Procedurally generating a planet using noise functions

Adam Söderström
TNM084 - Procedural images
Linköpings Universitet
Linköping, Sweden
adaso578@student.liu.se

***Abstract*— This paper discusses the theory and implementation of a method to procedurally generate an object. The model which is generated by this application is a 3D-planet, which is rendered in the browser using WebGL and the Three.js javascript library.**

I. INTRODUCTION

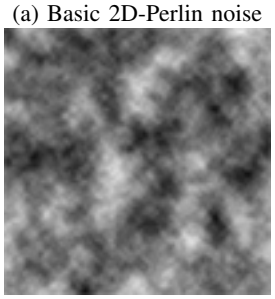
Computer graphics is a continuously developing subject. As the processing power of computer grows, the demand for creating more and more realistic models and 3D-sceneries for games and movies are increasing. To model terrain like ocean, mountains or atmosphere, and to make them appear random is not a trivial task. One way of achieving this is to make them procedurally generated using various noise functions. WebGL is a 3D rendering library which runs in a web-browser. It runs on the GPU and thus allows for complex modeling while still running a high and stable framerate. This paper will discuss one method of generating procedural objects by using noise functions in the shaders.

II. THEORY & IMPLEMENTATION

Perlin noise is a continous noise function which can be implemented in 1-,2- or 3D [1]. It was developed by Ken Perlin in 1983 with the purpose of creating more generic textures and models, and to make them appear more random and less ”machine-like”. A simple example of a basic 2D-perlin noise can be seen in Fig. 1a. To make this more interesting, this noise can be combined with another noise with higher frequency to achieve something like the pattern seen in Fig. 1b which might resemble smoke or fire.

There are much more uses for perlin noise than this simple example. If e.g the noise is thresholded to set all pixels above a certain noise value as one and all others as zero we get a speckled pattern. Only the creativity is the limit.

(a) Basic 2D-Perlin noise



(b) Combined Perlin noise

Fig. 1: 2D-Perlin noise

A. Noise implementation

The implementation of the Perlin noise, along with other noise functions used in this application, is implemented in the shaders using S. Gustavssons GLSL implementations. [2].

Three.js [3] is an lightweight, easy to use, 3D javascript library which can be configured to use the WebGL renderer. The application was implemented using this library to make the creation of lights, geometries and other object easy. The planet consists of three spheres, a surface sphere, an ocean sphere and an atmosphere sphere. Each of these spheres were then assigned their own materials with their own shaders.

By passing the various noise functions found in [2] into the shaders, the noise function can be calculated for each vertex position in the vertex shader. The benefit of defining the noise variable in the vertex shader and not in the fragment shader is that it then later can be passed into the fragment shader as a varying variable. This allows the fragment shader to have knowledge of the displacement made in the vertex shader, and use this information to define the color of the surface.

The landmass e.g was made using this method. Perlin noise was calculated and each surface was displaced in the

normal direction creating the mountains as seen in Fig. 2.

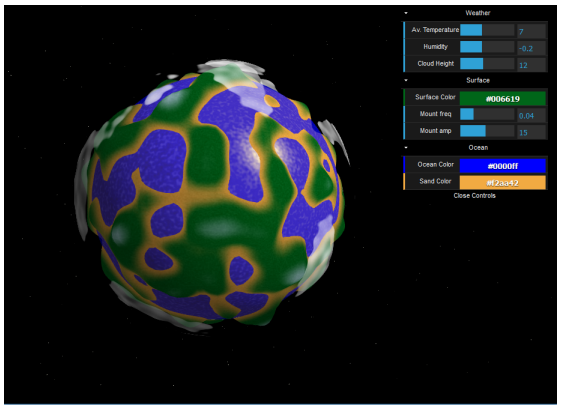


Fig. 2: The final application

As seen in Fig. 2 the grass and snow areas on the mountains of the planet is dependent of the altitude. Peaks will have snow a snow color, and shores will have a sand color. Since the noise is calculated in the vertex shader and passed to the fragment shader, all information needed to achieving this is already obtained. To achieve the gradient between e.g shore and grass is simple to gradient these two colors depending on the altitude of the mountain/amplitude of the noise passed from the vertex shader. In OpenGL Shading Language (GLSL), this is done using the smoothstep() function, see Listing 1

Listing 1: Planet fragment smoothstep

```
1/****/
2float shoreLineTop = mountAmp/3.0+avTemp-7.0;
3/****/
4
5vec3 finalColor=
6 mix(sandColor, surfaceColor,     smoothstep(0.0,
      shoreLineTop, noise));     // Sandy shores
7}
```

B. Controls

To give the user a bit more interaction with the application, environment and atmosphere controls were added to the application. The graphical user interface (GUI) was implemented with the free JavaScript framework Dat.gui [4]. For a user without knowledge of procedural methods, parameters such as noise density or noise amplitude, these controls isn't very intersting. Thus these were translated to parameters such as; mountain amplitude, mountain frequency, avarage temperature and humidity which are parameters relevant to the planet.

