Rendering noise-generated terrain with ray marching

Niklas Jonsson* Johan Ju†

Lund University
Sweden

Abstract

Noise is commonly used in the field of computer graphics and can be used for procedural generation among other things. This report will detail a project for terrain generation using simplex noise. It is rendered using raymarching which, given ever more powerful graphic processing units, becomes more and more of a useful tool in rendering.

1 Introduction

Noise is commonly used in the field of computer graphics and can be used for procedural generation among other things. This report will detail a project for terrain generation using simplex noise. It is rendered using real time raytracing in the form of ray marching which, given ever more powerful graphic processing units, becomes more and more of a useful tool in rendering.

The project features real time shadows using ray marching, realistic animated water with refractions and reflections corresponding to the surrounding environment. The OpenGL API is used to accomplish this, using a so called fragment shader, in which most of the computations are done. A framework called bonobo [1] is used as well, namely for handling texture creation from images, user input from mouse and keyboard and providing a GUI in the application.

Terrain generation using noise is interesting because it can create realistic visual effects without needing art assets. Ray marching enables small and compact code, avoiding creating large amounts of geometry in application code and passing it to the shader. The tradeoff is of course performance, where ray marching cannot compete with regular rasterization.

2 Ray marching

Raymarching is done by setting a camera position and then casting rays from the camera position through each pixel on the screen that should be colored. In the case of this project, this is done by rendering a quad that covers the entire viewport in the regular OpenGL pipeline and then drawing using the fragment shader, essentially the scene is drawn upon the quad.

The actual raymarching is implemented as stepping a fixed amount along a line in the direction of the ray until it intersects the object that is being raymarched, e.g. a sphere or terrain. The color of the origin pixel is determined from whatever the ray intersects, e.g. the color of the sphere, sky or terrain. Raymarching, together with some optimisations such as increasing step size as the distance from the camera increases, is described in [Quilez 2002].

2.1 Distance estimation

Instead of doing constant or linearly increasing step, distance estimation can be used. A distance estimation function is a function that, for any point, can give us a lower bound on the distance the ray needs to travel in order to intersect with what is being ray-traced, e.g. noise based terrain or a sphere. For a sphere is trivial, the distance $d(p)$ from a point p to the edge of a sphere, centered in p with a radius r, is:

$$d(p) = |p - c| - r \tag{1}$$

Since the terrain is modeled as a weighted sum of simplex noise as described in section 3, estimating the distance becomes more difficult. Assume the object that is being rendered is horizontal plane. with height y, then the distance to it is:

$$d(p) = |p.y - y| \tag{2}$$

The simplex based terrain can be thought of as perturbations to a horizontal plane so the above equation can be used together with a fudge factor f (that has to be adapted to different scenes and contents) to get:

$$d(p) = f * |p.y - y|, 0 < f \leq 1 \tag{3}$$

which can be be used to increase the step size while maintaining visual quality. It can be improved by backtracking the last step and then performing constant step raymarching in order to increase the accuracy of the intersection.

2.2 Implemented optimizations

The terrain have a maximal height that can be used to optimize the ray-marching. One optimization that is used is to remove all rays when the camera is over the maximum height and the ray direction have a positive y component. This scenario occurs when looking at the horizon with the upper part of the screen showing the sky.

Another optimization is that when the camera is above the max height, the ray can step from the camera position directly to the maximum allowed height of the terrain (as there cannot be terrain over that height) and start to march from there. These optimizations removes around half of the rays in the common scenarios.

3 Terrain Generation

The terrain is generated using a weighted average of simplex noise octaves. Simplex noise was originally devised by Ken Perlin [Perlin 2002] as an improvement to his famous Perlin noise [Perlin 1985]. Stefan Gustavson has created a introduction to Simplex noise and how it is used that is easy to understand [Gustavson 2005].

