

UNIVERSITÉ DE MONTPELLIER

---

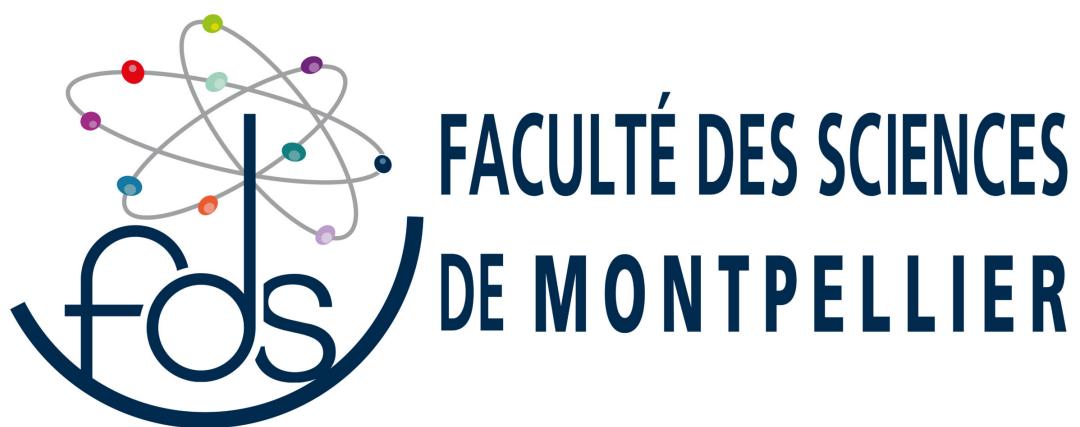
M1 IMAGINE - Moteur de jeu TPn°1/2  
Compte Rendu - Génération d'un plan, manipulation  
de camera, texture et heightmap

---

*Etudiant :*  
Guillaume Bataille

*Encadrant :*  
Noura FARAJ

Le git de moteur de jeu ici : [CLIQUEZ ICI](#)



## 0.1 Contexte et objectif

Lors des différentes cours, nous avons vu les bases simple derrière un moteur de jeu. Notamment l'importance d'une caméra, de réfléchir au fonctionnement des inputs et de la gestion de ce que l'on va rendre à l'écran.

L'objectif de ces TP est donc de :

- Rendre un plan à résolution variable
- Charger et appliquer une texture
- Appliquer une height map
- Manipuler la caméra via GLFW

Contrôles :

Z et S : zoom in/out

Fleches directionnelles : Bouger la camera

O : Active le mode de vue Orbite (Avec I et K pour accélérer/ralentir la rotation)

P : Active le mode de vue Stationnaire

W : Active le mode de rendu wireframe (Line)

X : Active le mode de rendu Fill

+ et - : Augmente la résolution du plane

Voici les textures que je vais utiliser dans ce tp :

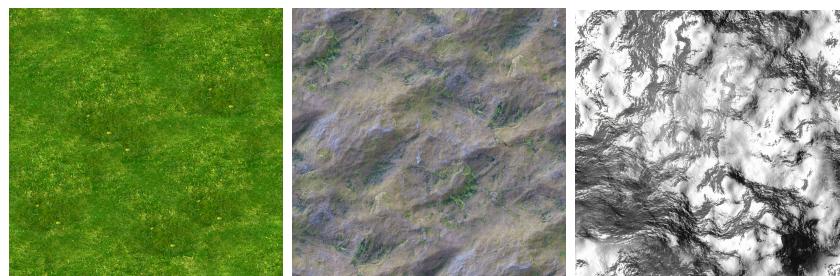


FIGURE 1 – grass.bmp rock.bmp et snowrock.bmp

# Sommaire

0.1	Contexte et objectif	1
<b>1</b>	<b>Construction d'un plan</b>	<b>3</b>
<b>2</b>	<b>Texture</b>	<b>5</b>
2.1	CPU	5
2.2	GPU	6
<b>3</b>	<b>HeightMap</b>	<b>7</b>
3.1	CPU	7
3.2	GPU	7
3.3	Blend de 3 textures	8
3.4	Visuels heightmap	9
<b>4</b>	<b>Gestion des inputs</b>	<b>12</b>
<b>5</b>	<b>Conclusion</b>	<b>13</b>

# 1. Construction d'un plan

Afin de construire notre plan, on part du principe qu'on va avoir un plan sur x,z avec y = 0. L'objectif étant d'avoir à la fin la liste des vertices et la liste des triangles indéxés par les vertices. On a deux paramètre pour la création de notre plan : sa résolution et sa taille.

