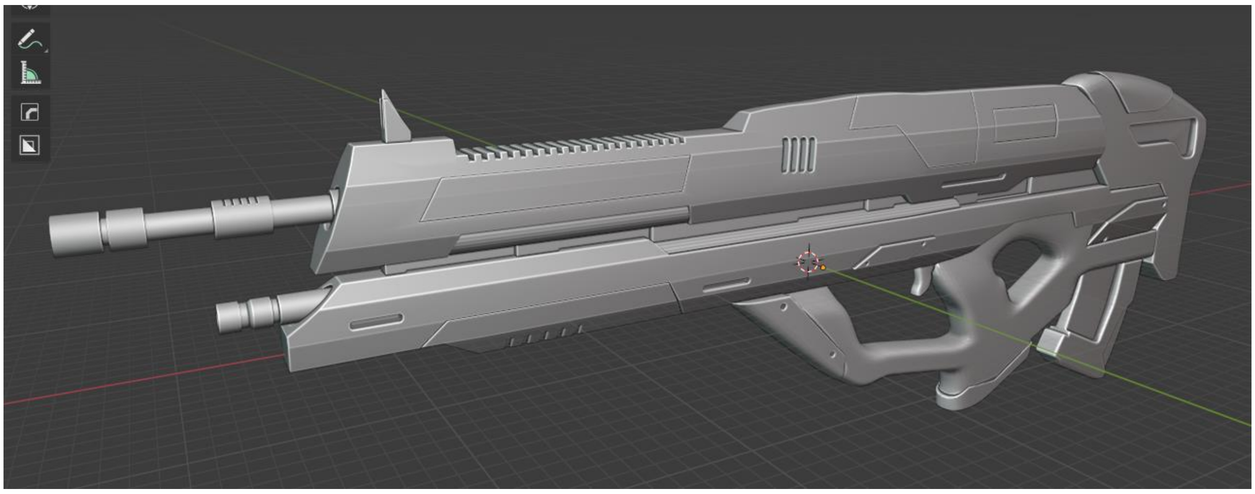


Figure 90: Weapon model inside blender

Following the asset pipeline methods, we talked about earlier we textured the weapon using Quixel Mixer.

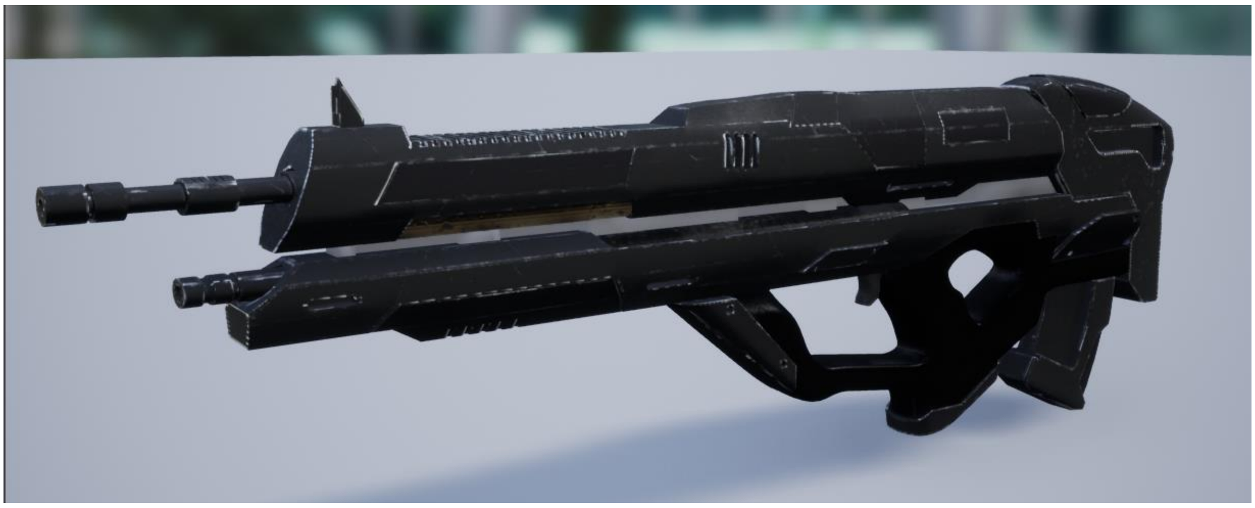


Figure 91: Weapon textured and rendered inside Unreal

The same way we created the missions using C++ classes and exposing them to blueprints, is the way we created our character. The base is done in C++ and more of the logic happens inside the Blueprints.

