

Interactive Graphics

Final Project: Procedural Solar System

Michele Cipriano, Ivan Bergonzani

June 22, 2017

1 INTRODUCTION

The theme of the project is about the generation of a solar system using procedural graphics. The aim is to study in depth procedural meshes developing a system that ease the creation of pseudorandomly generated planets.

The program is characterized by having a solar system composed by four planets, a star and a satellite (the Sun, the first four planets of our solar system and the Moon, orbiting the Earth). The meshes are generated using Perlin noise, each planet is composed by multiple chunks which change definition depending on the distance, allowing high performance even with a high number of details. The system supports multiple lights and textures, managed in shaders developed with GLSL. Objects are animated with rotation around themselves and revolution around the parents' object (the Sun for the planets, the Earth for the Moon). A simple flying system has been implemented with the help of THREE.js [1], which simplifies the development of the whole program as well.

The program obtains optimal performances running smoothly both when the camera is far away, where all the objects can be seen altogether, and when the camera is near a planet, where seas, plains and mountains can be seen in detail.

2 PERLIN NOISE

The main character of the project is the Perlin noise, responsible for the generation of the pseudorandom meshes of the planets.

Against regular noise, where each pixel has assigned a random value between 0 and 1, changes occur gradually in Perlin noise, as in natural terrains.

To generate realistic terrains it's necessary to add multiple noise maps together (called octaves), each of which has different amplitude and frequency, determined by the parameters lacunarity and persistance. These two are responsible for the number of small features of the terrain and how much these features change the shape of it.

Noise values are used to determine the height of the terrain for each vertex of the planet. To do this without loosing continuity between adjacent vertices, a 3D Perlin noise function has been implemented passing as parameters the points of a hypothetical sphere. Hence, for a point (x, y, z) on the sphere corresponds a height h . The algorithm for the generation of the heights is Algorithm 1. Note that the number of octaves, the persistance, the lacunarity and the scale are constant value (choosen before the execution of the function). The scale value directly changes the frequency of the function. An external PERLIN function is used to compute the gradient noise.

Algorithm 1 Noise height generator.

```

1 function NOISEHEIGHT( $x, y, z$ )
2   amplitude  $\leftarrow 1$ 
3   frequency  $\leftarrow 1$ 
4    $h \leftarrow 0$ 
5   for oct  $\leftarrow 1$  to octaves do
6     sampleX  $\leftarrow x / scale * frequency$ 
7     sampleY  $\leftarrow y / scale * frequency$ 
8     sampleZ  $\leftarrow z / scale * frequency$ 
9      $h \leftarrow h + PERLIN(sampleX, sampleY, sampleZ) * amplitude$ 
10    amplitude  $\leftarrow amplitude * persistence$ 
11    frequency  $\leftarrow frequency * lacunarity$ 
12  end for
13  return  $h$ 
14 end function

```

3 PLANET GENERATION

The first thing that comes to mind when creating a planet is to use a spherical geometry. Nevertheless the sphere has several drawbacks when dealing with chunks, when changing the level of detail (to preserve performances) and, in particular, when textures are handled. These are, in fact, stretched at the poles, creating a bad graphical effect that ruins the realism of the whole planet.

A solution is to use a cube and “spherify” it by modifying its vertices. In this way the triangles of the geometry will have a similar size and the stretching of the texture will be solved.

While this method solves the problem, a cube geometry is not suitable for chunks and level of details (LOD). In fact, it's necessary to divide the faces of the cube in multiple parts (chunks), and change the resolution of these depending on the distance from the camera. This is exactly what happens in graphic engines when huge terrains are rendered.

A solution is to compose the cube using multiple planes, each plane will behave like a chunk and will change its resolution dynamically using LODs. In particular, each face of the cube is divided in $N * N$ chunks, hence, a planet is composed by $6 * N * N$ chunks.

Vertices of the chunks are initialized using Perlin noise. The height of each vertex w.r.t. the center of the earth is determined by first spherifying the vertex itself and then computing the noise height in that point.

4 CHUNKS

As said before, planets are made of parts called chunks. Each chunk, in the implementation, is actually an element of a LOD object, multiple chunks of a LOD determine how the resolution of the terrain changes with distance. In this case, even if it's defined with a plane geometry, the chunk it's a custom geometry which follows the curvature of the planet.

