Université de Montpellier

M1 - HAI819I - Moteur de Jeux
Compte Rendu - TP1 & TP2

Verniol Baptiste

Année 2022-2023

Sommaire

1. Liste des commandes & Git

— z,q,s,d,a,e : déplacement libre de la caméra

— 3, 4 : augmenter / diminuer la résolution du plan

— p : activer le mode orbite libre

— o : activer le mode orbite automatique

— r,f : accélérer / ralentir la rotation de l'orbite

— w, x : mode de rendu, polygon / wired

Git : https://github.com/Batap2/TP_Moteur_de_Jeux

2. Création du plan

Pour créer un plan il faut définir sa résolution et sa taille, puis placer les points à l'intérieur espacés d'un pas de $\frac{taille}{resolution}$. Ensuite il faut relier les points entres eux en faisant deux triangles par groupe de 4. Puis créer les tableaux nécessaires aux différents buffer pour pouvoir les afficher.

Un tableau des coordonnées des points
Un tableau d'UV
Un tableau de normales
Un tableau avec les indices des points représentants les triangles dans l'ordre.

3. Affichage

Pour pouvoir afficher nos mesh il faut :

- Créer les buffer : glGenBuffers()
- Utiliser les shaders : gmUseProgram()
- Envoyer aux shader les matrice MVP : glGetUniformLocation(programID, "M/V/P")
- Envoyer aux shader les différents tableaux : glBindBuffer() -> glBufferData() -> glEnable-VertexAttribArray()
- Dans le vertexShader appliquer les matrices MVP aux vertex : gl_position = P * V * M * vec4(pos, 1)

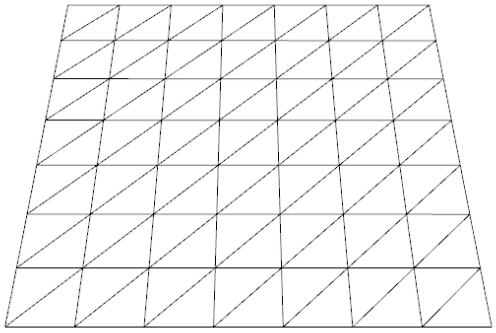

FIGURE 3.1 – 8x8

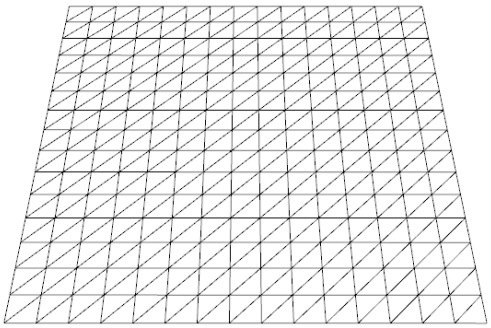

FIGURE 3.2 – 16x16

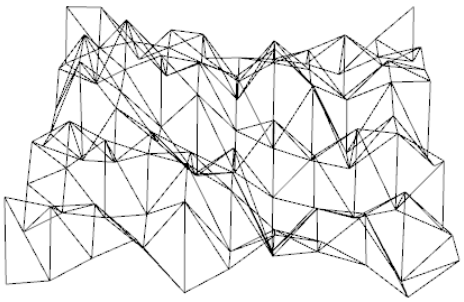

FIGURE 3.3 – Hauteurs random

4. Création et contrôle de la caméra

La caméra est défini par sa position, son vecteur UP et son vecteur FORWARD.
Le but est de faire pointer la caméra vers le mesh, donc on fais pointer le vecteur FORWARD vers le centre du mesh, On peut utiliser glm : :lookAt(cam_pos, target, cam_up) pour définir la view matrix.

Le but de la caméra est d'orbiter autour de l'objet. Il faut alors qu'à chaque instant la distance entre l'objet et la caméra soit constant. Pour calculer la position que devrait prendre la caméra selon deux angles, l'angle horizontal et vertical on utilise les formules :

$$\begin{cases} x & = target.x + radius \times cos(vertAng) \times cos(horiAng) \\ y & = target.y + radius \times sin(vertAng) \\ z & = target.z + radius \times cos(vertAng) \times sin(horiAng) \end{cases}$$

Pour faire tourner automatiquement la caméra autour de l'objet il faut incrémenter l'angle horizontal en fonction de deltaTime.

FIGURE 4.1 – Autre plan après déplacement de la caméra

5. HeightMap

Pour appliquer la heightMap il faut envoyer la texture au shader, puis dans le shader modifier la position du vertex en fonction de la couleur du pixel aux coordonnées UV de la texture heightMap.

pos.y += texture(Sampler2D, UV).r

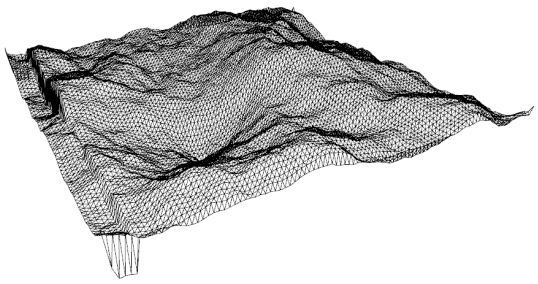

FIGURE 5.1 – En utilisant HM1.bmp

6. Texture & Blending

Pour les textures il faut :

- Ouvrir un canal de texture : glActiveTexture()
- Appeler LoadBMP_custom() qui va charger la texture dans le canal
- Envoyer le canal aux shaders : glUniform1i(glGetUniformLocation(programID, "tex"), 0) qui sera de type Sampler2D dans le shader
- Dans le fragment shader définir color = texture(Sampler2D, UV)

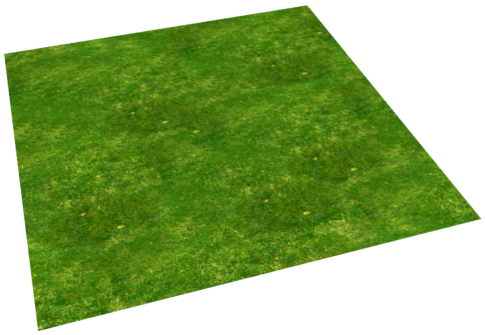

FIGURE 6.1 – En utilisant grass.bmp

Pour mélanger les textures nous pouvons utiliser la fonction glsl mix() qui va faire une interpolation linéaire entre 2 texture en utilisant un poids.
Les poids peuvent être définits avec la fonction smoothstep(hauteur1, hauteur2, valeurHeightMap), c'est grâce à cela que la texture n'apparaîtra qu'à partir d'une certaine hauteur.

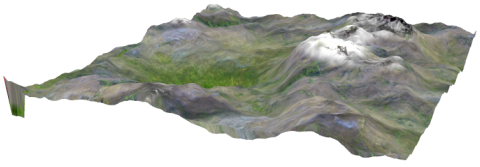

FIGURE 6.2 – En utilisant grass.bmp