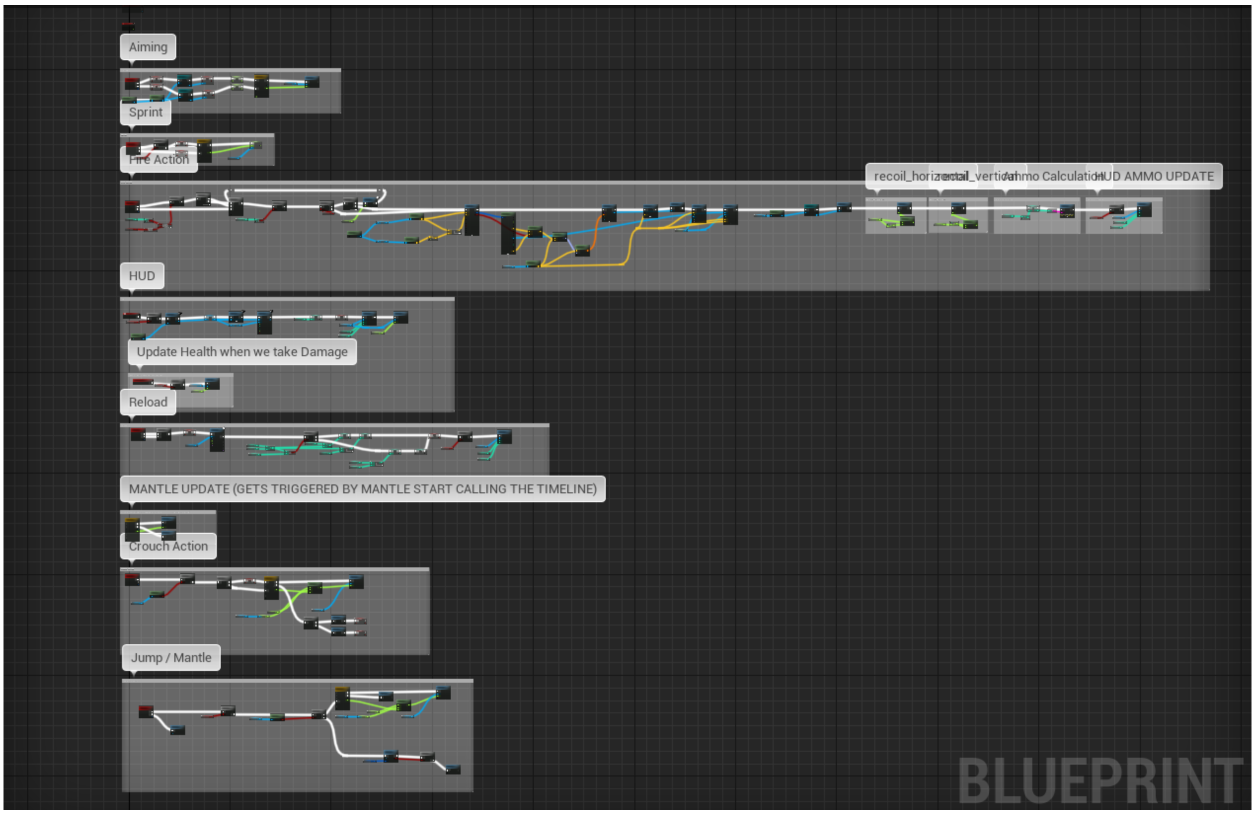


Figure 92: The character even graph Blueprint

Here are most of the events that the player can have. Things like Aiming where we activate the animation and changing the FOV of the camera, Sprinting, Mantling, and Firing our weapon are being handle here. For example at the Fire Action section we project a line from the middle of the screen to the world, then we take the vector hit result and the vector of the muzzle location we defined on the weapon mesh (where our bullet will spawn) and we spawn a projectile with the rotation to follow the line we projected from the muzzle location. After that we have some more things to handle. First, the damage from the bullet to the enemy will be handled from the projectile class itself so we do not have to do anything else for that part. But what we need to make sure we do is the following:

- Spawn the Noise Event for the AI to pickup if its close enough
- Spawn the Damage Event for the AI if we hit the AI
- Spawn the particles of firing the weapon
- Play the weapon firing sound
- Adjust the recoil of the weapon
- Update the ammo counter

Mantling is also being done inside blueprints. As we said earlier most of the logic came from the ALS v4 Locomotion system from the marketplace, but a few tweaks went into it to make it work for our character. This system is divided in 4 functions, one that checks if we can perform a mantle called Mantle Check, one to start the mantle called Mantle Start, one for updating the mantle in case we have a moving object that we want to climb on called Mantle Update and we also have the Mantle End which completes the mantle, and essentially what is does is to bring the character back to normal state.

