SAPIENZA UNIVERSITY OF ROME

Interactive Graphics

Final Project: Procedural Solar System

Michele Cipriano, Ivan Bergonzani

July 10, 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 performances 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, resposible 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 there is a corresponding 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 values (choosen before the execution of the function). The scale value directly affects 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 ← 1
3      frequency ← 1
4      h ← 0
5      for oct ← 1 to octaves do
6          sampleX ← x / scale * frequency
7          sampleY ← y / scale * frequency
8          sampleZ ← z / scale * frequency
9          h ← h + PERLIN(sampleX, sampleY, sampleZ) * amplitude
10         amplitude ← amplitude * persistance
11         frequency ← 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 dinamically 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 obtaining the same level of details. 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 borders 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 details. 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 both while being far away from a planet and while approaching it. In particular, this allows the

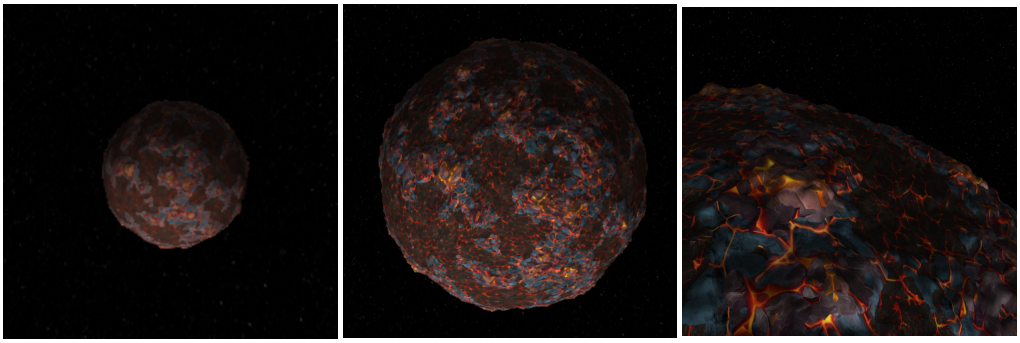

Figure 5.1: Level of detail increases while approaching Venus.

GPU to handle a lower 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 compute 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 around themselves (Figure 8.1).

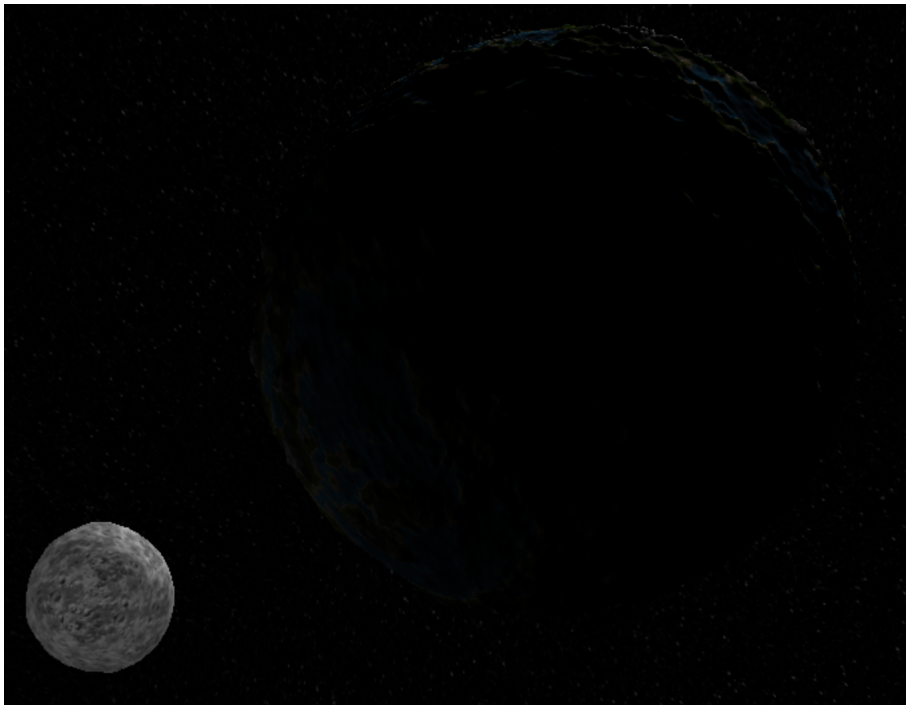

Figure 6.1: The light of the Moon affects the color of the Earth.

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 it's, in turn, parent of all the LOD objects of the planet itself. 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 choise 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, stratching 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 Collisions

11 Planet Labels

12 Earth's Ocean and Atmosphere

The previously introduces procedural planet system deals with the generation of mountains and valleys for a celestial corps; at the top of that textures are applied in order to give to the planets or stars a more suitable appereance. However for a planet like the earth these steps are not enough.
In fact the earth has oceans on the surface and like many other planets presents an atmosphere; these features were implemented using simple sphere meshes. In the first case we use a transparent blue sphere with a different shininess with respect to the ground. A texture mapped sphere was used for the atmosphere: the texture is a semi-transparent image of the Earth's clouds.
The ocean and atmosphere spheres were then added as child to the Earth object; to give a little animation to these entities, they didn't rotate as the parent do: in this way clouds will rotate around the planet and oceans will show some movement due to the imperfection of the sphere geometry.

