TNM084: Procedural Terrain

Adrian Andersson

March 23, 2020

I Introduction

Procedural methods for computer graphics are used in order to compute geometries or textures without having to load additional files. This means that a procedural texture is not limited by the resolution of a pixel image.

Procedural geometries are often used to create landscapes such as mountain ranges or rolling hills. The procedural methods are most often accompanied with a set of noise algorithms such as Perlin or Worley noise that produce natural looking patterns. Other applications for Perlin and Worley noise include special effects such as fire, fog, smoke and imperfections in materials such as dirty or cracked surfaces to name a few.

This paper presents a project made for the course TNM084 at Linköping University where a procedural scene with a moving terrain in a thunderstorm is created. The terrain is based off an example from the *Three.js* gallery which unfortunately seems to have been removed from the internet since. The project is made with the JavaScript library *Three.js* which in turn uses WebGL.

II Method

The geometry used to create the hills is a plane. As vertices of the plane need to be displaced the number of segments in each direction of the plane need to be significantly higher than the minimum.

II. I Terrain

The hills are a result of displacing the vertices of the plane according to the noise level of a two-dimensional Perlin noise. The Perlin noise is implemented by using Stefan Gustavson's 2D noise functions for WebGL [1]. In order to achieve more natural looking hills, some fractals of noise are used, this results in smaller hills on top of bigger hills.

After the plane has been displaced the normals need to be re-computed in order to have the correct lighting. The new normals are calculated in the fragment shader by using finite differences to get both the tangent and bi-tangent of the plane in that particular fragment. The new normal is the cross product of the tangent and bi-tangent.

Because the plane is of finite size the edges are visible, this led to fog being added in an attempt to hide the edges. See figure 1 below. Another color is added to create a more reddish color in the valleys between the hills, the sole purpose of this is to mimic the example which also had a mix of colors based on the altitude of the plane.

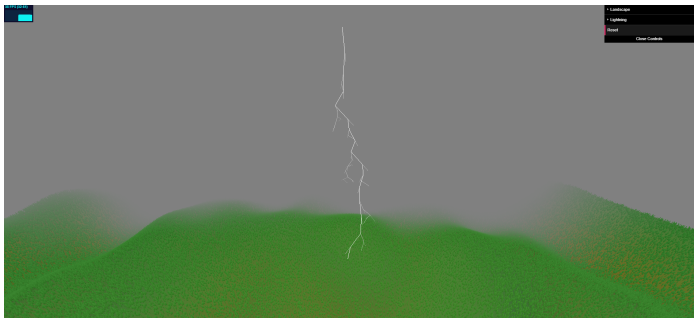

Figure 1: Lightning and fog.

II. II Grass

Grass is added to the plane to create a more pleasant scene. The positioning of the strands of grass are based on Poisson disk sampling. This creates a more natural looking distribution than having a regular distance between the grass. The JavaScript Poisson disk sampling function is created by Matthew Page [2] and is based on an algorithm presented by Robert Bridson [3]. The algorithm makes it possible to place the instances of grass within a minimum distance from one another, the result is tightly packed instances with some random nature to it. See figure 2 for a comparison between random sampling, regular sampling, and Poisson disk distribution.

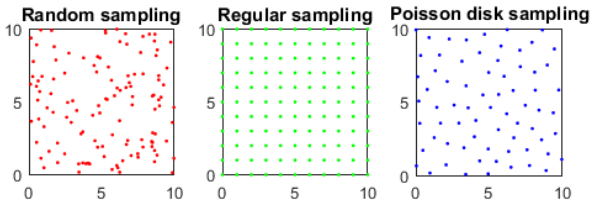

Figure 2: A comparison between different sampling methods.

The random sampling form clusters while the regular sampling is too uniform to seem realistic. The Poisson disk distribution resembles some natural looking pattern, uniformly spread out but with some randomness to it. The Poisson disk distribution is passed as an attribute to the shaders.

A single strand of grass is created by hand by specifying the vertices in two dimensions, the mesh created from those vertices is then used via instancing to create up to multiple thousands of strands of grass. As the meshes of the grass are passed to the vertex shader to be positioned correctly the exact same noise levels are needed to position the grass on top of the plane. Another attribute passed to the shaders is a rotation around the up-axis associated with each instance. The rotations are uniformly random for each instance. The grass is not rotated to be orthogonal to the plane as real grass always grows upright.

Furthermore the grass has a gradient of green across the length to create a more vibrant display, this is also performed in the fragment shader.

II. III Lightning

The thunderstorm is created by having a 3D mesh resembling a lightning bolt strike down every x seconds. The lightning bolts is created with *Three.js*'s LineSegment. To make the lightning bolts seem realistic an algorithm proposed in [4] is used to recursively split and fork the line segments for a set number of iterations.

The algorithm finds a midpoint of a segment and offsets it, and from the new midpoint two new segments are created along with a third segment that is forked from the bend. This is repeated for a set amount of iterations. Each new iteration offsets the midpoint by half of the previous offset value. An example in 2D is presented in figure 3.