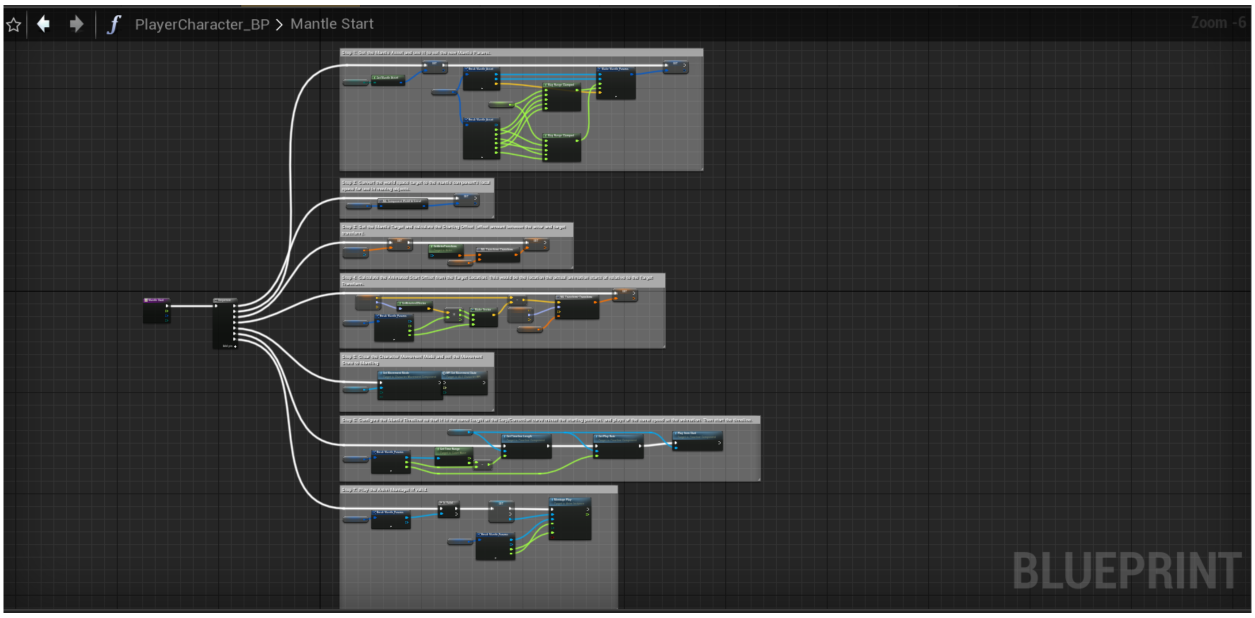


Figure 93: Mantle Check

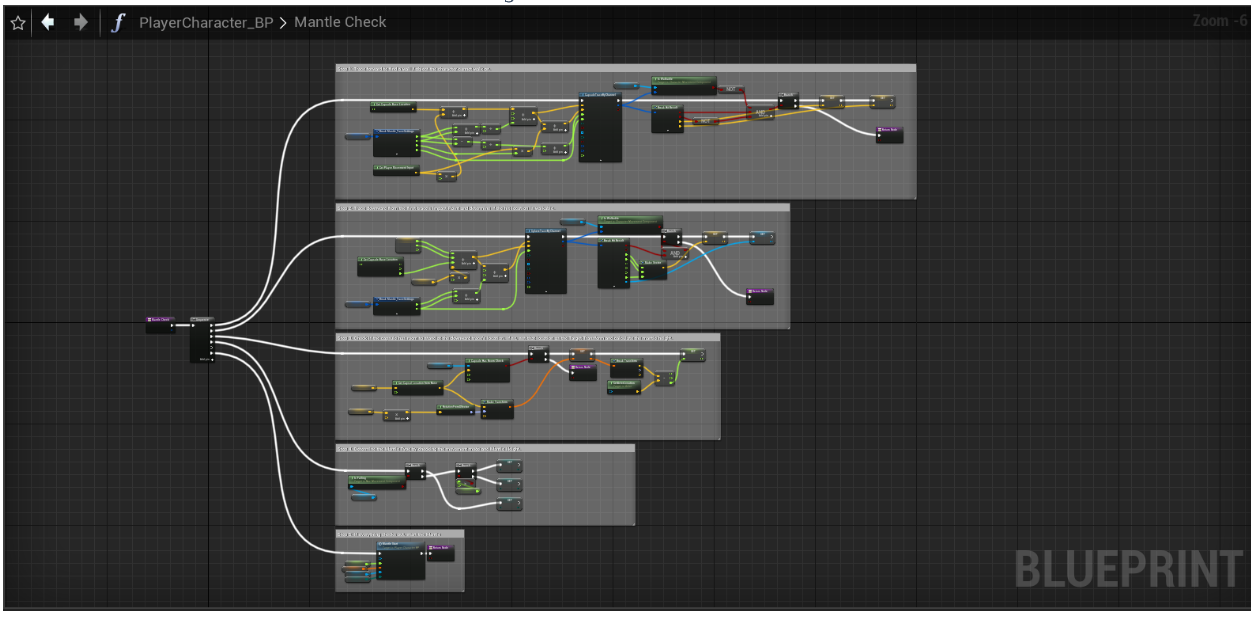
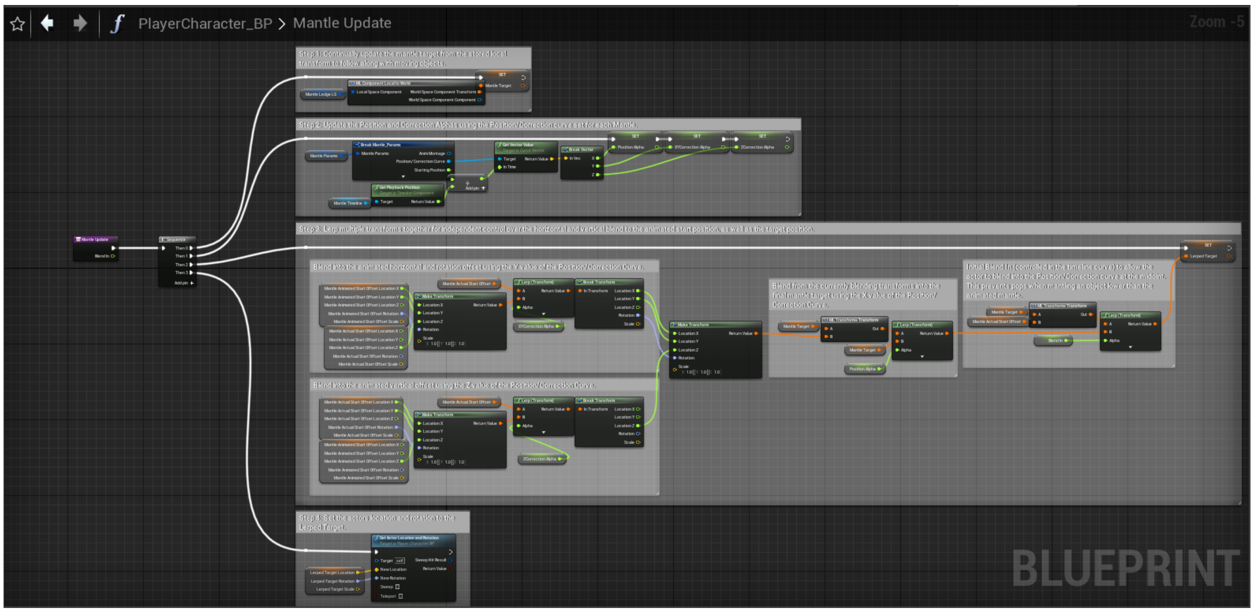


Figure 94: Mantle Start



Step 1: Continually update the mantle target from the stored local transform to follow along with moving objects.

Step 2: Update the Position and Correction Alphas using the Position/Correction curve set for each Mantle.

Step 3: Lerp multiple transforms together for independent control over the horizontal and vertical blend to the animated start position, as well as the target position.

Blend into the animated horizontal and rotation offset using the Y value of the Position/Correction Curve.

Blend into the animated vertical offset using the Z value of the Position/Correction Curve.

