



# Moteur de jeux

## Compte-rendu TP1 & TP2

Louis Jean  
Master 1 IMAGINE  
Université de Montpellier  
N° étudiant : 21914083

14 février 2024

## Table des matières

1	Introduction	2
2	Création du plan	3
3	Caméra libre	4
4	Caméra orbitale	6
5	Relief sur le plan	8
6	Texture	9
7	Height map	10
8	Conclusion	13

# 1 Introduction

Le but de ces deux premiers travaux pratiques est de réaliser une petite application de rendu 3D en utilisant GLFW/OpenGL. Les objectifs sont les suivants :

- Afficher une scène simple (une caméra, un objet)
- Apprendre à gérer les évènements (clavier, souris)
- Créer une surface et afficher ses triangles
- Contrôler les déplacements de caméra
- Utiliser une carte de hauteur pour mettre à jour le terrain
- Appliquer différentes textures en fonction de la hauteur
- Enrichir les déplacements de caméra

**N.B :** nous utiliserons une base de code fournie par l'enseignant pour ce TP.

## 2 Cr ation du plan

Afin de cr er un plan, j'ai rempli la fonction `set_plane`.

```
1 void set_plane(std::vector<glm::vec3> &indexed_vertices, std
2   ::vector<unsigned short> &indices, std::vector<glm::vec2>
3   &UVs, int nX, int nY) {
4   indexed_vertices.clear();
5   indices.clear();
6   UVs.clear();
7   float deltaX = width / (float)(nX - 1);
8   float deltaY = height / (float)(nY - 1);
9   // Points
10  for(int i = 0; i < nX; i++) {
11    for(int j = 0; j < nY; j++) {
12      float x = i * deltaX - width / 2;
13      float y = j * deltaY - height / 2;
14      float z = 0;
15      indexed_vertices.push_back(glm::vec3(x, y, z));
16    }
17  }
18  // Triangles
19  for(int i = 0; i < nX - 1; i++) {
20    for(int j = 0; j < nY - 1; j++) {
21      unsigned short topLeft = i * nY + j;
22      unsigned short topRight = topLeft + 1;
23      unsigned short bottomLeft = topLeft + nY;
24      unsigned short bottomRight = bottomLeft + 1;
25      indices.push_back(topLeft);
26      indices.push_back(bottomLeft);
27      indices.push_back(bottomRight);
28      indices.push_back(topLeft);
29      indices.push_back(bottomRight);
30      indices.push_back(topRight);
31    }
32  }
33  // UVs
34  for(int i = 0; i < nX; i++) {
35    for(int j = 0; j < nY; j++) {
36      float u = (float)i / (float)(nX - 1);
37      float v = (float)j / (float)(nY - 1);
38      UVs.push_back(glm::vec2(u, v));
39    }
40  }
```

Figure 1: Fonction `set_plane`

*width* et *height* sont des variables globales qui représentent la taille physique du plan, et *nX,nY* contrôlent la résolution du plan. Pour gérer l'incrémentation et la décrémentation en temps réel de *nX,nY*, j'ai écrit un callback **key\_callback**, je ne suis pas passé par **processInput** car cette dernière était appelée trop souvent, résultant en un mauvais contrôle des paramètres. On peut donc augmenter/diminuer *nX,nY* en appuyant respectivement sur P/M.

Après avoir créé tous les buffers nécessaires et les avoir envoyés au shader, voici ce que l'on peut observer.

