PROCEDURAL WORD

Real Time Graphics Programming Project

Playlab fOr inNovation in Games

UNIVERSITY OF MILAN, AA 22/23
STUDENT: CASSISI GIUSEPPE
PROFESSOR: GADIA DAVIDE

1 Summary

2 Project details

Procedural Word is a 3D application developed using OpenGL, where the world terrain is created in a procedurally way and some effects can be applied to the world. The user can change dynamically the shape of terrain by changing some parameters. He can also select the effect to apply to the 3D world. The 3D terrain is populated by trees that will change dynamically with the terrain. He is also composed by four different biomes:

- Lawn biome
- Forest biome
- Snow biome
- Mountain biome

To decide the shape of terrain, the positions of trees and biomes composition I use three height maps. I compute them using the Fractal noise. So basically, my three height maps are:

- Elevation map: The height value decides the height of vertex. This map changes the shape of terrain.
- Biome map: The height value decides the biome. This map changes the biomes composition. Basically, I compare the height value with a biome boundary and assign a colour to a fragment based on that.
- Tree map: The height value is compared with the threshold value. If the height value is greater or equal to the threshold value, I place the tree otherwise I don't place the tree.

The Elevation map and Tree map is used in the Application stage while the Biome map is used the Fragment shader.

2.1 App UI

The UI is very important to change the terrain generation parameters and to select the different effects. To create the UI, I use the ImGUI API. The UI is very easy to use, and she is based on buttons and sliders. The UI also shows the current frame time. I setup and initialize the app UI in the early phase of application setup. After that I create the UI frame. After the frame creation I define the UI structure. I use sliders to modify numeric parameters and buttons to select different effects. After that I render the application UI:

3 Techniques

During the development of this project, I used several techniques. In this chapter I will explain the theoretical overview of them. The techniques that I used in my project are:

- Environment Mapping
- Instancing
- Post processing
- Fractal noise
- Compute shader

3.1 Environment Mapping

The Environment Mapping is a technique that use a texture to describe an Environment. The texture is mapped into a sphere or into a cube (Cube Mapping). The sphere or the cube covers the whole scene. They are very larger than the scene; are seem at an infinite distance. The Environment Mapping can be used for several purposes:

- To add the environment reflection to an object.
- To create the illusion that the object is inside of an environment.

In this case I don’t use the UV coords to sample the texture, but I use the reflection vector. This direction needs to be converted in texture coordinate. How to do this conversion depends how the texture is mapped on the sphere. So, we have several cases:

- Latitude/Longitude format: in this case I look to the latitude and longitude to find the specific texture coordinate.
- Mirror sphere format: is generated doing a photo to a sphere that reflects the environment. We can obtain the texture coordinate doing an angular conversion of my direction.
- Cube format: it is also called Cube Mapping. In this case we have six texture which correspond to the six faces of cube. These six textures correspond to six directions (top, left, right, bottom, ecc…). I used this format in my project, and I will talk more thorough later.

The colour obtained from the sampling of the texture is blended with the colour obtained from the illumination model. The environment map can be an image captured from a real scene or can be a virtual scene.

3.1.1 Cube Mapping

In this technique we have a different convention, indeed, while OpenGL works in a right-handed system, the Cube Map works in a left-handed system.

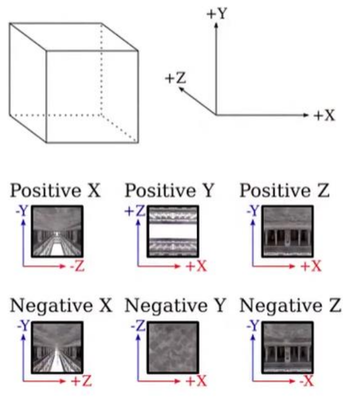

Figure 1 Cube Map coordinate system

Each of these six textures are oriented in a specific direction and represents a specific face of the cube. This format is very important because is fully supported by our graphics hardware.