Most of the noise generation is done in the fragment shader, only a permutation matrix is created in the application and fed to the fragment shader as a texture in order to ease the computational load. The reason for this that the noise is calculated for each ray step. How ray marching is done is further described in section 2 but suffice to say that the noise function is executed a considerable amount of times. We noticed the effects of this when optimizing noise calculations, but more on this in section 7.

The terrain height generation formula is on the form:

$$h(x, z) = \frac{\sum_i \frac{1}{f_i} noise(x * f_i, z * f_i)}{\sum_i \frac{1}{f_i}} \tag{4}$$

*dat11njo@student.lu.se
†elt12jju@student.lu.se

[1] https:/github.com/LUGGPublic/CG_Labs

Where f_i is the i:th frequency we which is include, and each f_i is increasingly larger. Usually frequencies are chosen from the set $\{1, 2, 4, 8, 16 \dots\}$, e.g. 2^k, where k is an integer. This summation of different frequencies is sometimes called different octaves of noise. The lowest frequency gives the largest effects since it has the largest factor, and creates the mountains and valleys type of appearance. Each subsequent frequency adds finer detail but with a lower impact, adding realistic variation on the mountain faces [Quilez 2002][Patel 2016].

3.1 Terrain coloring

For terrain coloring, a simple rock texture was used as a base for the mountain appearance. At high elevations, is is gradually blended into a bright white color as to create the appearance of snowy peaks, which can be seen in fig. 1 in section 7. Close to water level, the rock texture was blended with a sand-like color to create the appearance of a beach or sand at the base of the mountain, close to the water. This sand color was also used under the water to make it lighter in color and more true to reality, as can be seen in the appendix fig. 2.

4 Shadows and lightning

Shadows are calculated by casting a secular ray from the intersection point of the primary ray towards the light source. This is a naive implementation that is costly but works well in this limited project. If it intersects something on the path to the light source, the origin is given a darker shade of the same color. The shading of the terrain is a regular phong diffuse shading, where the normal is calculated by taking the cross product of the numerical tangent and binormal.

The light source is visualised by a globe of light, situated some distance above the ground. Assume the vector between the light source position and camera position is called v. For each pixel, a ray, u, with the same length as v is sent into the scene. If the distance between the end point of u and the light source is small we draw the light source by coloring those pixels.

5 Water rendering

Water in the world is modeled by having a global water level which means that everything under that height is under water. In the ray casting function there are a condition that stops the ray at the water level.

From the water surface a reflection and a refraction ray is casted. The reflection ray is just a regular ray that can reflect the terrain or the light source. The refraction rays direction is approximated by subtracting the normalized incoming ray with the normalized normal of the water. The color of the water is calculated by subtracting light from the rays depending on the dept. This makes shallow water transparent and deep waters dark.

There are also a bump map of small waves on the water to create a more realistic appearance.

6 Results

Some screenshots can be fund in 7 and the average render time is 110 ms on a GTX 1080, with a resolution of 800x800 pixels.

7 Discussion

Unfortunately it was not possible to use the distance estimation optimisation without creating unwanted artefacts in water appearance and effects. It might have been possible with more time but it could have been the nature of the water rendering techniques used that made this content unsuitable for this optimisation.

A drawback of the distance estimation optimization is that it is mostly usable when the camera position is above the point that is going to be rendered. If the camera is very close to the ground (i.e. y = 0) and is looking upwards toward a mountain, then each step distance will be very small. (since it is a function of the height above ground). The same applies to reflection rays which is used for water reflection, which is described in section 2. A reflection ray start at the water surface and travels until it intersects terrain (or not and travels into the sky), and the water is colored appropriately, thus the height above water will be very small in the beginning.

Another optimization that was attempted was creating the noise in the application (using the CPU) and writing it to a texture which was then sampled in the fragment shader. Since the noise function is the same for each fragment shader invocation, it makes sense to compute it only once and then use it for each invocation. This optimization was not usable however, as it introduced unwanted artefacts in the surface of the landscape. Filtering with several samples was tried as well but was not enough to alleviate the issues and in the end the partial optimization of creating the perturbation texture in the application code was used.