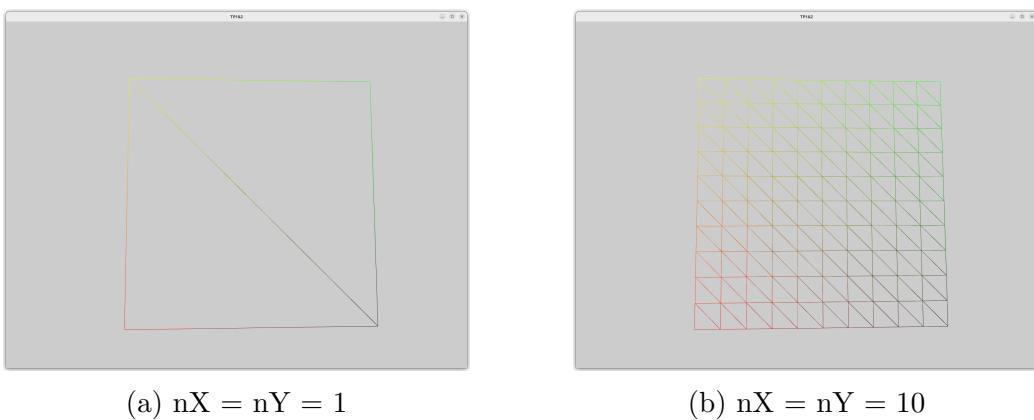


Figure 2: Plan avec différentes valeurs de  $nX,nY$

### 3 Caméra libre

Pour visualiser ma scène, j'ai commencé par implémenter une caméra libre. Pour cela, j'ai commencé par définir les attributs de ma caméra puis j'ai défini les matrices Modèle, Vue et Projection.

```

1 glm::vec3 camera_position = glm::vec3(0,0,0);
2 glm::vec3 camera_target = glm::vec3(height/nX,width/nY,0.0f);
3 glm::vec3 camera_up = glm::vec3(0.0f, 1.0f, 0.0f);

```

Figure 3: Attributs de la caméra

```

1 glm::mat4 model_matrix = glm::mat4(1.0f);
2 glm::mat4 view_matrix = glm::lookAt(camera_position,
3                                     camera_target,camera_up);
4 glm::mat4 projection_matrix = glm::perspective(45.0f,(float)
5                                               SCR_WIDTH/(float)SCR_HEIGHT,0.1f,500.0f);

```

Figure 4: Matrices Modèle, Vue et Projection

Après avoir envoyé les matrices au shader, on peut s'attaquer aux contrôles de la caméra. J'ai écrit cela dans **processInput**.

```

1 // Libre
2 if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS) {
3     camera_position.x += camera_speed; camera_target.x +=
4         camera_speed;}
5 if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS) {
6     camera_position.x -= camera_speed; camera_target.x -=
7         camera_speed;}
8 if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS) {
9     camera_position.y += camera_speed; camera_target.y +=
10        camera_speed;}
11 if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS) {
12     camera_position.y -= camera_speed; camera_target.y -=
13         camera_speed;}
14
15 // Zoom
16 if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) {
17     camera_position.z += camera_speed; camera_target.z +=
18         camera_speed;}
19 if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) {
20     camera_position.z -= camera_speed; camera_target.z -=
21         camera_speed;}

```

Figure 5: Contrôles de la caméra libre

On peut et zoomer/dézoomer avec Z/S et déplacer la caméra libre avec les flèches directionnelles.

## 4 Caméra orbitale

Pour créer une caméra orbitale, c'est-à-dire une caméra qui se déplace virtuellement sur une sphère autour de l'objet considéré, j'ai commencé par déclarer quelques attributs utiles.

```
1 float horizontal_angle = 3.14f;
2 float vertical_angle = 0.0f;
3 float radius = 20.0f;
4 bool orbital = true;
```

Figure 6: Attributs de la caméra orbitale

Ensuite, j'ai mis à jour les angles en fonction des touches pressées, puis la position de la caméra en fonction de ces angles. Remarquez la condition sur l'angle vertical, pour éviter d'avoir un effet de flip désagréable.

```
1 if(orbital) {
2     // Orbitale
3     if (glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS)
4         horizontal_angle -= camera_speed;
5     if (glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS)
6         horizontal_angle += camera_speed;
7     if (glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS)
8         vertical_angle += camera_speed;
9     if (glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS)
10        vertical_angle -= camera_speed;
11    // Zoom
12    if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS) radius
13    -= 10*camera_speed; // 10 fois pour que aller plus vite
14    sans modifier la vitesse de l'orbite
15    if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS) radius
16    += 10*camera_speed;
17    vertical_angle = std::max(-1.57f, std::min(1.57f,
18    vertical_angle));
```

Figure 7: Contrôles de la caméra orbitale

```
1 camera_position.x = camera_target.x + radius * cos(      vertical_angle) * sin(horizontal_angle);  
2 camera_position.y = camera_target.y + radius * sin(      vertical_angle);  
3 camera_position.z = camera_target.z + radius * cos(      vertical_angle) * cos(horizontal_angle);
```

Figure 8: Mise à jour de la caméra

Cet ajout permet d'avoir une caméra bien plus agréable pour naviguer dans l'application. Cette caméra se déplace de la même manière que la caméra libre.

**N.B :** on peut appuyer sur la touche C pour alterner entre caméra orbitale et caméra libre (voir **key\_callback**).

## 5 Relief sur le plan

Pour obtenir du relief sur le plan, je me suis permis d'utiliser la bibliothèque **FastNoiseLite.h** que nous avions déjà utilisée avec M. Hartley dans un TP, pour modifier les coordonnées z du plan.

```
1 float z = std::min(0.0f, noise.GetNoise((float)i*1000,(float)j *100));
```

J'ai pu obtenir un bruit sur l'axe z qui fait ressortir du relief. J'ai fait le choix de clamer la valeur à 0 pour n'avoir aucun creux dans mon plan mais uniquement des pics.