Pour y arriver on determine a partir de la valeur de résolution le nombre de vertices qu'on va avoir à générer.

Puis on les place tout à taille/resolution les uns des autres : On a maintenant tout nos points. Il suffit donc de parcourir les squares et d'y définir les deux triangles qui le compose.

```
1 void initPlane(std::vector<unsigned short> &indices, std::vector<std::vector<unsigned short>> &triangles,
2 std::vector<glm::vec3> &indexed_vertices, std::vector<glm::vec2> &uv, int resolution, int size)
3 {   int nbVertices = resolution * resolution;
4     float pas = size / (float)resolution;
5     float x = 0, y = 0, z = 0;
6
7     for (int i = 0; i < resolution + 1; i++)
8     {
9         for (int j = 0; j < resolution + 1; j++)
10        {
11            x = j * pas;
12            y = 0;
13            z = i * pas;
14            indexed_vertices.push_back(glm::vec3(x - size/2.f, y, z-size/2.f));
15            // Le "/size.2.f" c'est pour centrer le plane sur 0.,0.,0.
16        }
17    }
18    //Itère sur les carré de vertices pour en déduire les deux triangles qui les compose
19    for (int i = 0; i < resolution; i++) //hauteur
20    {for (int j = 0; j < resolution; j++) //largeur
21    {
22        unsigned short bottomLeft = j + i * (resolution + 1);
23        unsigned short bottomRight = bottomLeft + 1;
24        unsigned short topLeft = bottomLeft + (resolution + 1);
25        unsigned short topRight = topLeft + 1;
26
27        triangles.push_back({bottomLeft, topLeft, bottomRight});
28        triangles.push_back({topRight, topLeft, bottomRight});
29    }
30    // Recupère les id des sommets des triangles from "triangles" in "indices"
31    for (unsigned short i = 0; i < triangles.size(); i++)
32    {
33        indices.push_back(triangles[i][0]);
34        indices.push_back(triangles[i][1]);
35        indices.push_back(triangles[i][2]);
36    }
}
```