13 Sun Particle Effects

Until now the presented graphical objects do not show any particular effect, showing a static appeareance apart from the LOD system and the movement animation. For this reason we implemented a custom particles system with the aim of simulating a simple solar activity. The basic idea is to add a high number of moving particles around the surface of the sun; more precisiously every particle move from the surface of the star to the outer space and disappear after having traveled for a fixed distance.
This effects makes sense if thousands of small particles are used; in order to achieve this result, keep the particles as simple as possible is absolutly essential: for this reason each particle of the system is composed by only one triangle.
However using thousands of objects, even if constructed with a single triangle, is still not possible: the 3D managements of such a large number of elements, the enormous number of draw call needed and a greater communication between cpu and gpu will drastically compromise the performances of the webgl application.
The goal is then to minimize these compuations; a solution suitable to the described situation is to incorporate all the particles inside one geometry so to draw the particles system with only one draw call at render time.
It is important to remember that each particle moves independetly from the others so it is not possible to compute the right position from the cpu without modify the mesh geometry and loose in performance. For this reason all the particles are initialized in the same position (origin of the mesh) and then repisitioned in their right position directly from the vertex shader.

The set of informations required by the vertex shader for the particles placement includes movement direction, speed, scale, angular speed for each particle and the starting and ending movement distance from the centre of the entire system.
The information about each particle are concatenated and stored inside attribute buffers while the shared data like the distance range or the frame time are passed with uniforms. In THREE.js the only way to specify attribute buffers is by using a custom THREE.BufferGeometry, rewriting also the vertex, normal and UVs buffers. For the sun particle effects, normals and lights weren't use in the fragment shader considered the massive quantity of ligth emitted by a star: each particles is rendered from a semi-transparent texture[1] colored with a random shades between yellow, orange and red.
In conclusion, using the tecnique just described it was possible to render up to a million indipendent particles (i.e. a million triagle) but in order to optimize the performances this number has been reduced to 100 thousands elements.

14 ASTEROIDS CLOUD

After the four planet of the solar system, in the outer zone of the scene there is a cloud of asteroids. The cloud is composed by several asteroids which move circularly around the sun, each with a different speed and at different distance from the center; to reach a more realistic visual effect, every asteroid has also a different rotation speed, scale factor and geometry. With regard to the last property, there were used about one hundred randomly deformed THREE.SphereGeometry: the deformation includes a random translation from the center of each vertex and a random scaling of the whole mesh on the three axis. During the creation of the cloud one of these deformed geometries is then assigned to each asteroid.
Due to the high number of object desired inside the cloud, the idea behind the implementation of the cloud was similar to the one of the sun particle system: in this case we can see an asteroid as a single particle and the entire cloud is made up with a unique mesh.
Here a particle is clearly not built from a triangle alone but instead using a more complex mesh; in the BufferGeometry it was necessary to build the buffers related to vertices, normal and texture mapping by concatenating the equivalent buffers from the geometry associated with all the asteroids; furthermore, as in the case of the sun particles, addtional buffers were created in order to provide to the shader extra informations for the asteroids' appearence and behaviour.
The informations regarding each asteroid are computed once at initialization time without any further computations excepts for the updates of the uniforms: for example at every animation frame the *time* uniform is updated and then used in the vertex shader for calculating the right position and rotation of the asteroids in according with the informations stored in the attribute buffers.
The advantages of this approach is to show in the scene up to several thousand meshes with a single webgl draw call, decreasing the number of communications with the GPU and lightening the cpu load.
A graphical representation of the introduced custom geometry can be seen in figure 14.1

[1] The texture has a semi-transaprent white drawing that is multiplicated with the random color of the particle.

n DIFFERENT objects with THREE.Geometry

	ASTEROID 1	ASTEROID 2	ASTEROID n
Vertices, normals, UVs	geometry 1	geometry 2	geometry n
Speed uniform	speed 1	speed 2	speed n
Initial position uniform	position 1	position 2	position n
other uniforms			

One UNIQUE object with THREE.BufferGeometry

	ASTEROID 1	ASTEROID 2	_	ASTEROID n
Vertex Buffer	vertices 1	vertices 2	_	vertices n
Normal Buffer	normals 1	normals 2	_	normals n
Uvs Buffer	uvs 1	uvs 2	_	uvs n
Speed Buffer	speed 1	speed 2	_	speed n
Initial Position Buffer	position 1	position2	_	position n
other attributes buffers	_	_	_	_

Figure 14.1: Comparison between n single geometry and one BufferGeometry

Figure 14.2: Two views of the implemented cloud of asteroids

while a sneak peak of the asteroids cloud is show in figure 14.2.

15 Menu

In addition to the use of the FlyControl, the user can interact with the program also through a simple gui menu. This is was implemented using a third party library called *dat.GUI* [**?**] which offers different input controls like buttons, checkbox, combobox, sliders and text inputs; moreover all of these can be grouped in folders. In order to implement a menu with dat.GUI is necessary to give at each desired control an initial value and extra informations about its behaviour: for example, a button needs a event listener, a slider needs a numerical range while a combobox needs a list of items. Note that the control type is not specified but indeed is automatically detected from the given extra information.
In this application the overlay provides input controls useful to manage the time speed or handle the camera position. In fact, through the apposites buttons is possible to rotate or move the camera toward each planets of the solar system. As shown in figure 15.1, every input control is placed in a folder based on his scope: folders in dat.GUI are basically openable and closable menu.

Figure 15.1: Opened and closed dat.GUI menu

16 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

17 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.