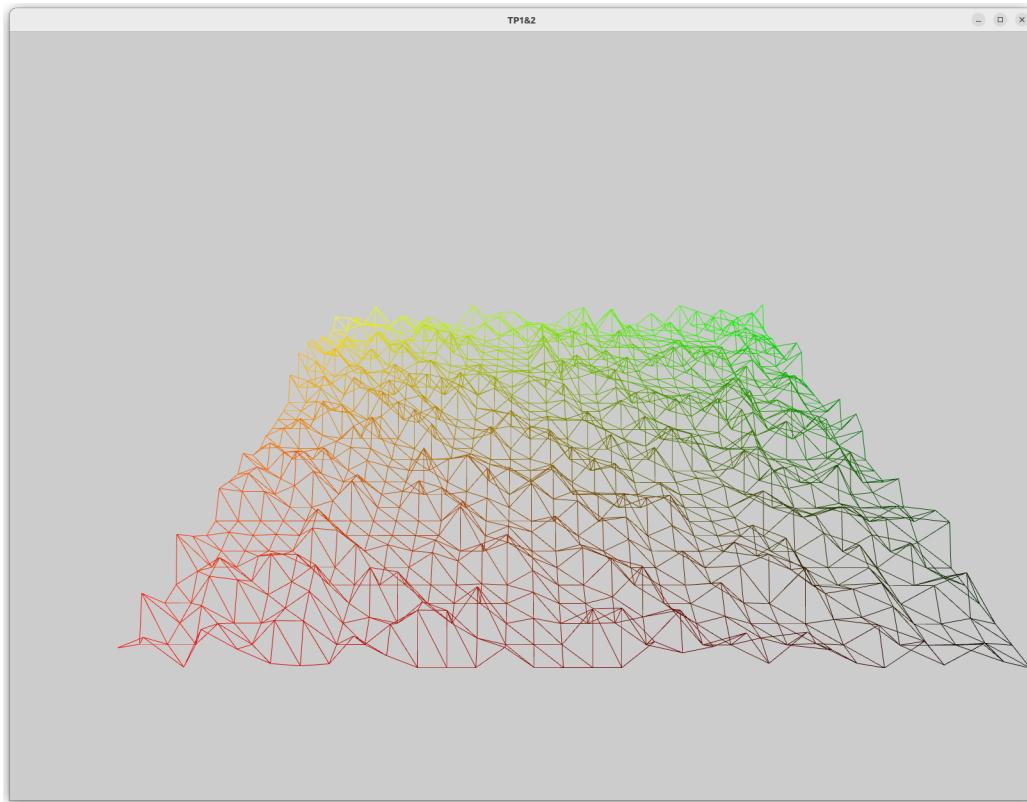


Figure 9: Plan avec relief

## 6 Texture

Pour appliquer une texture, je me suis servi de la classe **Texture** fournie par M. Beugnon lors de ses TP, ainsi que de **stb\_image**. De cette manière, j'ai pu ouvrir une image au format .png et m'en servir comme texture.

```
1 GLuint rock = loadTexture2DFromFilePath("textures/rock.png");
```

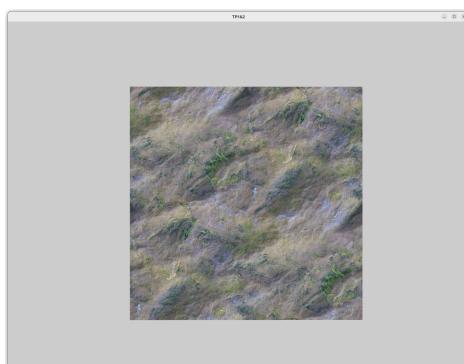
Figure 10: Chargement de la texture

```
1 glActiveTexture(GL_TEXTURE0);
2 glBindTexture(GL_TEXTURE_2D, rock);
3 glUniform1i(glGetUniformLocation(programID, "rockSampler"), 0);
```