When the user changes the parameters by dragging the sliders in the GUI, the associated parameter, or parameters, are instantly updated which leads to instant changes on the planet. The variables are passed as uniform variables to the shaders. All of the different shaders have different uniform variables, however some variables, such as the light position, avarage temperature and the ocean level are needed in multiple shaders. Thus these are places in a shared uniforms array, which is then passed along with the shader specific uniforms to all of the shaders.

The controls used in the application can be seen in 3

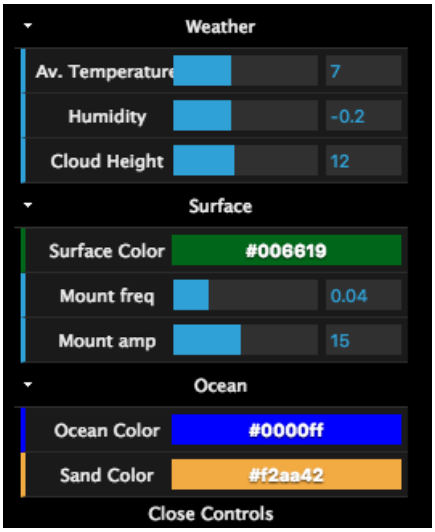


Fig. 3: Application controls

Some controls control multiple parameters. If the planet e.g. has a warm temperature, realistically the sea-levels will sink and there will be less flora on the surface of the planet. If however the temperature were low, the size of the icecaps would increase, and the snow-covered surface would be larger. If the temperature was low enough, the entire planet would be covered in ice and snow. Thus, variables such as: sea level, snow altitude and ice cap size all have to be dependent on the temperature variable. This is achieved by doing different actions in the shader depending on the temperature. If the temperature is high enough the sea-level will lower. This is handled by the ocean vertex shader and the code for this can be seen in Listings 2.

Listing 2: Ocean vertex shader

```
1uniform float time;
2uniform float avTemp;
3
4void main() {
5
6 vNormal = normal;
7
8 float oceanDisplacement = avTemp-7.0;
9 oceanDisplacement = max(0.0, oceanDisplacement);
10
11 pos = position-normal*0.4*oceanDisplacement;
12    gl_Position = projectionMatrix *
          modelViewMatrix * vec4( pos, 1.0 );
13
14}
```

As seen in Listings 2, the variable oceanDisplacement is set to the uniform variable avTemp−7.0 and then set to max(0.0, oceanDisplacement). This results in that if the avarageTemperature is less than 7.0°, the ocean level will start to sink, since each vertex is translated in the normal direction with a negative length that is dependent on the oceanDisplacement variable.

III. Improvements

The final application is working as intended, however some things could need improvement. lightning model used in application is the Phong shading model, however when the vertices are displaced in the vertex shader, the vertex normals isn't recalculated. When the displacement of the vertices has been done, some faces (such as the mountain slopes) don't face the light source anymore, but since the normals haven't been recalculated, these surfaces will be treated in the Phong shading as if they still were. This artifact is however very subtle and not very noticable. One solution to this would be to recalculate the normals in the fragment shader, which however would result in a flat shading which is unwanted.

The application could also need some more user controls in the GUI. The reason why there are so few controls in the application is simply because lack of creativity. Sliders that control multiple parameters which renders multiple changes in the application are more interesting, but also more difficult to think of.

The application only uses Perlin [1] and S. Worleys Cellular noise [5]. Other more advanced noise functions could also be used to make more diversive changes on the planet.

References

[1] K. Perlin , "An Image Synthesizer", SIGGRAPH, vol. 19, no. 3, 1985.

[2] S. Gustavsson, "WebGL-Noise". [Online]. Available: https://github.com/ashima/webgl-noise. [Accessed: Dec- 2016].

[3] Three.js [Online]. Available: https://threejs.org/. [Accessed: Dec-2016].

[4] Dat.gui [Online]. Abailable: https://github.com/dataarts/dat.gui. [Accessed: Dec-2016].

[5] S.Worley, "A Cellular Texture Basis Function", Proceedings of the 23rd annual conference on computer graphics and interactive techniques, 1996.