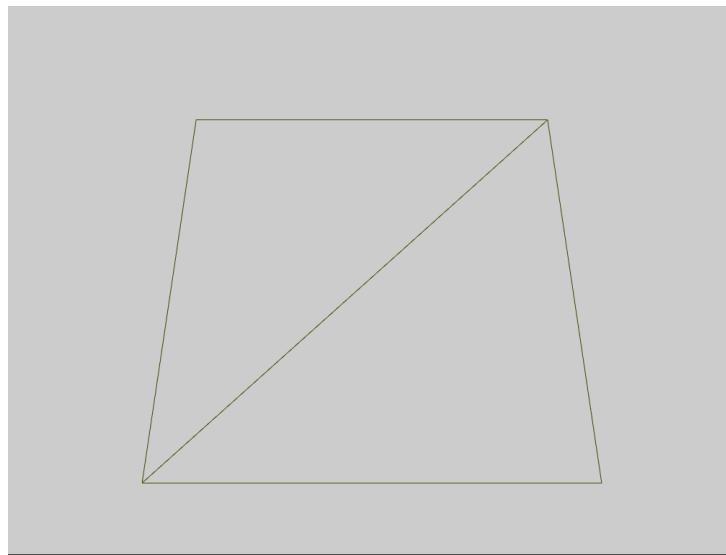


FIGURE 1.1 – Resolution 1

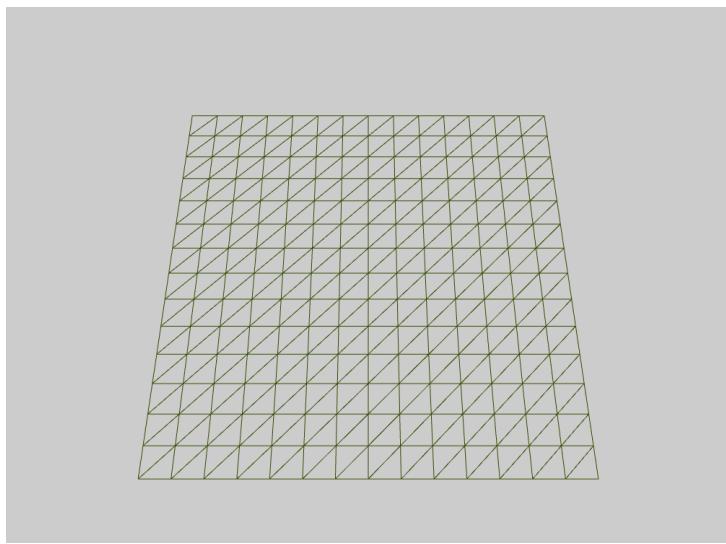


FIGURE 1.2 – Resolution 14

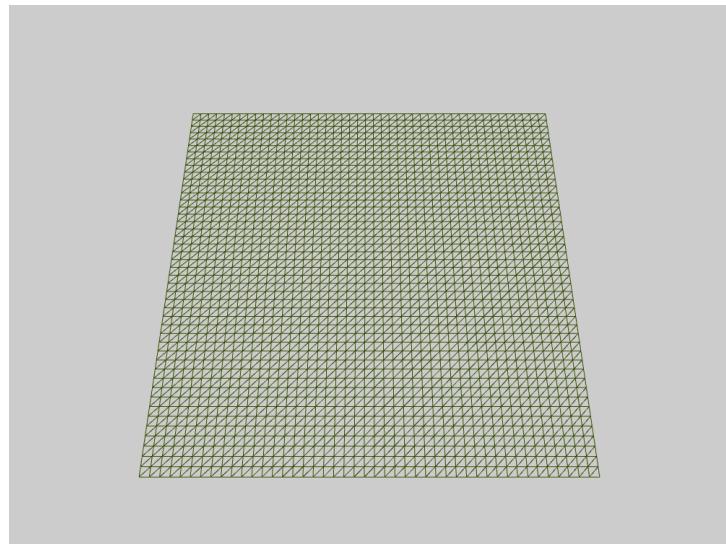


FIGURE 1.3 – Resolution 40

## 2. Texture

### 2.1 CPU

Comme vu le semestre dernier, pour pouvoir appliquer une texture il faut d'abord pouvoir la charger côté CPU afin de l'avoir accessible dans les shaders.

Pour ce faire côté cpu, on a 3 étape à faire :

- Activer un canal de texture (de texture0 a texture32)
- Charger la fonction fournie dans la base de code LoadBMP\_custom("path") qui va s'occuper de charger cette texture dans ce canal.
- Envoyer la texture dans le canal dédié pour qu'elle soit receptionnable dans le shader (attention le nom de la variable est important)

---

```
1 // Exemple pour la texture grass si on utilise le canal 1
2 glActiveTexture(GL_TEXTURE1);
3 loadBMP_custom("../textures/grass.bmp");
4 glUniform1i(glGetUniformLocation(programID, "texture1"), 1);
5 // Le nom texture1 est requis et le dernier argument correspond au canal utilisé
6
```

---

Puis, pour avoir les UV requis pour savoir comment placer cette texture sur nos pixels, on décide de créer une fonction compute UV qui va associer chaque point de notre plan avec un uv.

Avec cette méthode on remplit un buffer d'uv (vec2) qu'on envoie au shaders de la même manière qu'on envoie notre vertexbuffer.

---

```
1 //On bind, charge et envoie le contenu de l'uvbuffer (pour les textures/heightmap) dans le layout 1
2 glBindBuffer(GL_ARRAY_BUFFER, uvbuffer);
3 glBufferData(GL_ARRAY_BUFFER, uv.size() * sizeof(glm::vec2), &uv[0], GL_STATIC_DRAW);
4 glVertexAttribPointer(
5     1,           // attribute (layout)
6     2,           // size
7     GL_FLOAT,   // type
8     GL_FALSE,   // normalized?
9     0,           // stride
10    (void *)0 // array buffer offset
11 );
12 glEnableVertexAttribArray(1); //Activer le layout
13
```

---

## 2.2 GPU

Ce qu'on a fait côté CPU nous permet d'accéder dans les shaders :  
Aux uv via : **layout (location = 1) in vec2 textureCoordinates;**  
et  
A la texture chargé via : **uniform sampler2D texture1;**