3.2 Instancing

Instancing is a technique that increases the overall performance by sending the vertex data to the GPU once and drawing many objects with a single draw call. Typically, in a scene you may have many objects that sharing all the same vertex data, the only difference between an object to another is the transformation. So, the only parameter that we must change is the transformation of each object. We have several ways to do this:

- Transformation in vertex shader: we compute the transformation directly in the vertex shader. We can use `gl_InstanceID` to compute the transformation for a specific instance.
- Transformation in uniform: we can compute the transformation in the Application stage and then pass the model matrix as uniform to the Vertex shader. I can retrieve the correct transformation using the `gl_InstanceID`. The problem with uniform is that can't store much data. I can't have a lot of transformation.
- Transformation in VBO: is the best way to have a lot of transformation and not have the generation inside the Vertex shader. Basically, we pass the transformations as mesh attribute. So, we store the transformation data to a new VBO and attach it to the VAO of the mesh. I have chosen this way for my project.

3.3 Post processing

Sometimes inside our games (or 3D applications) we want to apply some effects that modifies the colour and look of our scene. One way to do this is using the Post processing techniques. To use Post processing techniques, we need to create our Framebuffer of the rendered scene, save his Colour buffer (If we want to change the colour) as a 2D texture and finally modify the colour of the texture to create a post processing effect. We modify the colour of a texture in the Fragment shader. The Framebuffer is a combination of three buffer:

- Colour buffer: stores colours values.
- Depth buffer: stores depth information.
- Stencil buffer: stores a value that allows us to discard some fragments.

3.4 Fractal noise

The first noise function that I want to talk is the Perlin noise function. The Perlin noise was invented by Ken Perlin in 1985. In the Perlin noise algorithm we have a grid of points and a point P. So, given an input point $P(x, y)$, we consider the four points of the cell where the point P is. We create four pseudo-random gradient vector (one for each point of the cell). In this way we generate the four pseudo vectors outside the cell. Then, we generate four vectors that are inside the cell and that are directed towards the point P. At the end we have eight vectors. The noise value is created by interpolating all the dot product results between the outer and inner vectors.

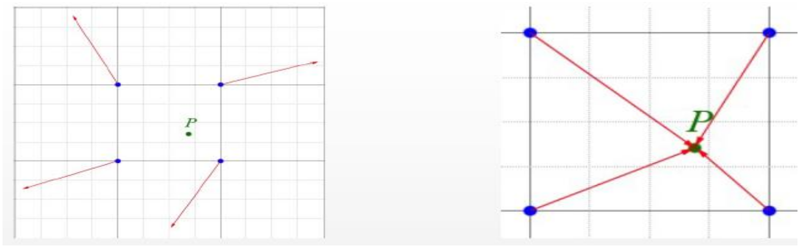

Figure 2 Perlin Noise

The noise generated by Perlin Noise is smooth and not completely random. A more complex version of Perlin Noise is the Simplex Noise. It is multidimensional and was proposed with the aim of avoiding the directional artifacts typical of the classical model. We can create different version of the noise function applying different function to the noise value. One of these versions is the Fractal Noise. The term “Fractal” refers to a thing that is made up of potentially infinite number of itself. We can apply this concept to the Simplex Noise to produce the Fractal Noise. Basically, in the Fractal Noise we generate several Simplex Noise and sum them together. We do this to obtain a terrain more realistic. The Fractal Noise uses some modifiers. In my project I have used only three (octaves, amplitude and frequency). The formula for the Fractal Noise is this:

$$FN = \sum_{i=0}^{octaves} snoise(x * f) * a$$

$$f \mathrel{*}= 2.0 \ \boldsymbol{and} \ a \mathrel{*}= 0.5 \ \boldsymbol{at\ each\ iteration}$$

Where:

- $octaves$: is the number of iterations of Simplex Noise that will sum together.
- a: is the amplitude. Indicate how much extreme is the elevation. In practice this parameter modifies the elevation of our mountains.
- f: is the frequency. Indicate how much detail should we show per unit space. In practice this parameter increases or decreases the number of mountains.