Choosing the number of chunks which compose the planet is important both to have high details and to have good performances. In particular, a low number of chunks improves the performances when the camera is far away; this is due to the fact that all the meshes of all the planets in the viewport must be rendered. A high number of chunks increases the number of details, but must not be too high, otherwise the performances will be bad when moving away from the planet. The number of chunks can be decreased by increasing the number of vertices of each chunk. Nevertheless, this number shouldn't be too high otherwise the rendering will be slow when approaching the planet.

One of the problem of creating a terrain with multiple meshes instead of just one is the wrong value assigned to the normals at the border. This is due to the fact that normals are computed considering the adjacent vertices, which are not available for the vertices at the border of the mesh.

A simple solution is to define an extended chunk which overlaps the chunk that will be used for the planet, but contains a new, bigger border. In this way the value of the normals of the extended chunk, with the exception of its border, can be copied back to the original chunk, which will then have correct normals also at the border. This eliminates the difference of colors at borders of the chunks, making the planet look like a single piece.

5 LOD

Dividing the planet in multiple pieces is not enough to obtain good performances, this is why level of details (LOD) are used when building terrains.

The idea is simple, instead of composing the planet with multiple chunks, the planet is composed by multiple LOD objects that contain the same chunk with different amounts of vertices. Thus, the same chunk is “computed more times” with different resolutions. Then, at rendering time, each LOD object is updated depending on the position of the camera, in this way it's possible to gradually change the amount of vertices of the mesh while approaching the planet (Figure 5.1).

Choosing the right amount of LODs' chunks its important to have good performances while being far away from a planet and approaching it. In particular, this allows the GPU to handle

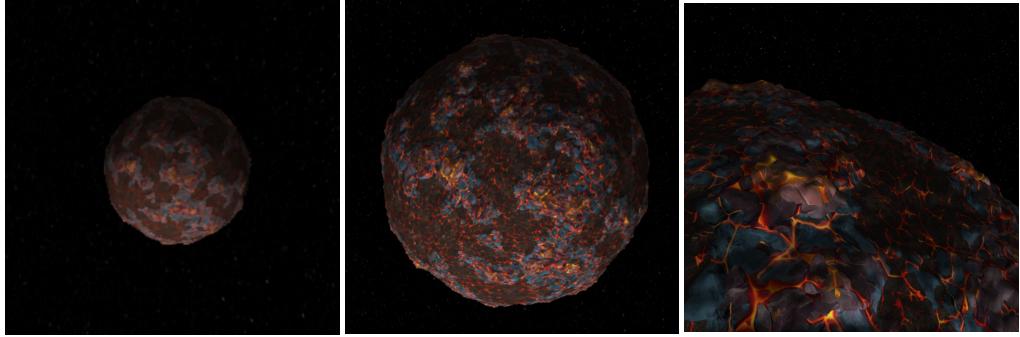


Figure 5.1: Level of detail increases while approaching Venus.

a slower number of vertices for the planets distant from the camera, making it possible to have more resources for the closer meshes, that needs to be more detailed.

6 LIGHTS

THREE.js simplifies a lot the management of the lights even when a custom shader is used, like in the case of this program. Here two point lights are used in the scene, one for the Sun and one for the Moon (Figure 6.1). Lights are directly added as a child of the Planet in the hierarchical structure, making it easy to manage their movement. Since the custom shader increases the flexibility in the development, the parameter “distance” of each light has been used to determine for how much distance the lights affects the objects, instead of being used to computing the attenuation value.

The shader supports also the light emitted by the planet, used to make the Sun, the Moon and Venus brighter.

7 FLY CONTROL

User interaction has been developed with a tool of THREE.js called FlyControl, which has been used for the creation of a simple flying system that allows to move around the solar system.

FlyControl manages the position and the orientation of the camera taking movements from the keyboard and the mouse.

8 ANIMATION

The solar system is composed by the Sun, positioned at the center of the scene, four planets orbiting around the Sun which have similar characteristics to Mercury, Venus, the Earth and Mars, and the Moon, which is orbiting around the Earth. Moreover all these objects are rotating on themselves (Figure 8.1).

IMAGE WITH STRUCTURE OF THE SCENE ?