Blend from the currently blending transforms into the final mantle target using the X value of the Position/ Correction Curve.

Initial Blend In (controlled in the timeline curve) to allow the actor to blend into the Position/Correction curve at the midoint. This prevents pops when mantling an object lower than the animated mantle.

Step 4: Set the actors location and rotation to the Lerped Target.

Figure 95: Mantle Update

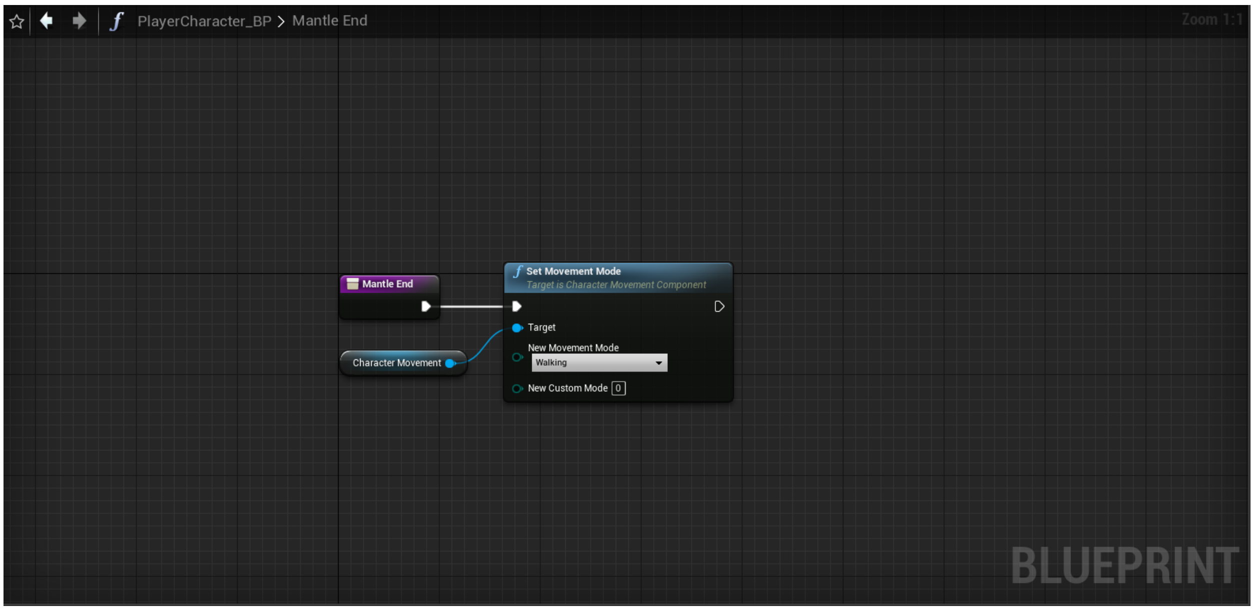


Figure 96: Mantle End

Particles

For the particles in the scene the Niagara system was used. It is the new particle system that Unreal provides us, and the particles now have some special features such as coding them in the blueprint editor and be able to communicate with other objects in the scene. We generally use particles when we interact with the game, like fire from the weapon, hit the target, gathering and crafting items. In the next example we are going to look how the particle effect of the crafting interaction was created, that looks like this.

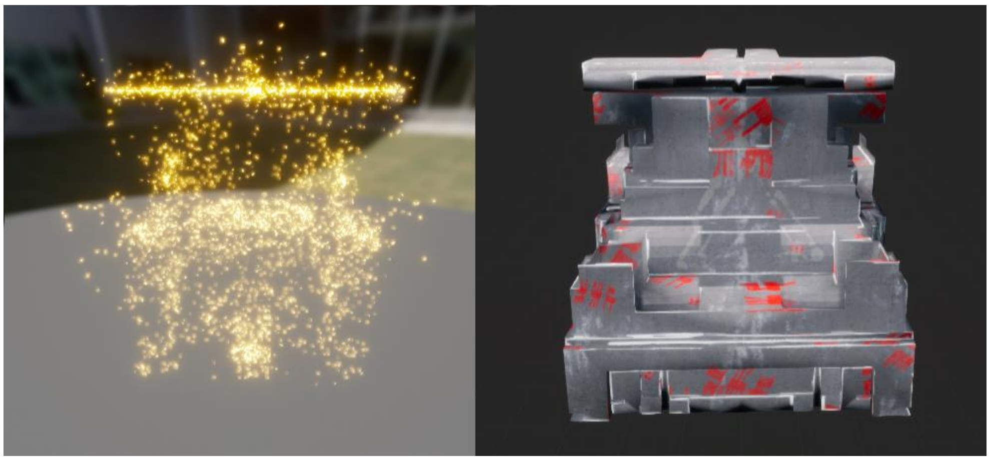


Figure 97: Particle Emitter

We can see that the particles are spawn to look like the mesh (relic) on the right. That is because we can communicate data for the particle simulation. What we did is to take the mesh of the relic and spawn particles randomly in each vertex of the mesh.