Each Simplex Noise, at each iteration, has the double of the frequency and the half of the amplitude than the previous one.

3.5 Compute Shader

“A Compute Shader is a Shader Stage that is used entirely for computing arbitrary information (OpenGL Wiki contributors 2019)”.

While the other shaders doing graphic computation, the Compute shader can do a general porpuse computation. It uses the parallelism of GPU to accelerate the computations. This new way to use the GPU takes the name of GPGPU (General Porpuse GPU). While the other shaders as Vertex shader and Fragment shader have a well-defined frequency of execution (for example the Vertex shader has a "per vertex" execution). The Compute shader works very differently, the number of executions of Compute shader is defined by the function used to execute the compute operation. Another difference of the Compute shader respect than the other shaders is that the Compute shader haven’t user-defined inputs and no outputs at all. However, a Compute shader can take data as input by loading an image and can send some data as output by writing on this image. The space that compute shaders operate within is abstract. In the Compute shader we have the concept of works groups, that represents the number of executions that the user can do. Each group has several threads and represents the number of instances of the compute shader. A group is a collection of threads running in the same warp. A warp is a collection of cores. The first group number can be intended as an X dimension, the second group number can be intended as Y dimension and the third group number can be intended as Z dimension.

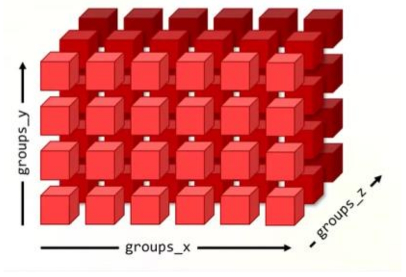

Figure 3 Compute space

4 Techniques implementation

During the development of this project, I implemented the techniques discussed in the previous chapter. In this chapter I will discuss their implementations and why I have chosen to use them into my project.

4.1 Cube mapping implementation

The first choice that I have made is to use a skybox rather than a simple coloured background. It is a simple an aesthetic choice that improve the overall look of the application. To implement a skybox, I have created a class called `SkyBox`. I use this class to setup the buffers of sky cube:

```
glGenVertexArrays(1, &VAO);
	glGenBuffers(1, &VBO);
	glGenBuffers(1, &EBO);
	glBindVertexArray(VAO);
	glBindBuffer(GL_ARRAY_BUFFER, VBO);
	glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices, GL_STATIC_DRAW);
	glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
	glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), &indices,
GL_STATIC_DRAW);
	glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
	glEnableVertexAttribArray(0);
```

Figure 4 Sky cube buffers setup

Then, I load and setup the six textures:

```
glGenTextures(1, &skyBoxTexture);
glBindTexture(GL_TEXTURE_CUBE_MAP, skyBoxTexture);

// Cycles through all the textures and attaches them to the cubemap object
for (unsigned int i = 0; i < 6; i++)
{
	int width, height, nrChannels;
	unsigned char* data = stbi_load(TexturesFacesDirs[i].c_str(), &width,  &height,
	&nrChannels, 0);
	if (data)
	{
	/*This because, unlike most textures in OpenGL, cubemaps are expected to start
in the top
	* left corner, not the bottom left corner */
	stbi_set_flip_vertically_on_load(false);
	/*Notice that I use 'GL_TEXTURE_CUBE_MAP_POSITIVE_X + i'. This represents the
side of the cube that
	*I currently assigning a texture to. I'm adding with 'i' to it in order to cycle
through all the sides */
		glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i,0,GL_RGB,width,height,0,
				GL_RGB,GL_UNSIGNED_BYTE,data);
	/*The cubemaps works in a left-handed system, while OpenGL works in a right-
handed system. In the case of cubemaps the front is positive z-direction  */
		stbi_image_free(data);
	}
	else
	{
	std::cout << "Failed to load texture: " << TexturesFacesDirs[i] << std::endl;
			stbi_image_free(data);
	}
}
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
// These are very important to prevent seams
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_R, GL_CLAMP_TO_EDGE);
```

Figure 5 Textures loading and setup.