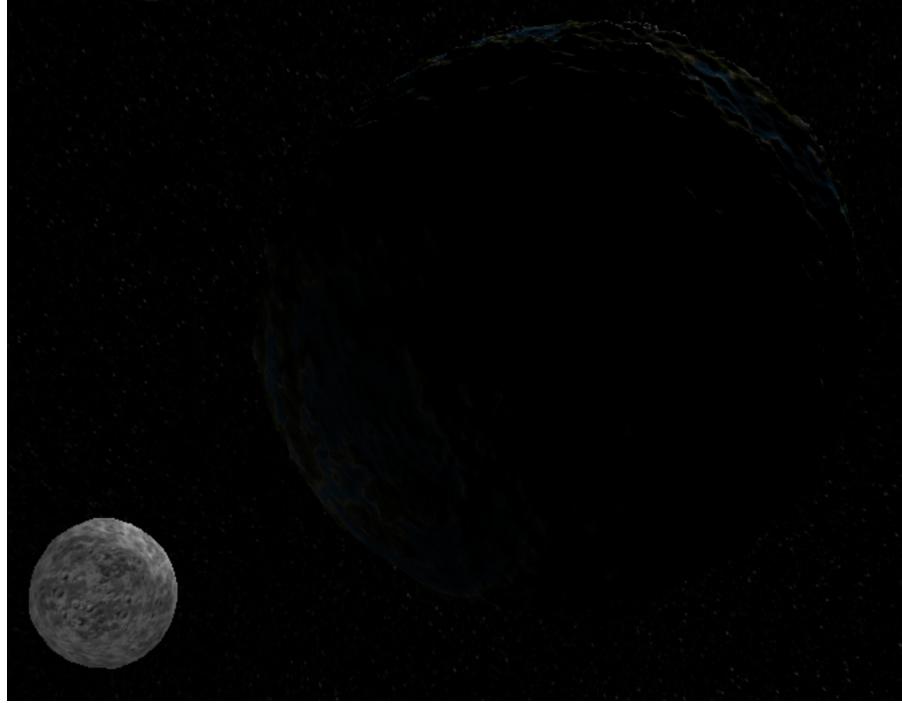


Figure 6.1: The light of the Moon affects the color of the Earth.

The implementation of the orbital revolution is quite straightforward, it's necessary in fact, to slightly change the position of the planets following a circular path. Since this movement must affects also the children in the scene, nothing else has to be done.

On the other hand, the rotation has to be managed in a different way, otherwise the children will rotate with the rotation of the parent. To solve this problem, instead of adding the LOD objects directly to the planet, a pivot object is used. This object is added to the planet and is parent of all the LOD objects of the planet. Thus, instead of rotating the planet, only the pivot is rotated, preserving the position of the children.

9 SHADERS AND TEXTURES

As said before, the shaders are made by hand. This design choice is due to the fact that it becomes easier to manage the textures of the terrain.

In fact, the texture changes on the height of the fragment, which was previously determined using Perlin noise. This is necessary to assign, for the example of the Earth, sea textures to low heights, grass and mountain textures, or snow, to higher heights, creating more realistic planets and differentiating seas by plains and mountains.

Moreover, a texture for the “background” is used, mapping a starfield on a spherified cube, instead of a sphere, to solve, again, stretching of the image at the poles.



Figure 8.1: The planets are rotating around the Sun while the Moon is rotating around the Earth.

10 HOW TO USE

Before launching the HTML file on the browser, it's necessary to execute a server to load external files directly from the Javascript files. This can be easily done with Python: `python3 -m http.server`.

The server will be created on port 8000, if available. It's then possible to load the HTML file from `http://localhost:8000/main.html`.

The program has as inputs the inputs managed by the fly controller, which are:

- W, A, S, D, mouse: move
- R, F: up/down
- Q, E: roll
- up, down: pitch
- left, right: yaw

11 CONCLUSION

Procedural graphics in the last years have seen a big adoption in both videogames and movie industry, this project is just an example of what can be done using a simple tool like Perlin noise.

The creation of a complex planet full of details is not straightforward, even by using tools made available by a library like THREE.js. Optimizing the structure of the program, in particular using chunks and level of details is a must to deal with large terrains and with a high number of meshes.

THREE.js simplifies the management of objects like planets and lights, making it easy to manage the scene, moreover, having a large number of tools, like different geometries and controls, speeds up the development of the project, allowing the developers to concentrate more on the details of the program than the structure of it.

REFERENCES

- [1] “three.js.” <https://threejs.org>. A Javascript 3D library build on top of WebGL.