Moteur de jeu : compte rendu TP1 et TP2

Marie Bocquelet

Jeudi 9 Mars

I TP1 :

I.1 Génération d'un plan :

La génération d'un plan d'une certaine résolution nécessite de calculer les coordonnées de chacun de ses sommets qui ont chacun un décalage de $\frac{1}{résolution}$. Afin de dessiner le plan, il est également nécessaire de calculer les indices de chacun des triangles qui vont former le maillage. Un nouveau sommet constituant le maillage sera donc calculé en fonction des "lignes" et des "colonnes" avec une division par la résolution et ce pour chacune de ses coordonnées :

```
std::vector<glm::vec3> Vertices(int resolution){
    std::vector<glm::vec3> list_vertices;

    for(unsigned int i=0; i<resolution; i++)
    {
        for(unsigned int j=0; j<resolution; j++)
        {
            glm::vec3 vertex = glm::vec3(((float)i/(float)(resolution-1)) - 0.5, 0, ((float)j/(float)(resolution-1)) - 0.5);
            list_vertices.push_back(vertex);
        }
    }

    return list_vertices;

}
```

Afin de centrer le plan en 0, il est important d'enlever 0.5 aux coordonnées x et z.

Les indices des triangles seront calculés par carré formé de deux triangles. Le premier triangle possède les indices supérieur gauche, inférieur gauche et supérieur droit. Le second triangle lui sera constitué des indices supérieur droit, inférieur droit, et inférieur gauche :

```
    std::vector<unsigned short> Indices(int resolution){
    std::vector<unsigned short> list_indices;

    for(unsigned int i=0; i<resolution-1; i++)
    {
        for(unsigned int j=0; j<resolution-1; j++)
        {
            int i1 = i*resolution+j;
            list_indices.push_back(i1);
            int i2 = (i+1)*resolution+j;
            list_indices.push_back(i2);
            int i3 = i*resolution + j+1;
            list_indices.push_back(i3);

            int i4 = i*resolution + j+1;
            list_indices.push_back(i4);
            int i5 = (i+1)*resolution + j;
            list_indices.push_back(i5);
            int i6 = (i+1)*resolution + j+1;
            list_indices.push_back(i6);
        }
    }

    return list_indices;
}
```

Grâce à ceci, OpenGL va comprendre quels indices correspondent à quel triangles. Les positions des sommets sont envoyés au vertex shader et une couleur est ajoutée au niveau du fragment shader. Vous pouvez voir le résultat obtenu ci-dessous :

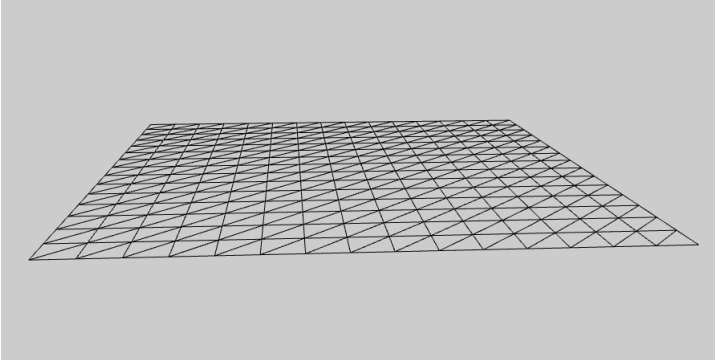


Figure 1: Polygone Mode du plan

I.2 Application d'une texture :

Une fois le terrain obtenu, on peut y appliquer une texture. Le plaquage de texture se fait de la même manière qu'habituellement. On load la texture avec une fonction **loadTexture2DFromFilePath()** et on définit ses coordonnées UV dans le vertex shader qui seront envoyées au fragment shader pour pouvoir afficher la texture grâce à la fonction **texture**. Ci-dessous le rendu obtenu :