We can see in this implementation that I use a *for* statement to load a different texture for a different face of the sky cube. The Cube map texture use the type of GL_TEXTURE_CUBE_MAP because it is a texture composed by six textures. The following code shows the Vertex shader implementation:

```
vec4 SkyBoxVert()
{
   /*I set the incoming local position vector as the outcoming texture coordinate for
(interpolated) use in the fragment shader. The fragment shader then takes these as
input to sample a samplerCube*/
      skyVUVCoord = aPos;
      vec4 pos = proj * view * vec4(aPos, 1.0);
/*when the perspective division is applied its z component translates to w / w = 1.0.
      The perspective division is performed after the vertex shader has run*/
      return pos.xyww;
}
```

Figure 6 Vertex shader implementation

We have a skycube with a position that has a *Z* coordinate equal to 1. So, everything will be in front of skybox (the skybox only be rendered wherever there are no objects visible). To render the skybox, we must set the Depth function to GL_LEQUAL. After the skybox rendering, we must set the Depth function to GL_LESS, that is the default one. Obviously in this case I have used the Cube mapping technique explained in the previous chapter.

4.2 Instancing implementation

The second choice that I have made is to use instancing technique to draw many objects with a single draw call. I have chosen to use instancing technique to significantly improve the performance of my 3D application. I have created an std::vector of glm::mat4 to store the model matrices of my trees. I have passed model matrices data via Vertex Buffer Object (VBO). Following an image that shows the whole process:

```
/*If the 'meshPositions' vector is not empty I define my 'instanceVBO' that contains a
model matrix used for trees positioning*/
      if (!meshPositions.empty())
      {
         glGenBuffers(1, &instanceVBO);
         glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
         glBufferData(GL_ARRAY_BUFFER, meshPositions.size() * sizeof(glm::mat4),
            meshPositions.data(), GL_DYNAMIC_DRAW);
         glVertexAttribPointer(6, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4),
(void*)(0));
         glEnableVertexAttribArray(6);

          glVertexAttribPointer(7, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4),
(void*)(sizeof(glm::vec4)));
         glEnableVertexAttribArray(7);

         glVertexAttribPointer(8, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(2
* sizeof(glm::vec4)));
         glEnableVertexAttribArray(8);

         glVertexAttribPointer(9, 4, GL_FLOAT, GL_FALSE, sizeof(glm::mat4), (void*)(3
* sizeof(glm::vec4)));
            glEnableVertexAttribArray(9);

      /*Here I plug-in these 4 attributes in one instance. So I use these 4 attributes
non per vertex but per instance, one instance at the time. These attributes will be
used for the whole instance*/
         glVertexAttribDivisor(6, 1);
         glVertexAttribDivisor(7, 1);
         glVertexAttribDivisor(8, 1);
         glVertexAttribDivisor(9, 1);
         glBindBuffer(GL_ARRAY_BUFFER, 0);
      }
      glBindVertexArray(0);
```

Figure 7 Buffer setup

As the image shows, I have filled the Vertex Buffer Object with my data (using `glBufferData` function), then I setup the four attributes and finally I have plugged-in these four attributes in one instance using the `glVertexAttribDivisor` function. So, in the Vertex Shader, I will use these four attributes non per vertex but per instance, one instance at the time.