Pour les textures uniquement, c'est dans le fragment shader qu'on va les utiliser afin de donner la couleur aux fragments en fonction de l'uv et de la texture associé. C'est simplement réalisé avec : **color = texture(TextureSampler, UV);**

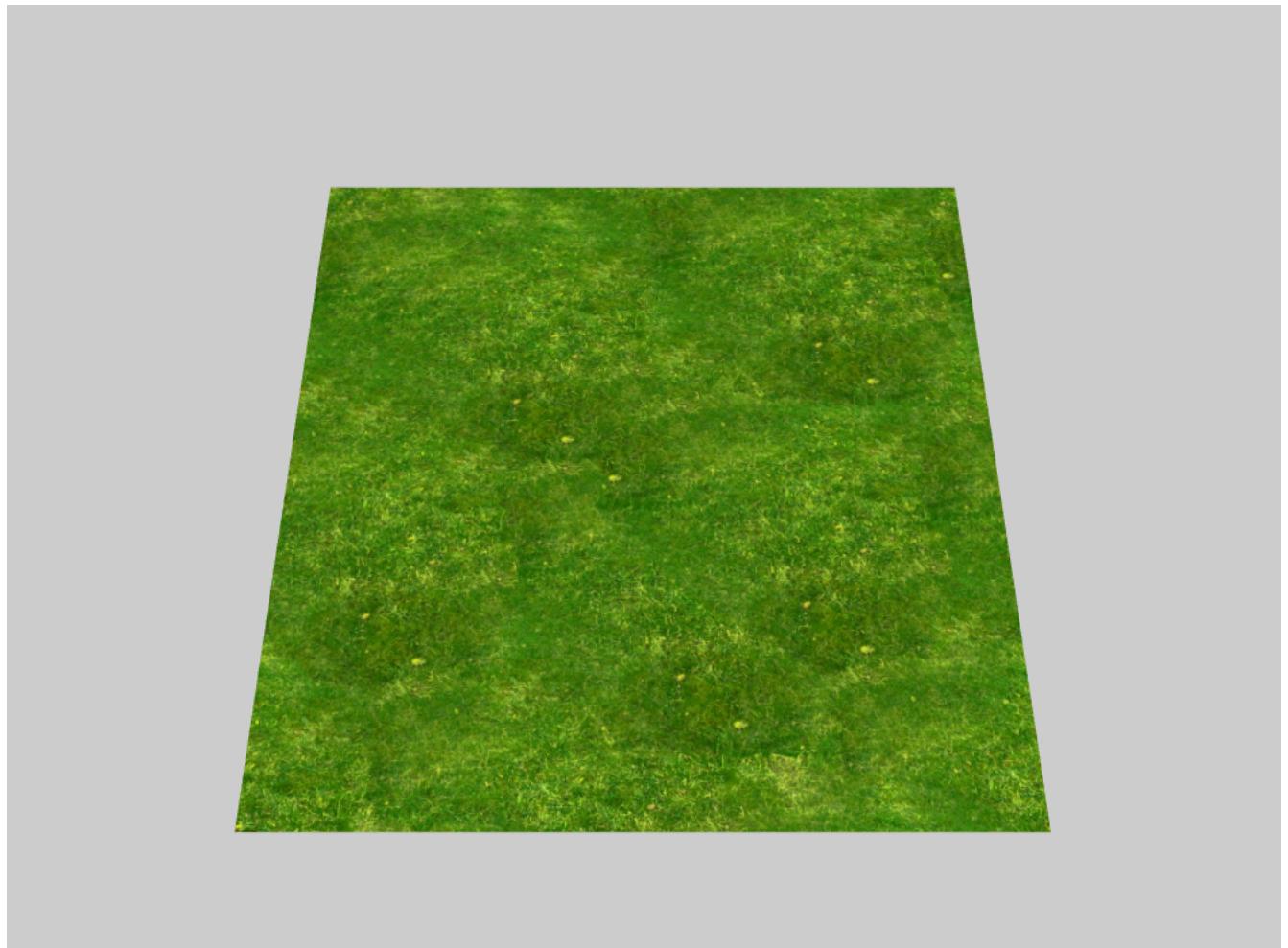


FIGURE 2.1 – Application de la texture grass.bmp sur le plan

Attention ! Les textures .bmp doivent être au format 24bits sinon le LoadBMP\_custom ne va pas savoir comment les lire.  
On peut les obtenir via l'export de gimp > options avancé > 24bits. Si options avancé est grisé il faut la passer au format RVB dans images>mode.