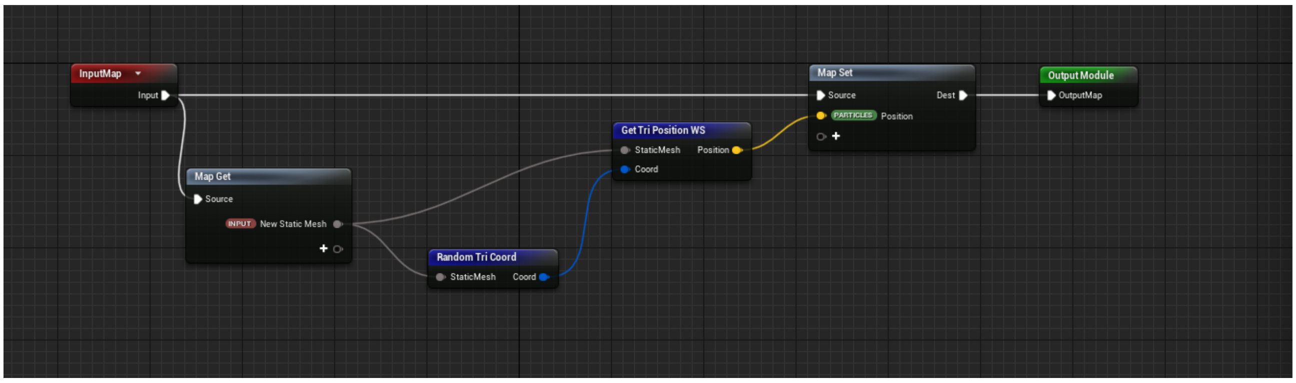


Figure 98: BP for the Mesh data

Final Overview

Here are some photos of the final scene complete.