4.3 Framebuffer implementation

The third choice that I have done is to create my own framebuffer to create some post-processing effects. I have created a class called `FrameBuffer` to implement my own framebuffer. First, I have created the framebuffer object and bind it with `GL_FRAMEBUFFER`:

```
//I generate the framebuffer obj
glGenFramebuffers(1, &FBO);
//I bind the framebuffer obj. The GL_FRAMEBUFFER target allows both read and write
operations
glBindFramebuffer(GL_FRAMEBUFFER, FBO);
```

Figure 8 Generation and binding of FBO

The `GL_FRAMEBUFFER` allows both read and write operations on the framebuffer. After that I have created the texture to attach it at the framebuffer:

```
/*Texture generation:
* I create a texture in order to store the render output inside the texture.
* The texture can easily manipulated inside the fragment shader in order to
* create post-processing effects
*/
glGenTextures(1, &frameBufferTexture);
glActiveTexture(textureUnit);
glBindTexture(GL_TEXTURE_2D, frameBufferTexture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, static_cast<GLsizei>(WIDTH),
static_cast<GLsizei>(HEIGHT), 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

//Now I attach the texture to the framebuffer
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D,
frameBufferTexture, 0);
```

Figure 9 Texture creation

Basically, I have created an empty texture to store the render colour (Colour buffer) inside the texture. In this way I can use a shader to manipulate this texture in order to create a Post processing effects. Then, I have created a Render Buffer Object (RBO) and I have attached the Depth buffer and the Stencil buffer to it. The Render buffer has the disadvantage that can't be modified into a shader, but it is faster than a texture. So, I can't modify the content attached to the Render buffer. In my project I don't need to modify the Depth buffer and the Stencil buffer, so is perfectly fine for me. The following code shows the creation and setup of the Render buffer:

```
/*Renderbuffer generation:
 *The renderbuffer can't be read from a shader but is faster than texture. So, because
 *I won't modify the depth buffer and stencil buffer, I store them to the renderbuffer */
glGenRenderbuffers(1, &RBO);
glBindRenderbuffer(GL_RENDERBUFFER, RBO);
//I configure the renderbuffer storage to store both depth buffer and stencil buffer
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, static_cast<GLsizei>(WIDTH),
static_cast<GLsizei>(HEIGHT));
//Now I attach the stencil buffer and depth buffer to the framebuffer
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_RENDERBUFFER,
RBO);
```

Figure 10 Creation and setup of Render buffer

To show the post-processing effects I need to render a rectangle mesh, where I "attach" the texture previously created. So, I have to setup his buffers and attributes as usual.

4.4 Fractal Noise implementation

The fourth choice that I have done is to use the Fractal Noise as noise function to generate terrain. I choose it as noise function because it allowed me to obtain more realistic terrain than the Simplex Noise or classic Perlin Noise. I execute the noise function into a Compute shader. To store the noise value generated by the Fractal Noise I have created a class called `HeightMap`. It provides: an `std::vector<float>` where I store the noise values and a convenient interface to access to these data. I have also created a texture to have output data from the Compute shader and, also, to pass the height map to a shader. The steps to create a height map are the following:

- I Create an empty texture.
- I bind the texture with the image of Compute shader with `glBindImageTexture` function.
- I Execute noise function in the Compute shader.
- I write the noise value in the image previously bound with a texture.
- I get the values to the texture from a vector with `glGetTexImage` function. I store these values into the `std::vector<float>` provided by `HeightMap` class.

4.5 Use of the Compute shader

The last choice that I have done is to use the Compute shader to compute the noise values and fill an image with them; in other words, to create the height map. As I said the previous chapter, the Compute shader is used to speed up general porpuse computation, thanks to the high parallelism provided by the GPU. So, I have decided to use the Compute shader to compute the height map in a fast way. The following code shows the noise computation and the filling of the image:

```
/*I pick the current pixel coordinate. gl_GlobalInvocationID indicate the invocation
ID in all works group*/
ivec2 pixelCoord = ivec2(gl_GlobalInvocationID.xy);
//Range values between [0,1]
float U = float(pixelCoord.x)/(gl_NumWorkGroups.x);
float V = float(pixelCoord.y)/(gl_NumWorkGroups.y);
//I compute the height value using fractal noise
float height = fractalNoise(vec2(U,V), seed);
vec4 valueToStore = vec4( height, 0.0, 0.0, 0.0 );
/* store new value in image where:
--heightMap: Is the image2D I will write the data
--pixelCoord: Is the integer pixel cordinate where I will write the new height value*/
imageStore( heightMap, pixelCoord, valueToStore);
```