## 3. HeightMap

### 3.1 CPU

Pour le côté CPU, c'est le même processus que pour l'envoi d'une texture et les uv seront les mêmes que celle des textures.

### 3.2 GPU

Cette fois ci, c'est du côté du vertexshader qu'on va appliquer notre heightmap.

---

```
1 //dans le vertexshader
2 vec3 pos = vertices_position_modelspace;
3 float height = texture(texture0,textureCoordinates).r;
4 pos.y += height;
```

---

L'obectif étant d'attribuer "l'intensité lumineuse" de la height map comme valeur y de notre map. Cela aura pour effet de donner de la hauteur au valeur claire de la heightmap et a l'inverse ne pas éléver les valeur sombre de cette ci.

### 3.3 Blend de 3 textures

Enfin, dans le fragment shader, on choisit de blend 3 textures en fonction de la valeur de la hauteur  $y$  pour avoir les valeur basse représentant la texture grass, les valeurs intermédiaire représentant la texture rock et les valeurs haute représentant les textures mountainsnowrock.

---

```
1 #version 330 core
2
3 // Output data
4 out vec4 color;
5 in vec2 TexCoord;
6
7 uniform sampler2D texture0;
8 uniform sampler2D texture1;
9 uniform sampler2D texture2;
10 uniform sampler2D texture3;
11
12 void main(){
13 // Sample the textures at the current texture coordinate
14 vec4 tex1 = texture(texture1, TexCoord);
15 vec4 tex2 = texture(texture2, TexCoord);
16 vec4 tex3 = texture(texture3, TexCoord);
17
18 // Calculate the weights for each texture based on the height value
19 float height = texture(texture0, TexCoord).g;
20
21 float weight1 = smoothstep(0, 0.5, height);
22 float weight2 = smoothstep(0.5, 0.8, height);
23 float weight3 = smoothstep(0.8, 1, height);
24
25 // Blend the textures together based on the weights
26 vec4 blendedColor = mix(tex1, tex2, weight1);
27 blendedColor = mix(blendedColor, tex3, weight2 + weight3);
28
29 // Output the final color
30 color = blendedColor;
31
32 }
```