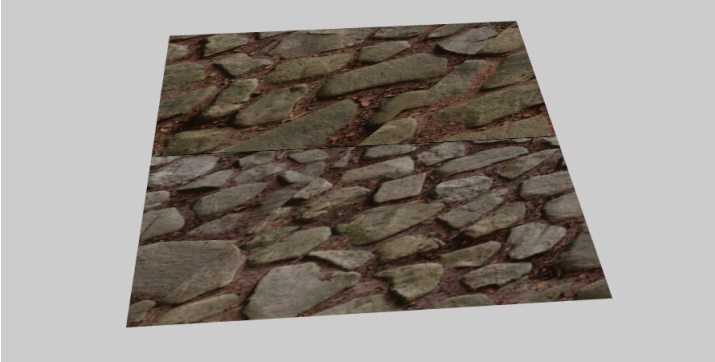


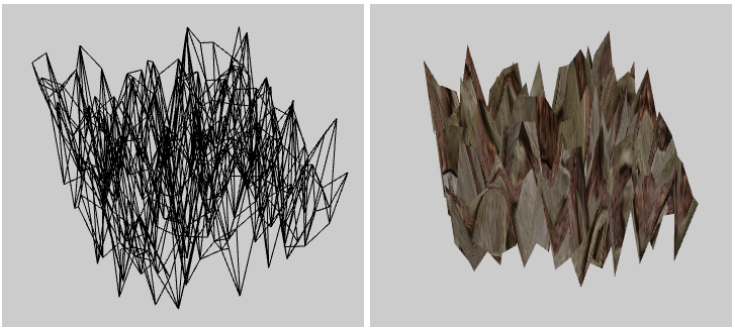
Figure 2: Plaquage de texture

I.3 Ajout de reliefs :

Une fois ceci obtenu, nous souhaitions modifier l'altitude des sommets afin de donner du relief à notre terrain. Pour ceci il suffit de générer des float aléatoire :

```
1       float r = -0.5 + static_cast<float>(rand()) * static_cast<float>(0.5 - (-0.5)) / RAND_MAX;
```

Puis il suffit d'attribuer ces valeurs aux coordonnées y de chacun des sommets constituant notre maillage. On peut voir ci-dessous le résultat :



(a) Maillage polygonal avec modification des altitudes

(b) Application de texture sur le maillage en relief

C'était tout pour le TP1. Passons maintenant au TP2.

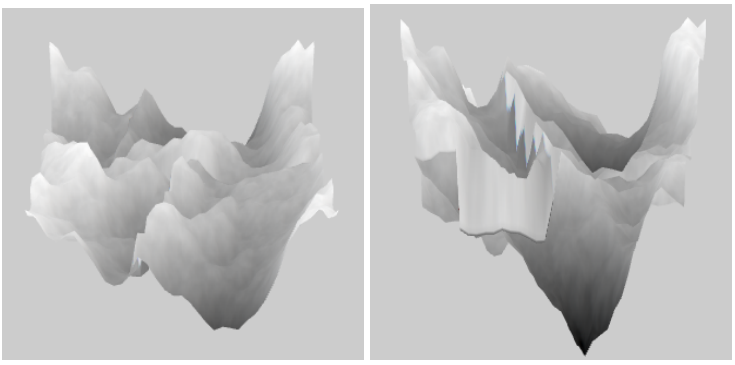
II TP2 :

II.1 Heightmap :

Dans ce début de TP2, nous voulons non plus appliquer des valeurs d'altitude aléatoires aux coordonnées y des sommets, mais appliquer des valeurs d'une heightmap, une heightmap étant une "carte" en nuances de gris dont chaque nuance correspond à une altitude. Ainsi pour appliquer ces valeurs il suffit de récupérer la "texture" heightmap voulue et dans le shader d'appliquer aux coordonnées y de chaque vertex la hauteur correspondant à la nuance de gris de la heightmap :

```
1     uv = vec2(Position.x, Position.z);
2         float height = texture(heightmap, uv).r;
3         vec3 pos=vec3(Position.x, height, Position.z);
4         HEIGHT = height;
5         gl_Position = projection * view * model * vec4(pos, 1.0f);
```

Une fois ceci effectué voici le résultat obtenu :