Figure 11 Noise computation and the filling of the image

5 Implementation of key points

In this chapter I will discuss the overall implementation of key points of my project. The key points are:

- World creation and updating
- Tree positioning and updating
- Visual effects

5.1 World creation and updating

I have created a terrain as a mesh with a resolution equal to the height maps. I have defined a constant variable called `MAP_RESOLUTION` to represents the resolution of the mesh and height maps. The class `WorldGeneration` is responsible of world generation and updating. This class provides two `std::vector`:

- `std::vector<TerrainVertex>`: it is used to store the mesh vertices.
- `std::vector<int>`: it is used to store the mesh indices.

I have also defined two struct useful for world creation:

- `TerrainVertex`: represents the attributes of a terrain vertex.
- `TerrainMat`: it is the material of terrain.

First, I create the terrain mesh. The following code shows this process:

```
for (int i = 0; i < MAP_RESOLUTION; i++)
    {
        for (int j = 0; j < MAP_RESOLUTION; j++)
        {
            float heightValue = ElevationMap.At(i, j);
            const auto width = static_cast<float>(MAP_RESOLUTION);
            const auto height = static_cast<float>(MAP_RESOLUTION);
            const auto fi = static_cast<float>(i);
            const auto fj = static_cast<float>(j);
            const auto fMeshResolution = static_cast<float>(MAP_RESOLUTION);
            TerrainVertex vertex;
            float heightOfVertex;
            if(heightValue < 0.0f)
            {
                heightOfVertex = heightValue * (HEIGHT_SCALE / 2.0f);
            }
            else
            {
                heightOfVertex = heightValue * HEIGHT_SCALE;
            }
            vertex.Position = glm::vec3
            (
                -width / 2.0f + width * fi / fMeshResolution //X
                , heightOfVertex //Y
                , -height / 2.0f + height * fj / fMeshResolution //Z
            );
            vertex.UVCoords = glm::vec2
            (
                fi / fMeshResolution , //U
                fj / fMeshResolution //V
            );
            //Initially the normal is set to a zero vector
            vertex.Normal = glm::vec3(0.0f);
            vertices.push_back(vertex);
        }

    }
```

Figure 12 Terrain mesh creation

Notice that if the `heightValue` is less than zero the height of the vertex is equal to: `heightValue * (HEIGHT_SCALE / 2.0f)`, otherwise is equal to: `heightValue * HEIGHT_SCALE`. So, I reduce the height of vertex when the height value, of the height map, is negative. I have done this to avoid too deep valleys. Then, I setup the mesh indices to draw a mesh in an efficient way:

```
int rowOffset = 0;
    for (int i = 0; i < MAP_RESOLUTION - 1; i++)
    {
        for (int j = 0; j < MAP_RESOLUTION - 1; j++)
        {
            //First triangle
            indices.emplace_back(j + rowOffset + MAP_RESOLUTION);
            indices.emplace_back(j + rowOffset);
            indices.emplace_back(j + 1 + rowOffset);


            //Second triangle
            indices.emplace_back(j + 1 + rowOffset);
            indices.emplace_back(j + rowOffset + MAP_RESOLUTION + 1);
            indices.emplace_back(j + rowOffset + MAP_RESOLUTION);

        }
        rowOffset += MAP_RESOLUTION;
    }
```