---

### 3.4 Visuels heightmap

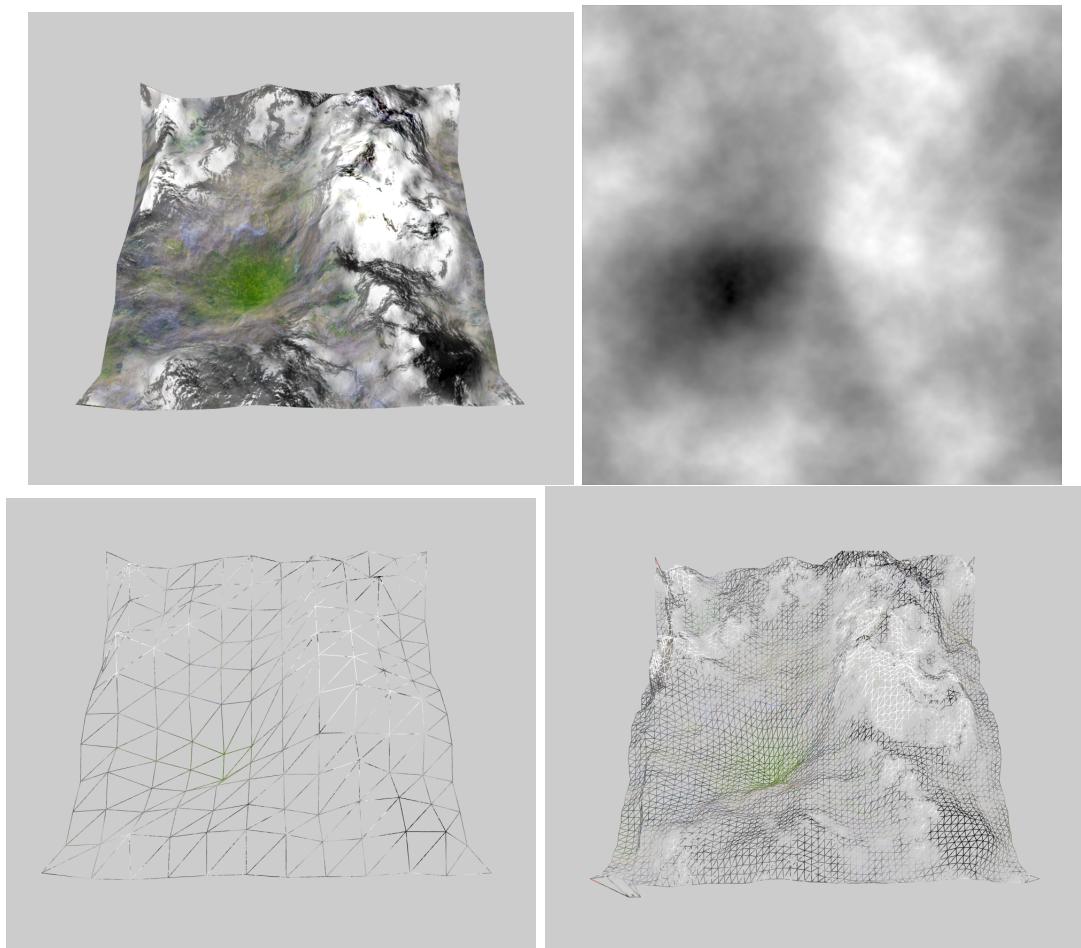


FIGURE 3.1 – Le rendu avec la height map associé et deux version avec des resolutions différentes