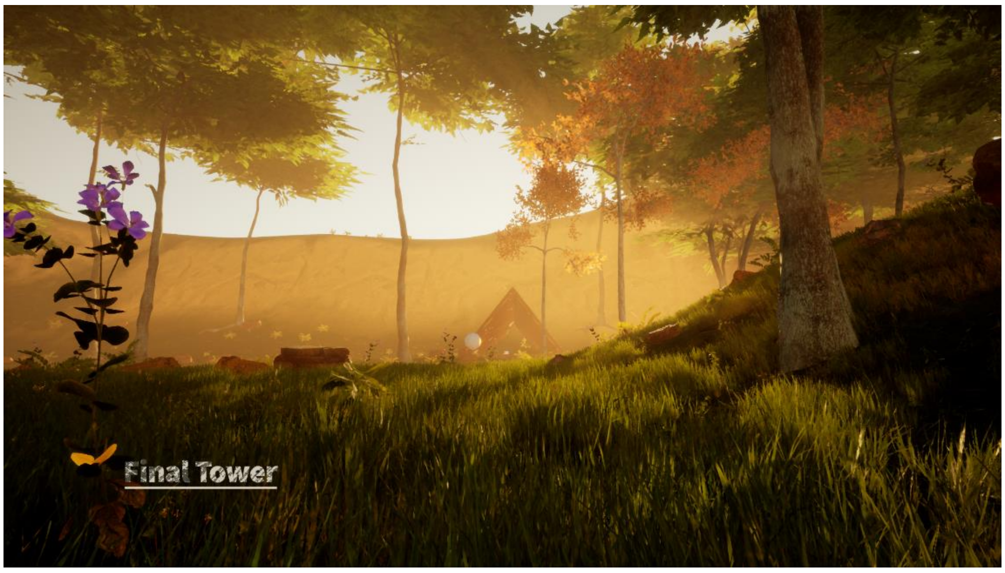


Figure 99: Final Image 1

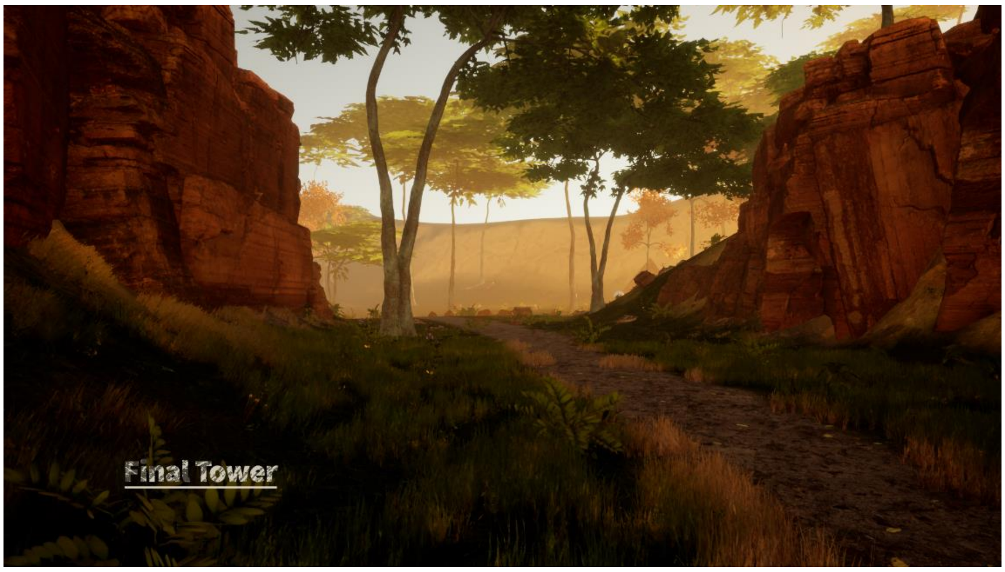


Figure 100: Final Image 2

And some more,

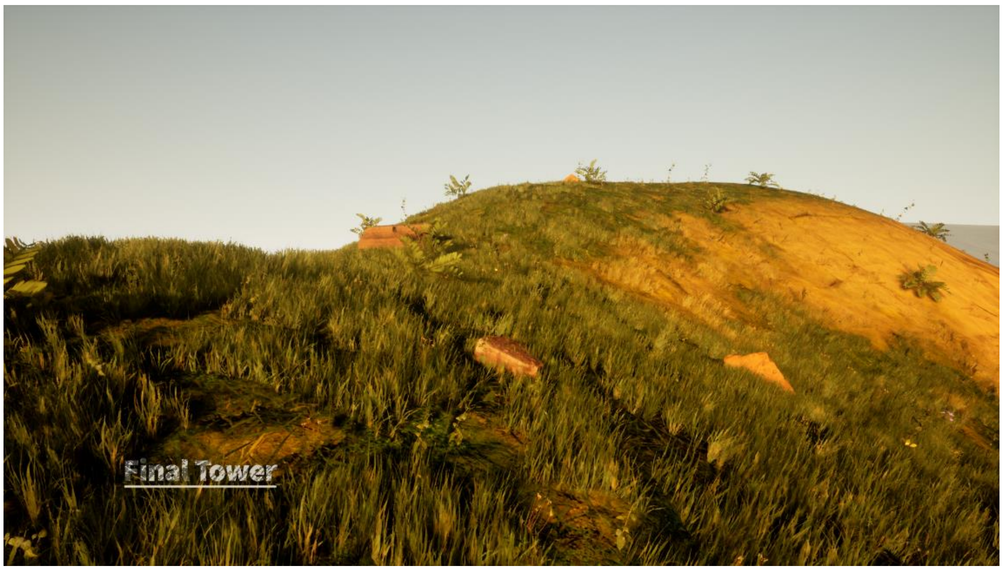


Figure 101: Final Image 3

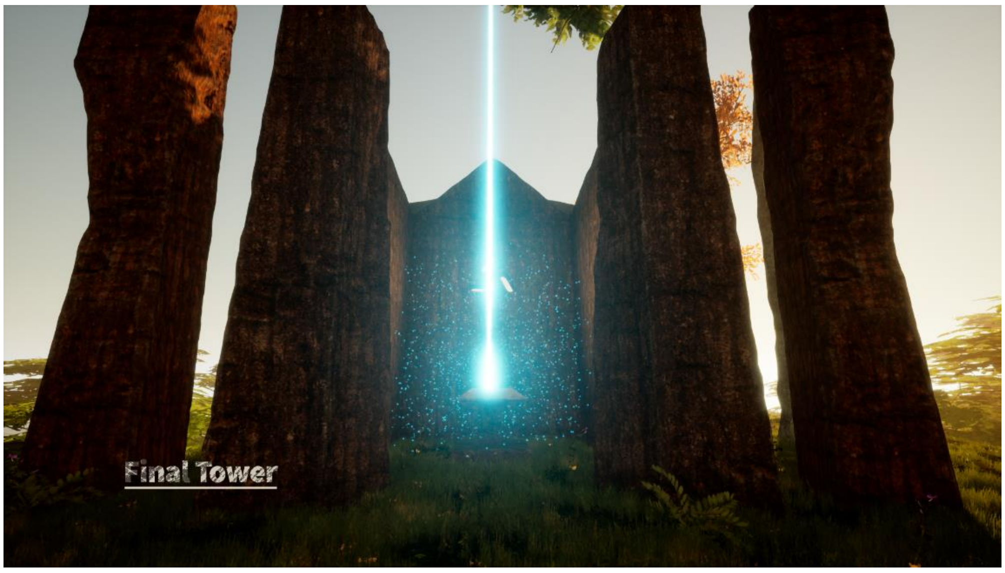


Figure 102: Final Image 4

Results

The game runs very good in maximum settings and that was the goal after all, have the best settings and great performance. On average we hit more that 70-80 fps, we never dip bellow 60, and there are many times we get 100+ fps. This result is achieved with an 1080p resolution and Epic settings, with memory not exceeding 2000mb, in an average hardware (Ryzen 2600x, RTX 2060 Super, 16 GB ram). The results look very promising, and the good thing is that they scale easy, making it also ideal for next-gen platforms.

Conclusion

Having achieved all the above, with delivering performance and quality in graphical fidelity, this project was a very valuable lesson, and a great learning experience. It is hard and takes a lot of time to develop a triple A game, and it is also hard for a solo developer. There are many areas which need to be explored more, that is why it takes huge team to develop the recent AAA games, because its area needs its own expertise. Modeling, Animation, VFX, Engine, Editor, Particles, Storytelling, and many more goes into creating a game, and to create a high quality one it a difficult task. Creating this project made me understand somewhat what goes through creating a game, and how many different technologies can be combined to achieve that result.

Feature Development