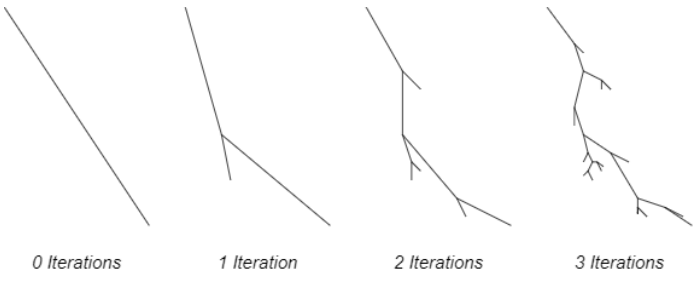

Figure 3: The idea behind the algorithm proposed in [4] in 2D.

The algorithm is not based on any physical properties in actual lightning bolts. Furthermore the proposed algorithm considers a 2D implementation and thus some modifications were needed in order to extend it to 3D.

Moreover the lightning should decrease in intensity, and width, with each fork, to achieve this a opacity value is stored alongside all vertices, for each new iteration the opacity value decreases. An example in 3D with decreasing opacity can be seen in figure 1. Unfortunately there is no support for altering the line width of *gl.LINES* which are used by *Three.js*'s LineSegment. This results in undesirably thin lines.

II. IV Animation

To make things more appealing the plane is animated to make it appear as if the hills are moving. This is achieved by passing a time variable to the noise functions. The plane itself is not moving, the noise is. The grass need to be translated in the same direction as the noise moves to make it move with the hills, this creates a problem when the grass reaches the end of the plane.

To fix this a vertex belonging to a grass instance is moved to the opposite edge of the plane when it has reached the end. To avoid the mesh being stretched across the plane when only a portion of it's vertices are repositioned the vertices are first being moved below the plane, and thus out of sight for the user, slightly before they reach the very end. Figure 4 below illustrates this.

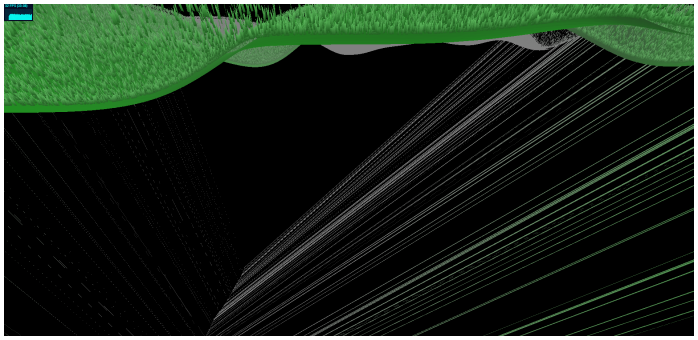

Figure 4: Repositioning of grass vertices.

The lightning strikes are also animated by changing the opacity based on time and y-coordinate (up vector is $(0, 1, 0)$) so that it appears to strike downwards and fade away.

II. V Interactivity

To let users influence the scene some user interface was added. The interface was implemented by using the JavaScript library *Dat.GUI*. The controllable parameters are the velocity of the plane, the frequency of the lightning strikes, and the radius of the Poisson disk sampling (titled "sparsity" in the "Landscape" folder). There is also an option to reset the parameters to their original values. It is worth mentioning that changing the Poisson disk ra-

dius value can cause the scene to temporarily freeze as the new distribution is calculated. The radius might also affect the frame rate because it controls the amount of grass instances being rendered and thus a frame per second counter is visible in the top left corner.

III Results

The entire scene consists of moving hills on a plane with lightning striking down with a certain time interval. The positions of the lightning strikes are random. A user interface lets the user control some parameters such as velocity and frequency as well as how tightly packed the instances of grass should be. When choosing the lowest sparsity possible approximately 121000 instances are being drawn compared to around 300 for the maximum value. See figure 1 for a visual preview. The full result is available *here*.

IV Conclusion

By using Perlin noise and Poisson disk sampling in combination with the algorithm to create lightning bolts a rather interesting scene was created.

Further work might include more adjustable parameters such a the noise levels or fractals used to create the hills, adding a glow to the lightning bolts, and generating new terrain if the user comes close to an edge. Moreover the scene can be further expanded with procedural sounds such as wind and thunder to create a fuller experience. The code is by no means optimized and thus some performance enhancements could be made.

References

[1] S. Gustavson. Webgl noise, 2011. https://github.com/stegu/webgl-noise/blob/master/src/classicnoise2D.glsl.

[2] M. Page. Fast poisson disc sampling in arbitrary dimensions, 2019. http://mjp.co/js/poisson-disc/.

[3] R. Bridson. Fast poisson disk sampling in arbitrary dimensions. https://www.cs.ubc.ca/ rbridson/docs/bridson-siggraph07-poissondisk.pdf.

[4] Drilian's House of Game Development. Lighting bolts, 2009. http://drilian.com/2009/02/25/lightning-bolts/.