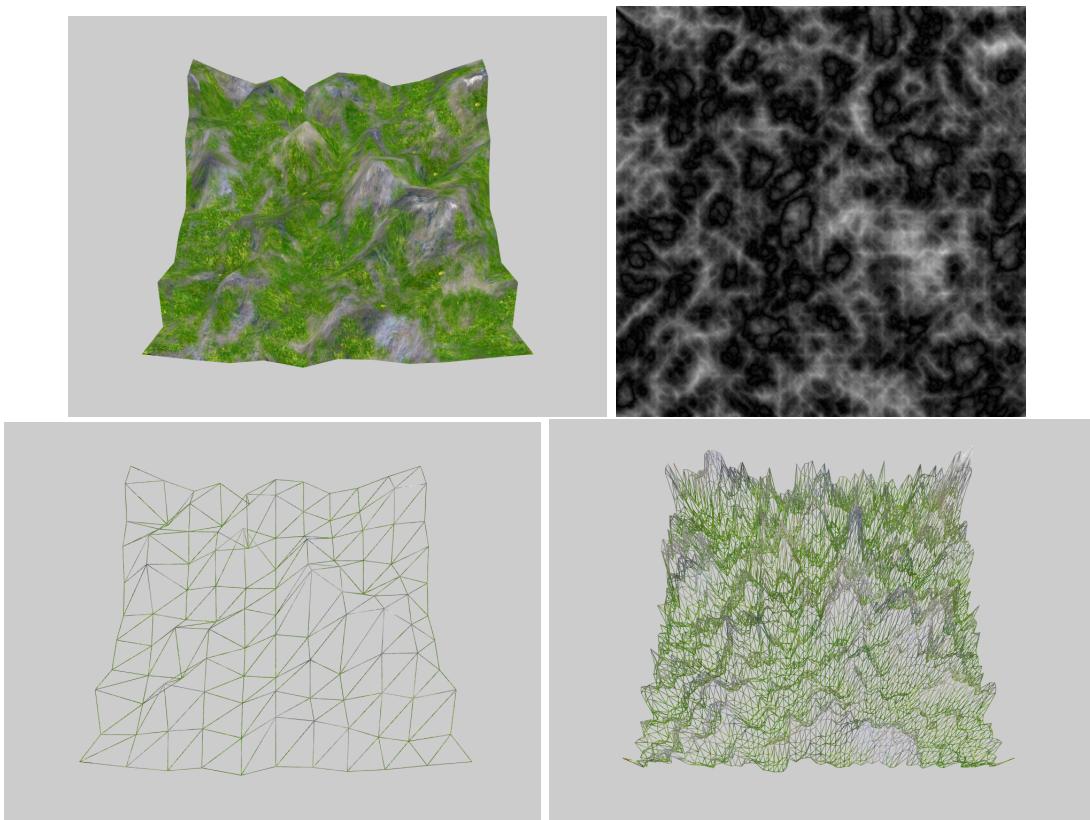
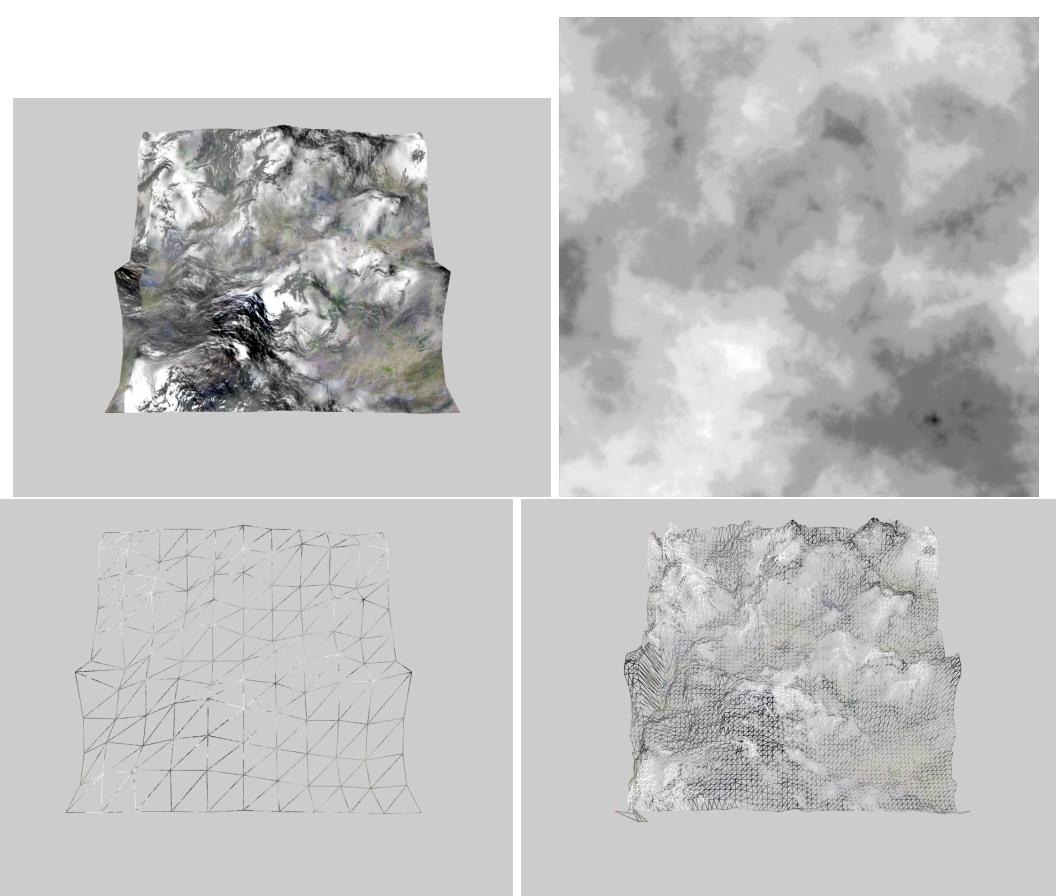


FIGURE 3.2 – Le rendu avec la height map associé et deux version avec des resolutions différentes



## 4. Gestion des inputs

Durant ce tp, on a aussi developper la gestion des inputs via GLFW. On va simplement verifier dans notre boucle de rendu si une touche pressed est detectée. Si c'est le cas on effectue le necessaire : Transformation sur la position de la camera si on veut la déplacer Augmenter ou diminuer la valeur de résolution et d'autres qui ont été préciser dans l'introduction du compte rendu.

## 5. Conclusion

Pour conclure, on a su utiliser nos connaissances des shaders/buffers/OpenGL afin d'afficher dans une scène un plan à résolution variable avec des textures blendés et une normal map, le tout en gérant nos input.

La prochaine étape serait de hierarchisé tout cela et d'introduire la notion de scène.