Report: 3D Terrain - rendering smooth was yesterday

Table of Contents

1 Introduction

This report is part of the assignment 5. It covers the 3D terrain rendering and especially the interesting parts of this project as well as a description how this was developed.

You can skip to the interesting parts using the table of contents or read the entire report. It should be an easy and hopefully fun read :).

1.1 Inspiration and backstory

I'm really interested in game development in general. This is what got me into this degree. I used to match *ThinMatrix* and *The Cherno* both are YouTubers that focus on game development from scratch.

I even tried to develop my own game engine but I gave up after several month, when I saw the source code of the *Unreal Engine*. Anyways my focus shifted from game engine development to 2D game development and low level stuff. It was at this time that ThinMatrix started to develop a low poly game. And I started to love the aesthetics of low poly games and objects. It's crazy how models with very little detail can look better than some fancy models.

I always played with the idea of coming back to 3D game development to create a low poly game. This and the compiler course was actually one of the main reasons why I choose *Reykjavik University* for my semester abroad. I couldn't be happier with my choice. I've come to a point where I can fluently interact with OpenGL. My focus will be on other projects after this semester but I hope that there will be time to continue working on something like this. Vulkan and RLSL (Rust Like Shading Language) are two topics that I would like to look into if there is the time :)

1.2 Goal

At the start of this project I defined the following goals:

1. Create a low poly terrain with water animation and lighting support
2. Use a geometry shader somewhere in the project
3. Create a performance friendly particle system (Maybe like the one used in old LEGO Star Wars games)
4. Have a better and reusable project structure
5. Calculate as much as possible in the shaders them self (This one was added during development)

Most of these goals are self explanatory. However, I want to add some notes to 2. and 4.:

- My goal is always to learn, this is the reason why I chose this course. I believe that I'm fairly fluent with vertex and fragment shaders but I've never tried to work this geometry shaders. I believe that these shaders add so many abilities that I really wanted to work with it

- I knew that I wouldn't be able to create an entire game with learning a new shader type. So, I set my self the goal to create an architecture that is good enough that I can continue working with it after the end of this course.

Okay, that is enough introduction for now. Let's get into some fun and technical stuff

2 Geometry shader

This section will try to give a really really rough overview of the geometry shader. Further information should be taken from the documentation it self. The implementation section will go into more detail about how what feature was used inside the project.

2.1 Formal information

Our course has covered the vertex and fragment shader. In this project I wanted to specificity take look at the geometry shader. This shader sits between the vertex shader and the *Tessellation* stage. It can operates *primitives* like points, triangles and lines.

This shader receives the primitive vertices with their data and can emits a set amount of vertices with vertex data it self. The number of possible vertices it determined by the size of the vertex data and the hardware. OpenGL 4.3 which I'm using for these shaders defines a minimum size of 1024 components with one float having a component size of one.

2.2 My view

The definition was an over simplified definition of a super power powerful yet uncomplicated tool in OpenGL. In this section I want to give a even rougher view on the shader that worked well during my development.

The geometry shader is basically a powerful vertex shader that has access to all vertices of the triangle (or other primitive) that gets drawn. It is also the only shader that can create more data it self. This and the fact that it operates on multiple vertices at once distinguishes it from the other shaders and makes it so powerful in comparison.