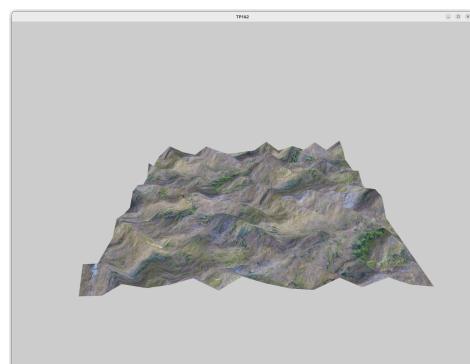
Figure 11: Liaison de la texture

```
1 in vec2 UV;
2 out vec3 color;
3 uniform sampler2D rockSampler;
4 void main() {
5     color = texture(rockSampler, UV).rgb;
6 }
```

Figure 12: Code du fragment shader



(a) Texture de pierre sur le plan



(b) Texture de pierre sur le plan en relief

Figure 13: Plan texturé

## 7 Height map

La height map est une texture qui a pour but de décider de la hauteur des sommets de notre plan. Ici, on oublie donc le bruit pour attribuer la valeur de z. On lit la texture, on calcule la valeur de z dans le vertex shader, et on décide de la texture à appliquer dans le fragment shader. On va donc lire trois textures (herbe, pierre, neige) et une height map.

```
1 #version 330 core
2 layout(location = 0) in vec3 vertices_position_modelspace;
3 layout(location = 1) in vec2 vertexUV;
4 uniform mat4 model_matrix;
5 uniform mat4 view_matrix;
6 uniform mat4 projection_matrix;
7 uniform sampler2D heightmap;
8 uniform float heightmapScale;
9 out vec2 UV;
10 out float height;
11
12 void main() {
13     height = texture(heightmap,vertexUV).r * heightmapScale;
14     vec3 pos = vertices_position_modelspace;
15     pos.z = height;
16     UV = vertexUV;
17     gl_Position = projection_matrix * view_matrix *
18     model_matrix * vec4(pos,1);
19 }
```

Figure 14: Code du vertex shader

```

1 #version 330 core
2 in vec2 UV;
3 in float height;
4 uniform sampler2D grassSampler;
5 uniform sampler2D rockSampler;
6 uniform sampler2D snowrockSampler;
7 uniform float heightmapScale;
8 out vec3 color;
9
10 void main() {
11     float grassHeight = 0.2 * heightmapScale;
12     float rockHeight = 0.6 * heightmapScale;
13     float snowrockHeight = 0.7 * heightmapScale;
14     vec3 grass = texture(grassSampler,UV).rgb;
15     vec3 rock = texture(rockSampler,UV).rgb;
16     vec3 snowrock = texture(snowrockSampler,UV).rgb;
17     if(height <= grassHeight) {
18         color = grass;
19     }
20     else if(height <= rockHeight) {
21         float t = (height - grassHeight) / (rockHeight -
22         grassHeight);
23         color = mix(grass,rock,t);
24     }
25     else if(height <= snowrockHeight) {
26         float t = (height - rockHeight) / (snowrockHeight -
27         rockHeight);
28         color = mix(rock,snowrock,t);
29     }
30     else {
31         color = snowrock;
32     }
33 }
```

Figure 15: Code du fragment shader

J'ai utilisé une interpolation entre deux textures en fonction de la hauteur relative par rapport aux seuils que j'ai défini, afin d'avoir des transitions douces entre les textures. Aussi, j'ai passé une variable **heightmapScale** aux shaders, qui a pour but d'amplifier l'effet de la height map pour obtenir un rendu plus visuel.