We could evolve this project in many ways. Lighting could be explored more, to create a more dynamic day/night system by baking the indirect lighting not just once but many times for different hours and blend them as the day progresses, but that would require many work on the engine level as this is not possible yet. Models could be more optimized and ready for shipping with better remeshing and accuracy to detail. Shadows are one more area we could see, because as of right now they take a big percentage of the render time.

And lastly, although we did some work on the gameplay, and the character, we could add a bigger story by adding missions, paying more attention to the character model and animations, and creating an open world game experience, by scaling up the landscape and add more variations to it.

References

[1] https://docs.unrealengine.com/en-US/Engine/Rendering/Overview/index.html
[2] https://docs.unrealengine.com/en-US/Engine/Performance/ForwardRenderer/index.html
[3] https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/VolumetricLightmaps/index.html
[4] https://docs.unrealengine.com/en-US/Engine/Rendering/LightingAndShadows/MeshDistanceFields/index.html
[5] https://docs.unrealengine.com/en-US/Engine/Niagara/Overview/index.html
[7] https://github.com/EpicGames/BlenderTools/wiki
[8] https://github.com/SavMartin/TexTools-Blender/releases/tag/exTools_2.8x
[9] https://twvideo01.ubm-us.net/o1/vault/gdc2018/presentations/TerrainRenderingFarCry5.pdf
[10] https://www.youtube.com/watch?v=wavnKZNSYqU&ab_channel=GDC
[11] https://www.youtube.com/watch?v=ToCozpl1sYY&t=1042s&ab_channel=GDC
[12] https://www.youtube.com/channel/UCnN2iiHapP1uzJq64wlg3Zw
[13] https://www.reddit.com/r/unrealengine
[14] https://www.youtube.com/user/RVillaniCG
[15] https://www.unrealengine.com/marketplace/en-US/product/advanced-locomotion-system-v1
[16] https://www.unrealengine.com/marketplace/en-US/product/megascans-goddess-temple