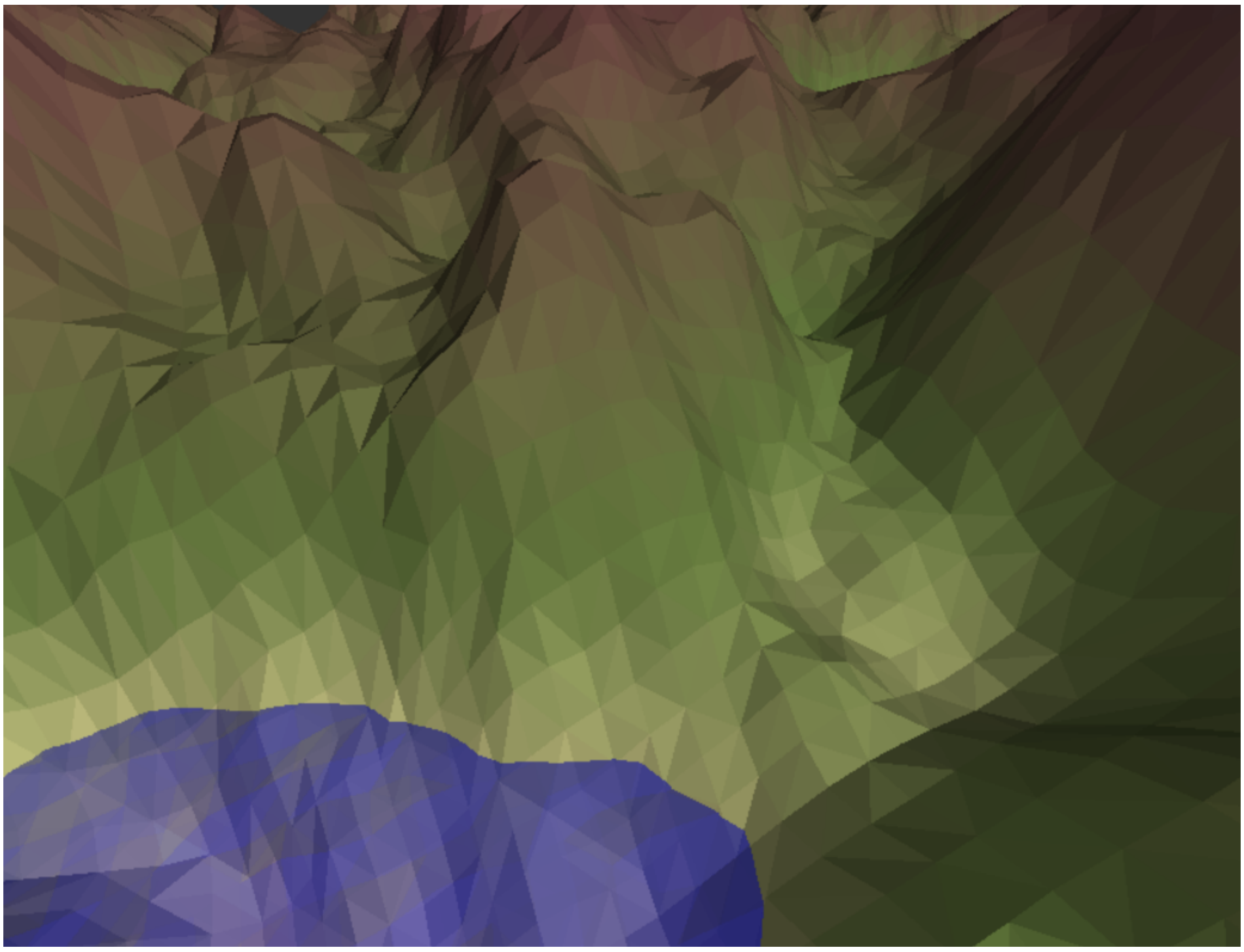
I'm sure that my implementations hasn't even scratched the surface of what is all possible with this shader.

3 Implementation

This section will go into detail about the challenges I've encountered during the development and how the systems work them self. It will only go into detail about interesting and new stuff. This means that I will not explain every line but more the concepts that where being used.

3.1 Art style

The art style this project uses is called low poly and is characterized by models with a very low polygon count. This leads to clearly visible edges. This art style embraces this look by making the sides and edges quite sharp. Here is an example image from the game itself:



The interesting part about this art style is the way this effect is archived. Light plays a major role when it comes to 3D-rendering in general. Surfaces use normal vectors to calculate the effect light should have. Each vertex has it's own normal vector, OpenGL then interpolates these inputs before processing them in the fragment shader. This archives a smooth surface effect.

Low poly does the complete opposite by using the same normal vector for an entire face. This makes edges look hard and causes a almost constant light effect. This usually goes hand in hand with single colored faces. It is actually quite simple for how different it looks.

3.2 General

Here I want to explain some general concepts I used during my development with geometry shaders. Mainly the the `AVG_INPUT(x)` macro I use in both shaders. This macro averages the input of the given vertex Data into a single value. It does this by adding them together and just dividing them by three. It's actually quite simple.

3.3 Terrain

The terrain rendering is only done in the vertex and geometry shader.

3.3.1 Vertex shader

I've tried to reduce the vertex data as much as possible and I ended up just passing the texture coordinate of this vertex for height_map. The vertex shader than samples the color of the height_map at the given coordinate and divides the sum of the RGB values by three. This returns a value between 0.0 for pitch black and 1.0 for snow white.