(a) Terrain avec modification des alti-tudes par la heightmap vue de dessus

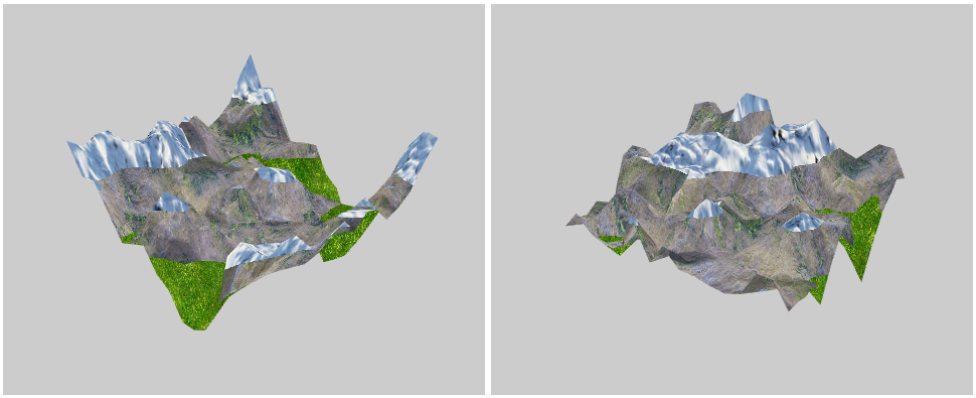
(b) Terrain avec modification des alti-tudes par la heightmap vue de dessous

II.2 Application de textures multiples :

Afin d'appliquer des textures en fonction de la hauteur des sommets il suffit de faire plusieurs seuils de hauteur. Pour ma heightmap j'ai choisit 0.6 et 0.8, et dans le fragment shader j'effectue 3 tests : si la hauteur est inférieure à 0.6 alors on applique la texture herbe, si elle se situe entre 0.6 et 0.8 alors on applique la texture roche et enfin dans les autres cas on applique la texture neige :

```
1       float s1 = 0.6f;
2        float s2 = 0.8f;
3
4        if(HEIGHT<s1){
5          FragColor = texture(grass, uv);
6        }
7        else if(HEIGHT>s1 && HEIGHT<s2){
8          FragColor = texture(rock, uv);
9        }
10       else{
11         FragColor = texture(snow, uv);
12       }
```

Et voici le résultat :



(a) Heightmap mountain texturée (b) Heightmap rocky texturée

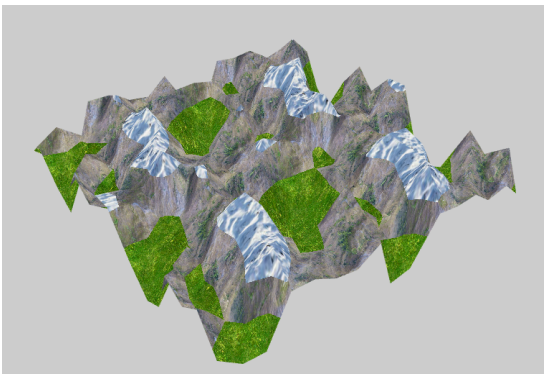
II.3 Modification de la résolution et de la taille :

Afin de pouvoir manipuler mon terrain du mieux possible, j'ai créer des entrées clavier permettant de modifier la résolution et la taille du terrain. En appuyant sur les flèches gauche et droite, vous pouvez augmenter ou diminuer la résolution. Aussi en appuyant sur R ou G vous pouvez respectivement réduire ou agrandir le terrain :