Pre-generated noise textures could be extended to even create a noise image outside the program and load it as an image and then make texture out of it, but it would probably introduce the same artifacts. Whether these artefacts where inherent to the optimization technique, i.e. inherent in texture sampling, or whether there is a bug in the noise generation code or something else entirely is not clear. It could possibly have been usable with more work but because of prioritizing other features, it was decided not to pursue this optimization. Possibly, the whole height map (sum of simplex noise octaves) could have been created beforehand and sampled in the shader, whether or not this was possible was not evaluated.

There are a obviously a myriad of other features that could have been implemented. Some that might have been implemented are:

- More terrain types, e.g. ray tracing trees or grass in valleys
- Clouds which can also be made using noise
- Soft shadows, possibly without casting several shadow rays
- Other types of terrains such as plains, savannas or forests

Overall a very fun project that enables students to be more creative than in most other school projects.

References

GUSTAVSON, S. 2005. Simplex noise demystified. May.

PATEL, A. 2016. Making maps with noise functions. Sep.

PERLIN, K. 1985. An image synthesizer. *SIGGRAPH Comput. Graph. 19*, 3 (July), 287–296.

PERLIN, K. 2002. Improving noise. *ACM Trans. Graph. 21*, 3 (July), 681–682.

QUILEZ, I. 2002. Terrain raymarching.

A Screenshots

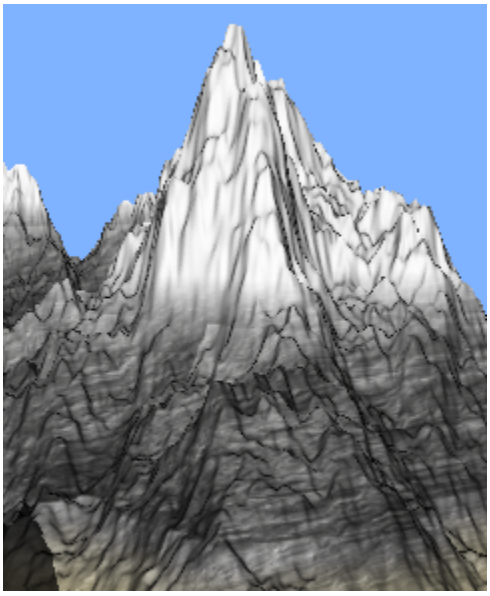

Figure 1: Single mountain

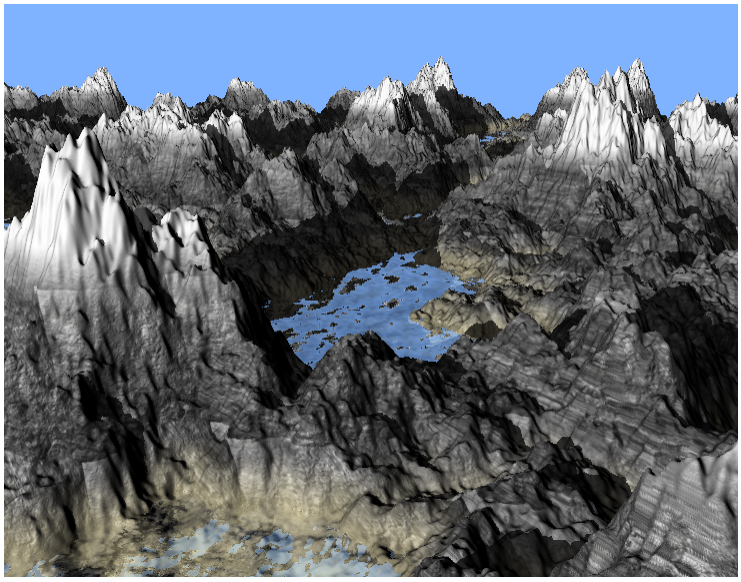

Figure 2: Mountain view