The texture coordinates are used for the x and z value of the position and the y is set via the height_map. This vertex position is than scaled by some defined constants in the vertex shader. This gives us a Terrain that is varying in height. Note that an offset is added to the y value to have the water level be at 0.0. The actual terrain height can range from -30.0 to 70.0. The shader also apples the transformation matrix to the position.

I, le Fred, also wanted to have a color effect depending on the height of the terrain. I first defined several materials I want to have to the different height. I than needed to select two materials to interpolate between. This is done by using the previous height value and scaling it my the amount of defined materials. In our case 5. This gives us a value in the range from 0.0 and 5.0. This value is than simply converted into an integer and used as an index for the material list. The fraction is directly used in the interpolation as an added bonus.

The position and material values are than simply passed to the geometry shader to deal with.

3.3.2 Geometry shader thingy

The geometric shader starts of by building the average for the input values like the position and material values. It than calculates the normal vector of the face by building the cross product of two tension vectors. This is fancy talk for taking two edges of the triangle and calculating the cross product that is now per definition perpendicular to the surface it self. (I'm actually quite proud of myself for this one. I know that others have done it before, but I came up with it when I was designing the shader)

The shader than proceeds to do the normal lighting calculation for the center point of the triangle. This process was documented in a previous report. I than apply the calculated color to the entire triangle.

The last step is just to apply the projection and view matrix to the position and pass the vertices and triangle onto the next stage.

3.3.3 The long fragment shader

The fragment shader just passes the input color as an output to the next stage.

Yep and that is that. It's actually surprisingly simple for such a cool effect in my opinion.

3.4 Water

This section will explain the water implementation in the project. The water is rendered using it's own shaders. However, we use the same vertex array like the terrain. This also enables us to implement vertex clipping below the terrain. Let's get into it.

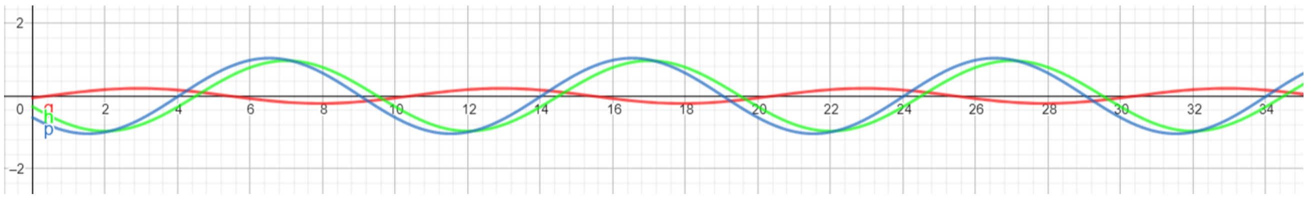
3.4.1 Vertex shader

The vertex input is again the texture coordinates of the height_map, like in the terrain shader. These are also again multiplied by defined constants to expand the water mesh to the size of the terrain. They Y value is set to 0.0 for now. This leaves us with a flat mesh at y value 0.0.

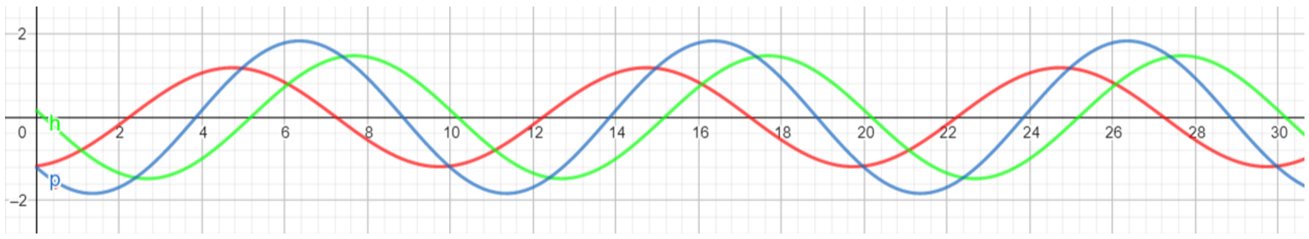
The wave effect

The basic idea to archive this wave effect is to apply a time and position based offset. The shader throws a combination of the time and the texture coordinates scaled by some value into a sinus and cosinus function to receives values -1.0 and 1.0. These values are than added together. The texture coordinates and are scaled differently for the x y and z offset. The exact code can be found in the shader it self. I want to add some visual explanations to the formula. I therefore visualized the offset calculations for three different world positions. The colors red, green and blue correlate with the x y and z offset.