Figure 13 Mesh indices computation

After that I compute the mesh normals, fundamental for light computation. At last, I setup the VAO and VBO buffers. As I said in the first chapter, the user can change dynamically the shape of terrain, so for this reason I have created the method ReComputeMesh() of the WorldGeneration class, to update the vector of vertices and update the VBO with new vertices data. The following code shows this process:

```
void WorldGeneration::ReComputeMesh()
{
    for (int i = 0; i < MAP_RESOLUTION; i++)
    {
        for (int j = 0; j < MAP_RESOLUTION; j++)
        {
            float heightValue = ElevationMap.At(i, j);
            float heightOfVertex;
            if (heightValue < 0.0f)
            {
                heightOfVertex = heightValue * (HEIGHT_SCALE / 2.0f);
            }
            else
            {
                heightOfVertex = heightValue * HEIGHT_SCALE;
            }
            vertices[i * MAP_RESOLUTION + j].Position.y = heightOfVertex;
        }
    }
    ComputeNormals();
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(TerrainVertex),
vertices.data(), GL_DYNAMIC_DRAW);
}
```

Figure 14 Terrain mesh updating

5.2 Trees positioning and updating

For the trees positioning I use random indices generated at run-time, the vertices vector, and the Tree map:

- Random indices: are used to access to Tree map and vertices vector.
- Tree map: is used to decide if I must place a tree or not.

- Vertices vector: is used to get the tree position (the tree position is equal to the vertex position).

I have created the method `SetupTreePositions` to setup the trees positions and updating their positions. The following code shows this process:

```
void SetupTreePositions(WorldGeneration& terrainData, int numberOfIterations,
HeightMap& NoiseTreeMap, float thresholdTreeV)
{
  int count = 0;
  while (count < numberOfIterations)
  {
    //I pick a random column and random row of my TreeMap
    const int randCol = rand() % MAP_RESOLUTION;
    const int randRow = rand() % MAP_RESOLUTION;
    /*If the noise value is greater or equal of 'thresholdTreeV' I create a new
    *model matrix with a new position and a new scale*/
    if (NoiseTreeMap.At(randRow, randCol) >= thresholdTreeV)
    {
      //I pick the position using the random indices computed before
      const glm::vec3 treePosition = terrainData.vertices[randRow * MAP_RESOLUTION +
randCol].Position;
      glm::mat4 TreeModelModel = glm::mat4(1.0f);
      //I translate my model using 'treePosition'
      TreeModelModel = glm::translate(TreeModelModel, treePosition);
      TreeModelModel = glm::scale(TreeModelModel, glm::vec3(0.03f, 0.017f, 0.03f));
      /*I add the model matrix in the 'treeModels' vector.I will pass this vector
      *in the tree mesh buffer*/
      treeModels.emplace_back(TreeModelModel);
     }
     count++;

  }
}
```

Figure 15 Setup tree positions

Then, I update the `instanceVBO` of tree mesh with the `treeModels` vector.

5.3 Visual effects shader implementation

In my project I have implemented five visual effects:

- Toon shading
- Outline effect
- Grayscale effect
- Pixel effect
- Night vision effect

The first one is implemented in the main Fragment shader. The last four effects are implemented in another Fragment shader and are Post processing effects.

5.3.1 Toon shading

The Toon shading is the unique effect that can be combined with another effect, for instance the user can combined the Toon shading with the Outline effect. As I said before this effect was implemented in the main Fragment shader. To create this effect, I have defined the number of transitions between colour levels and the reciprocal of toon levels:

```
//Number of transitions between color levels.
const int toonLevels = 4;
//Reciprocal of toonLevels
const float toonScaleFactor = 1.0 / toonLevels;
```

Figure 15 Toon shading vars

Then, if the Toon shading is enabled, I replace the cosine wave of Lambertian model with several stairs, where every stair has the size of this reciprocal:

```
if(toonShadingIsEnabled)
{
   lambertian = ceil(lambertian * toonLevels) * toonScaleFactor;
}
```

Figure 16 Toon shading computation

I multiply the `lambertian` factor with the `toonLevels`, this provides a fraction between zero and number of `toonLevels`. I use the `ceil` function to obtain the nearest integer greater or equal to: `lambertian * toonLevels`. Then, I multiply the result with `toonScaleFactor` to get the stairs between 0 and 1.