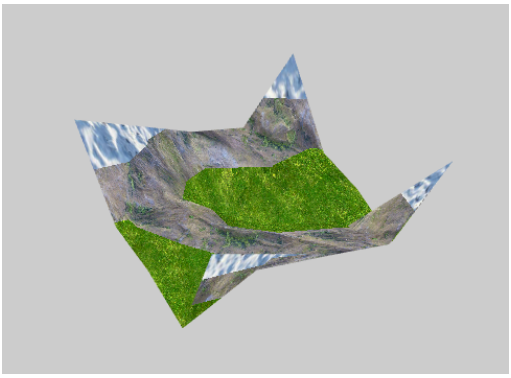
```
1       if(glfwGetKey(window, GLFW_KEY_RIGHT) == GLFW_PRESS){
2           resolution++;
3           indexed_vertices = Vertices(resolution, taille, offset);
4           indices = Indices(resolution);
5           glGenBuffers(1, &vertexbuffer);
6           glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
7           glBufferData(GL_ARRAY_BUFFER, indexed_vertices.size() * sizeof(glm::vec3), &
        indexed_vertices[0], GL_STATIC_DRAW);
8
9           glGenBuffers(1, &elementbuffer);
10          glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
11          glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned short), &
        indices[0] , GL_STATIC_DRAW);
12      }
13
14      if(glfwGetKey(window, GLFW_KEY_LEFT) == GLFW_PRESS){
15          resolution--;
16          indexed_vertices = Vertices(resolution, taille, offset);
17          indices = Indices(resolution);
18          glGenBuffers(1, &vertexbuffer);
19          glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
20          glBufferData(GL_ARRAY_BUFFER, indexed_vertices.size() * sizeof(glm::vec3), &
        indexed_vertices[0], GL_STATIC_DRAW);
21
```

```
22          glGenBuffers(1, &elementbuffer);
23          glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
24          glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned short), &
        indices[0] , GL_STATIC_DRAW);
25      }
26
27      if(glfwGetKey(window, GLFW_KEY_R) == GLFW_PRESS){
28          taille--;
29          offset -= 0.5;
30          indexed_vertices = Vertices(resolution, taille, offset);
31          indices = Indices(resolution);
32          glGenBuffers(1, &vertexbuffer);
33          glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
34          glBufferData(GL_ARRAY_BUFFER, indexed_vertices.size() * sizeof(glm::vec3), &
        indexed_vertices[0], GL_STATIC_DRAW);
35
36          glGenBuffers(1, &elementbuffer);
37          glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
38          glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned short), &
        indices[0] , GL_STATIC_DRAW);
39      }
40
41      if(glfwGetKey(window, GLFW_KEY_G) == GLFW_PRESS){
42          taille++;
43          offset+= 0.5;
44          indexed_vertices = Vertices(resolution, taille, offset);
45          indices = Indices(resolution);
46          glGenBuffers(1, &vertexbuffer);
47          glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
48          glBufferData(GL_ARRAY_BUFFER, indexed_vertices.size() * sizeof(glm::vec3), &
        indexed_vertices[0], GL_STATIC_DRAW);
49
50          glGenBuffers(1, &elementbuffer);
51          glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
52          glBufferData(GL_ELEMENT_ARRAY_BUFFER, indices.size() * sizeof(unsigned short), &
        indices[0] , GL_STATIC_DRAW);
53      }
```

On peut voir ci-dessous deux exemples que ce qui peut être obtenu avec le code ci-dessus :



(a) Terrain agrandit



(b) Résolution diminuée

III Déplacement libre de la caméra :

Pour avoir un déplacement libre de la caméra, il suffit, lors de l'appuit sur une certaine touche du clavier, soit de faire avancer ou reculer la caméra grâce à l'axe z de celle-ci (**cameraFront**) multiplié par la vitesse de celle-ci. Pour monter ou descendre c'est exactement pareil mais en utilisant l'axe y de la caméra (**cameraUp**). Enfin pour aller à gauche ou à droit, il est nécessaire de faire le cross product entre le **cameraUp** et le **cameraFront** afin d'obtenir l'axe x de la caméra qui lui permettra de se déplacer de gauche à droit ou l'inverse :

```
1     if(free_mode){
2             ModelMatrix = glm::mat4(1.f);
3             ViewMatrix = glm::lookAt(cameraPosLibre, cameraPosLibre + cameraFront, cameraUp);
4             ProjectionMatrix = glm::perspective(glm::radians(fov), (float)4/(float)3, 0.1f,
      100.f);
5         }
6         .
7         .
8         .
9         .
10    if(free_mode){
11        if (glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
12            cameraPosLibre -= cameraSpeed * cameraFront;
13        if (glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
14            cameraPosLibre += cameraSpeed * cameraFront;
15        if (glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
16            cameraPosLibre -= glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
17        if (glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
18            cameraPosLibre += glm::normalize(glm::cross(cameraFront, cameraUp)) * cameraSpeed;
19        if(glfwGetKey(window, GLFW_KEY_Z) == GLFW_PRESS)
20            cameraPosLibre += cameraSpeed * cameraUp;
21        if(glfwGetKey(window, GLFW_KEY_X) == GLFW_PRESS)
```

```
22              cameraPosLibre -= cameraSpeed * cameraUp;
23      }
```

Vous pouvez voir dans la vidéo intitulée **mode-orbital-camera** du fichier **video.zip** le résultat obtenu.