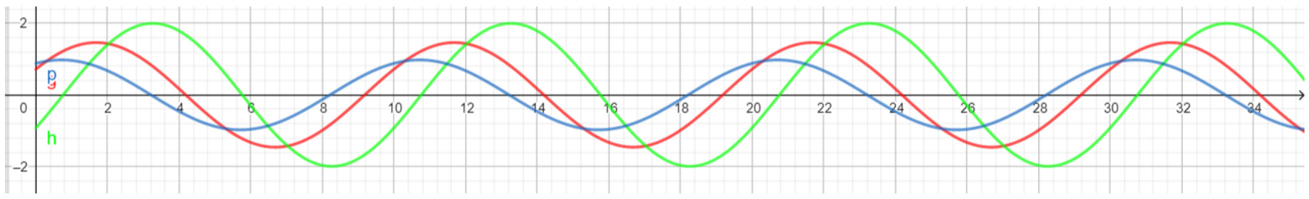
The wave offset for x= `0.3` and y= `0.2`



The wave offset for x= `0.5` and y= `0.4`



The wave offset for x= `0.6` and y= `0.6`



This pattern is by far not ideal but that's not the goal. All we want to archive is a human unpredictable pattern and a continues smooth motion and this definitely does the job quite well. Finding the right values for the offset and even the combination of the time and factors took quite some time, it was basically a lot of try and error with a lot of different constellations.

The position with the applied offset is than further passed to the geometry shader.

Clipping

An optimized game would usually calculate which water vertices can actually be seen by the player and wouldn't even generate the other ones. However, this simulation calculates the height of the terrain in the vertex shader and limits us therefor to clip the unseen water vertices beforehand. I still wanted to clip at least some of the water to gain a bit of performance.

The idea behind clipping water is basically just to remove all vertices where the terrain is above the sea level in our case >= 0.0. However, we want to make sure that all vertices are below the terrain before we clip the face completely. The vertex shader therefor only calculates if this vertex is below the terrain and passes this information to the geometry shader to deal with. Note that this calculation uses the y value with applied waves. This would even allow us to simulate a tsunami with the correct clipping calculations.

3.4.2 Geometry shader

Clipping

The geometry shader starts where the vertex shader has left of by checking if all vertices are below the terrain. It aborts drawing this triangle if all of them are invisible. This saves us from doing the lighting and alpha calculations.

Water alpha value

The water opacity is calculated using a formula Fresnel that describes how light gets diverted when it hits a different optical media. I sadly have to say that I didn't quite understand the formula it self. I decided to simply copy the equation and move on to a more interesting aspect.

3.4.3 Fragment shader

The fragment shader simply takes the given color and alpha value and applies it to the pixel it gets drawn on. The actual blending it self is done by OpenGL it self. I used the normal alpha blending with `gl.glBlendFunc(gl.GL_SRC_ALPHA, gl.GL_ONE_MINUS_SRC_ALPHA)`. The water alpha was setup to simply draw over the color it self.

3.5 Particles

The particle system in this project is mostly a prove of concept that was inspired by an old particle system for the PS2 console. The basic idea of this system is that we only store the start values and the point in time when the particle was created. All changes get recalculates by the GPU each frame. This is actually a reasonable approach to implement particles in a game.

The particles are the last objects that get drawn before the the frame is presented. This allows for great blending magic.

This is in my opinion also a more interesting system as it combines a bunch of knowledge about computer graphics into a single working system. So let's dive into the explanation.

3.5.1 System setup

All information to draw a group of particles is stored in the `ParticleEmitter` component. This component is used to configure which particles should be drawn where and how. It's basically the control center for a group of particles. The component stores two types of data:

1. Data that is configured by the user, like particle spawn time, textures and the life time
2. Data that is manages by systems and should not be touched. These are the values are loaded into the uniform arrays. There is no validation before loading this data to improve performance.

The vertex buffer to draw the particles is just a simple array with IDs ranging from 0 to `MAX_PARTICLE_COUNT` which is currently set to 256. The entire magic is done by the shaders.

3.5.2 The vertex shader

The vertex shader is responsible for translating the input particle ID into particle specific data. The vertex shader first translates the ID into the position, sprite_id and other particle specific data. This is a simple uniform array access. The ID is used as the array index in this case.

This shader also applies all translations like defined movement or scaling. This data is than passed further onto the geometry shader that is responsible for displaying the single processed point into a rectangle that a texture can be rendered on.