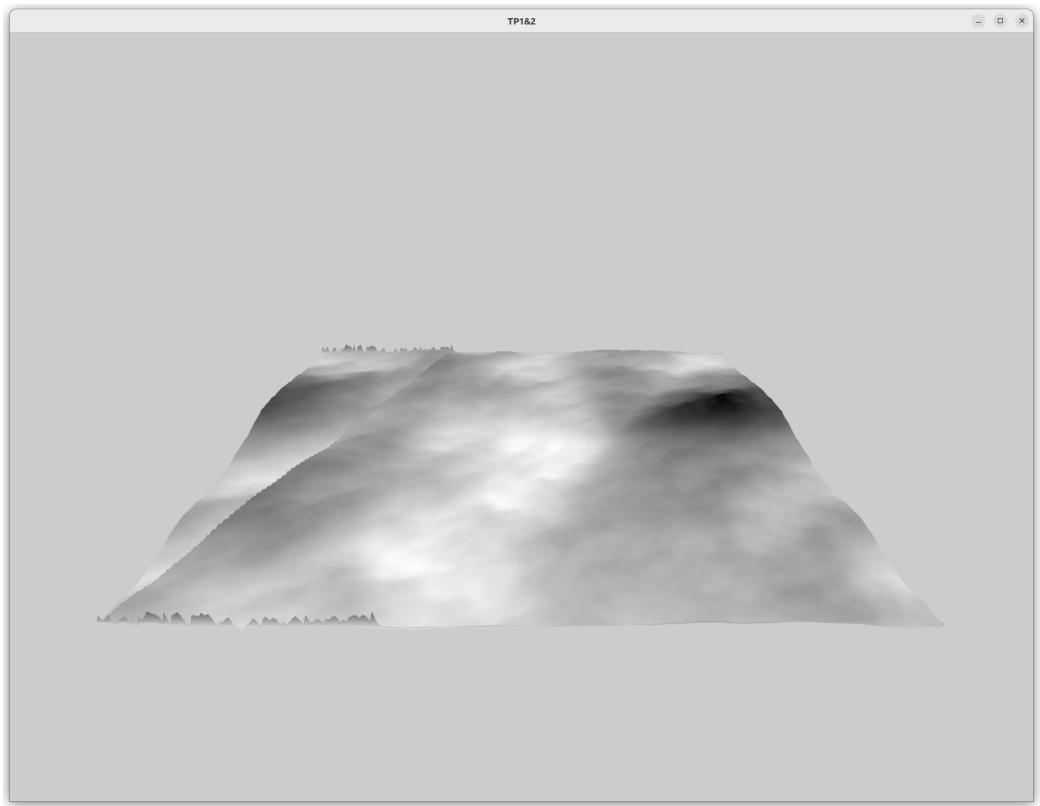


Figure 16: Height map sans amplification

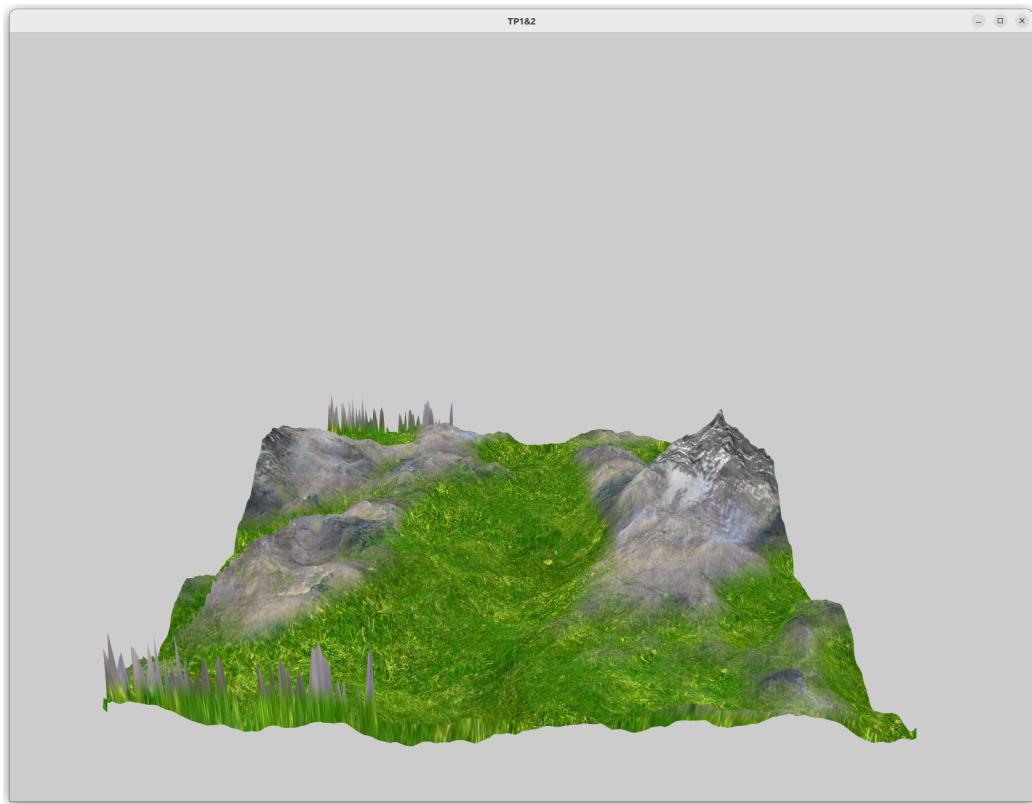


Figure 17: Plan avec height map

## 8 Conclusion

Même si je n'ai pas fait l'intégralité des questions du TP, j'ai pris du plaisir à réaliser ce projet, notamment à apprendre à bien manier la caméra et les interactions utilisateur. Je suis certain que c'est un aspect fondamental lors de la création d'un moteur de jeu, qui sera bientôt une réelle préoccupation.

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.