III.1 Mode normal de la caméra :

Pour obtenir le mode normal de la caméra, cette fois on utilise le **cameraTarget**. La caméra pointe de manière constante sur l'objet peu importe le mouvement. Dans le même temps il est possible de l'orientée pour observer l'objet avec un angle de 45 degrés. Enfin, afin de pouvoir voir l'objet dans son intégralité, celui ci tourne sur lui même :

```
1       if(normal_mode){
2               ModelMatrix = glm::rotate(ModelMatrix, rotation_speed, glm::vec3(0., 1., 0.));
3               ViewMatrix = glm::lookAt(cameraPosNormal, cameraTarget, cameraUp);
4               ProjectionMatrix = glm::perspective(glm::radians(fov), (float)4/(float)3, 0.1f,
        100.f);
5           }
6           .
7           .
8           .
9           .
10              if(normal_mode){
11          if(glfwGetKey(window, GLFW_KEY_Q) == GLFW_PRESS){
12              cameraPosNormal.y = cameraPosNormal.z;
13              ViewMatrix = glm::lookAt(cameraPosNormal, cameraTarget, cameraUp);
14              glUniformMatrix4fv(glGetUniformLocation(programID, "view"), 1, GL_FALSE, &
        ViewMatrix[0][0]);
15          }
16          if(glfwGetKey(window, GLFW_KEY_UP) == GLFW_PRESS)
17              rotation_speed += 0.01;
18          if(glfwGetKey(window, GLFW_KEY_DOWN) == GLFW_PRESS)
19              rotation_speed -= 0.01;
20      }
```

Vous pouvez voir la vidéo du résultat obtenu dans le dossier **video.zip** fichier **mode-presentation-camera**.

III.2 Mode orbital de la caméra :

Le mode orbital de la caméra consiste au fait de pouvoir tourner autour de l'objet avec la caméra tout en continuant à fixer l'objet. Cela nous permet d'observer l'objet sous tous les angles. Pour ceci, on travaille sur **cameraTarget** afin de ne pas perdre l'objet de vue. Pour pouvoir observer l'objet de dessus ou de dessous il suffit de multiplier le vercteur up de la caméra par caméra speed. Puis pour faire le tour de l'objet il faut faire le cross product entre la direction de la caméra (la soustraction entre le centre de la caméra et le centre de l'objet), le tout normalisé et multiplié par la vitesse de la caméra :

```
1       if(orbit_mode){
2           ModelMatrix = glm::mat4(1.f);
3           ViewMatrix = glm::lookAt(cameraPosOrbit, cameraOrbitTarget, cameraUp);
4           ProjectionMatrix = glm::perspective(glm::radians(fov), (float)4/(float)3, 0.1f, 100.f)
        ;
5           }
6           .
7           .
8           .
9           .
10          if(orbit_mode){
11          if(glfwGetKey(window, GLFW_KEY_S) == GLFW_PRESS)
12              cameraPosOrbit += glm::normalize(cameraUp) * cameraSpeed;
13          if(glfwGetKey(window, GLFW_KEY_W) == GLFW_PRESS)
14              cameraPosOrbit -= glm::normalize(cameraUp) * cameraSpeed;
15          if(glfwGetKey(window, GLFW_KEY_A) == GLFW_PRESS)
16              cameraPosOrbit += glm::normalize(glm::cross(cameraOrbitTarget - cameraPosOrbit,
        cameraUp))*cameraSpeed;
17          if(glfwGetKey(window, GLFW_KEY_D) == GLFW_PRESS)
18              cameraPosOrbit -= glm::normalize(glm::cross(cameraOrbitTarget - cameraPosOrbit,
        cameraUp))*cameraSpeed;
19      }
```

Vous pouvez voir la vidéo du résultat obtenu dans le dossier **video.zip** fichier **orbit-mode-camera**.