3.5.3 The geometry shader

The geometry shader receives a single point with values how the particle should be displayed. The output should be a rectangle with a texture that is always rotated towards the camera.

Positioning

The rectangle calculations are similar to calculation the view matrix. The shader takes the `particle_position` and the `camera_position` to calculate the `normal_vector` that should be archived. This vector is called `forward` in the view matrix calculation. We than calculate the `right` vector with a cross product between the `forward` and the `camera_up` vector. The right vector is already a vector inside the plane we want to draw. We than calculate the `up` vector by building a last cross product between the `normal_vector` and the `right` vector. This gives us two tension vectors that are used to create map the rectangle corners.

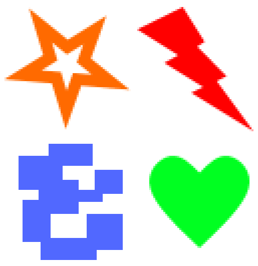
```
// ^        |
// |        | <-- The direct view from the camera onto the particle
// |        |     with x being the normal / forward vector
// x----->  |
```

I wanted to have the defined particle position be the center of the drawn rectangle. I therefor translate the tension vectors into offset vectors. This fives us the following construct:

```
// 0       2   | --- := right vector like in the view matrix
//   \   /     |
//     x ---   | /   := offset1 = right * 0.5 + up * 0.5
//   /   \     |
// 1       3   | \   := offset2 = right * 0.5 - up * 0.5
```

Texturing

The last step in this shader is then to calculate the texture coordinates of these vertices. The particle system used a sprite sheet to have multiple textures available. A sprite sheet is basically a texture that contains multiple textures. This is an example of a sprite sheet that was used during testing:



We simply translate the normal texture coordinates from `0.0` to `1.0` to only map onto one single sprite. The range from `0.0` to `0.5` would only cover the star in the upper left corner. The geometric shader can therefor only accept a single ID and map it onto a specific sprite on the sheet when the row and column count of the sprite sheet is known. These values are provided via two uniforms.

We than calculate the sprite origin, width and height in texture coordinates and map these values onto the vertices. This example shows a slightly simplified version on how the 4th sprite would be mapped into the example texture:

```
// Sprite sheet position
int row = 1;
int column = 1;

float sprite_width = 0.5;
float sprite_height = 0.5;

vec2 origin = vec2(column * sprite_width, row * sprite_height)

// c1    c3
//
//
// c2    c4

vec2 c1 = (origin.x               , origin.y                );
vec2 c2 = (origin.x               , origin.y + sprite_height);
vec2 c3 = (origin.x + sprite_width, origin.y                );
vec2 c4 = (origin.x + sprite_width, origin.y + sprite_height);
```

These texture coordinates are than combined with the previously calculates positions and passed to the fragment shader.

3.5.4 The fragment shader

The fragment shader only receives a texture coordinate that should be mapped using the bound sampler.

Blending between the texture and the already drawn stuff is done by OpenGL. The `ParticleEmitter` component allows the user to set custom blending options if desired. The default is set to `[gl.GL_SRC_ALPHA, gl.GL_ONE_MINUS_SRC_ALPHA]` correlating no normal alpha blending.

3.5.5 Smooth movement

Note: I added smooth movement as a last addition after writing this report. I don't want to go into to much detail, because it's just the Bezier curve motion that we learned in class and used in the last hand-in.

The curve uses the spawn position and two randomly generated control points to calculate the motion. The implementation can be found in the particle vertex shader. In the future I would like a to have options how these are generated and maybe add multiple points. But that's a story for a different time. I just wanted to note that that is also now implemented :D.

4. Final thoughts

I really like the visual result of the simulation. It is really impressive to see, how simple such a visual representation can be at the end. That being said it took quite some time to get fully used to the geometry shader. This is of course logical but I still would have liked to get a bit more done, as usual. The particle system was quite a system to figure out but damn it was worth it. I'm super happy with how it turned out. There is still a bug with the drawing order and I would probably need to to the clipping manual it's still a system I'm proud of.

The structure and architecture of the entire project has also really improved since the last project. This also makes me very happy and increase the ability to implement new features.

And that's it! I hope you had fun reading this report and looking at the simulation it self!

5. Sources

- OpenGL: Geometry Shader. https://www.khronos.org/opengl/wiki/Geometry_Shader (2020.11.01)