5.3.2 Outline effect

The Outline effect is a Post processing effect and for this reason her implementation is in a different shader. Her implementation is showed by the following code:

```
vec4 OutlineEffect()
{
  //Here I define an array of offset to sample the texture values around the current
texel coordinate
  vec2 offsets[9] = vec2[]
  (
     vec2(-offsetX,  offsetY), vec2( 0.0f,    offsetY), vec2( offsetX,  offsetY),
     vec2(-offsetX,  0.0f),    vec2( 0.0f,    0.0f),    vec2( offsetX,  0.0f),
     vec2(-offsetX, -offsetY), vec2( 0.0f,   -offsetY), vec2( offsetX, -offsetY)
  );

  /*Here I define a kernerl. A kernel (or convolution matrix) is a small matrix-like
array of values centered on the current pixel that multiplies surrounding pixel values
by its kernel values and adds them all together to form a single value.*/
 float kernel[9] = float[]
 (
   -1, -1,  -1,
   -1,  8.2,-1,
   -1, -1,  -1
 );
 vec3 sampleTex[9];
 //Here I sample the texture values (It is colour) using my offsets
 for(int i = 0; i < 9; i++)
 {
    sampleTex[i] = vec3(texture(frameTexture, vTexCoords.st + offsets[i]));
 }
 vec3 col = vec3(0.0);
 //Here I multiply my texture values with kernel values and sum the results
 for(int i = 0; i < 9; i++)
 {
   col += sampleTex[i] * kernel[i];
 }
 return vec4(col, 1.0);
}
```

Figure 17 Outline effect

In the implementation of the Outline effect, I have defined an array of offsets and a kernel matrix. The array of offsets is useful to sample the texture values around the current texel coordinate. The values of kernel matrix are multiplied with the texture values. The results are summed together, and the result of this sum is the final colour.

5.3.3 Grayscale effect

The implementation of the Grayscale effect is the simplest one. Basically, I do the weighted sum of the three colours, then I use the result to construct the final colour.

5.3.4 Pixel effect

The Pixel effect create a pixelated version of the world. To do this I have computed the 2D dimension of each pixel. I have used this dimension to sample the colour from the lower left corner of each pixel, and I fill the whole pixel with that colour. The implementation of this effect is showed by this image:

```
vec4 PixelEffect()
{
    /*Less is the value of pixel more is the dimension of 'dx' and 'dy' and
consequently the screen will appear in a pixelated way. 'dx' and 'dy' represents the
dimension of each pixel. */
    float dx = 15.0 * (1.0 / pixel);
    float dy = 10.0 * (1.0 / pixel);
    /*Now I sample the color from the lower left corner of each pixel, and I fill that
whole pixel with that color*/
    vec2 Coord = vec2(dx * ceil(floor(vTexCoords.x / dx)), dy *
ceil(floor(vTexCoords.y / dy)));

    return texture(frameTexture, Coord);
}
```

Figure 18 Pixel effect implementation

5.3.5 Night vision effect

The implementation of this effect is very similar to Outline effect implementation, the only difference is that I take only green component of final colour and I set the other two colours to zero.

6 References

[1] Interactive Graphics 13 - Environment Mapping, Cem Yuksel, URL: https://www.youtube.com/watch?v=PeAvKApuAuA&t=252s

[2] Instancing, URL: https://learnopengl.com/Advanced-OpenGL/Instancing

[3] OpenGL Tutorial 21 – Instancing, Victor Gordan, URL: https://www.youtube.com/watch?v=TOPvFvL_GRY&t=70s

[4] Framebuffer, URL: https://learnopengl.com/Advanced-OpenGL/Framebuffers

[5] Fractal Noise | Procedural Generation | Game Development Tutorial, White Box Dev, URL: https://www.youtube.com/watch?v=Z6m7tFztEvw&t=143s

[6] Compute Shader, OpenGL Wiki contributors, URL: http://www.khronos.org/opengl/wiki_opengl/index.php?title=Compute_Shader&oldid=14536