INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Informatique Graphique - TP 2

Modélisation

GRAINDORGE Amance

VERNET Hector

Université de Rennes

I. INTRODUCTION

Dans ce TP, nous découvrirons les techniques de modélisation 3D paramétrique et descriptive. Nous créerons d'abord un cylindre, puis nous verrons comment importer des modèles venant d'autres programmes, et enfin nous commencerons à modéliser la tortue pour notre projet.

II. EXERCICE 1

Dans ce premier exercice, on commence par créer un cylindre de rayon et de hauteur 1, centré le long de l'axe z, avec la classe `CylinderMeshRenderable`. Cette classe utilise la fonction `getUnitCylinder`, qu'il nous faut compléter à partir de l'équation d'un cylindre :

```
1   void getUnitCylinder(vector<glm::vec3>& positions, vector<glm::vec3>& normals,
std::vector<glm::vec2>& tcoords, unsigned int slices, bool vertex_normals)
2   {
3     size_t number_of_triangles = slices * 4;              // four triangles per slice
4     size_t number_of_vertices = number_of_triangles * 3;  // three vertices per triangle
(unindexed version)
5     float angle_step = 2.0 * M_PI / double(slices);
6
7     positions.resize(number_of_vertices, glm::vec3(0.0, 0.0, 0.0));
8     normals.resize(number_of_vertices, glm::vec3(0.0, 0.0, 0.0));
9     tcoords.resize(number_of_vertices, glm::vec2(0));
10
11     float previous_angle = (slices - 1) * angle_step;
12     float angle = 0;
13     float previous_cos;
14     float previous_sin;
15     float cos = std::cos(angle);
16     float sin = std::sin(angle);
17
18     for (size_t i = 0; i < slices; ++i)
19     {
20       size_t voffset = 12 * i;  // 4 x 3 = 12 vertices per slice
21       previous_angle = angle;
22       angle += angle_step;
23       previous_cos = cos;
24       previous_sin = sin;
25       cos = std::cos(angle);
26       sin = std::sin(angle);
27
28       // Positions
29
30       // top triangle
31       positions[voffset + 0] = glm::vec3(0.0, 0.0, 1.0);
32       positions[voffset + 1] = glm::vec3(cos, sin, 1.0);
33       positions[voffset + 2] = glm::vec3(previous_cos, previous_sin, 1.0);
34
35       // side triangles
36       positions[voffset + 3] = glm::vec3(cos, sin, 1.0);
37       positions[voffset + 4] = glm::vec3(previous_cos, previous_sin, 1.0);
38       positions[voffset + 5] = glm::vec3(previous_cos, previous_sin, 0.0);
39       positions[voffset + 6] = glm::vec3(previous_cos, previous_sin, 0.0);
40       positions[voffset + 7] = glm::vec3(cos, sin, 0.0);
41       positions[voffset + 8] = glm::vec3(cos, sin, 1.0);
```

```
42
43        // bottom triangle
44        positions[voffset + 9] = glm::vec3(0.0, 0.0, 0.0);
45        positions[voffset + 10] = glm::vec3(cos, sin, 0.0);
46        positions[voffset + 11] = glm::vec3(previous_cos, previous_sin, 0.0);
47
48        // Normals
49
50        // Texture coordinates
51      }
52    }
```

Le cylindre renvoyé par cette fonction est approximé en un certain nombre de parts (`slices`), composées d'un triangle pour la face supérieure, un pour la face inférieure, et deux qui forment le rectangle du côté. On complète également le constructeur de la classe `CylinderMeshRenderable` pour utiliser cette fonction et donner une couleur différente à chaque face du cylindre :

```
1    CylinderMeshRenderable::CylinderMeshRenderable(ShaderProgramPtr shaderProgram, bool
indexed, unsigned int slices, bool vertex_normals) : MeshRenderable(shaderProgram, indexed)
2    {
3      getUnitCylinder(m_positions, m_normals, m_tcoords, slices, vertex_normals);
4      m_colors.resize(m_positions.size(), glm::vec4(0));
5      for (size_t i = 0; i < m_colors.size() / 12; ++i)
6      {
7        glm::vec4 topColor = glm::vec4(1, 0, 0, 1);
8        glm::vec4 botColor = glm::vec4(0, 1, 0, 1);
9        glm::vec4 latColor = glm::vec4(0, 0, 1, 1);
10
11       // Top triangle color
12       m_colors[12 * i + 0] = topColor;
13       m_colors[12 * i + 1] = topColor;
14       m_colors[12 * i + 2] = topColor;
15
16       // Side triangles color
17       m_colors[12 * i + 3] = latColor;
18       m_colors[12 * i + 4] = latColor;
19       m_colors[12 * i + 5] = latColor;
20       m_colors[12 * i + 6] = latColor;
21       m_colors[12 * i + 7] = latColor;
22       m_colors[12 * i + 8] = latColor;
23
24       // Bottom triangle color
25       m_colors[12 * i + 9] = botColor;
26       m_colors[12 * i + 10] = botColor;
27       m_colors[12 * i + 11] = botColor;
28     }
29     update_all_buffers();
30   }
```

On obtient ainsi ce rendu :

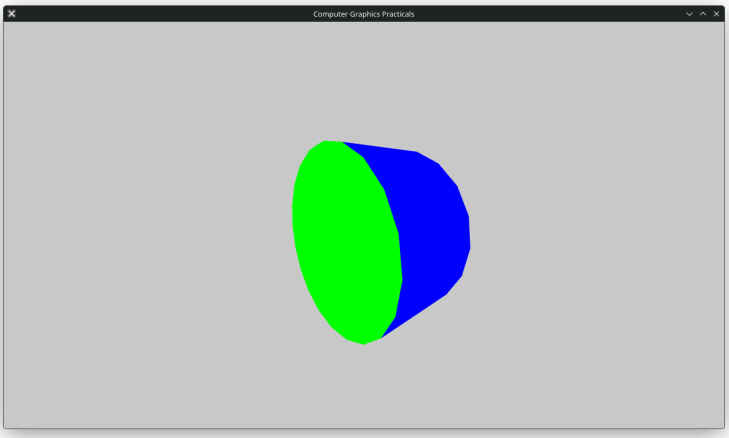

Figure 1 : Cylindre sans normales

III. EXERCICE 2

Ce second exercice ajoute le calcul des normales du cylindre selon deux méthodes : par point ou par triangle. Cet ajout se fait en modifiant la fonction `getUnitCylinder` du fichier `Utils.cpp` :

```
1   // Normals
2
3   // top triangle
4   normals[voffset + 0] = glm::vec3(0.0, 1.0, 0.0);
5   normals[voffset + 1] = glm::vec3(0.0, 1.0, 0.0);
6   normals[voffset + 2] = glm::vec3(0.0, 1.0, 0.0);
7
8   // side triangles
9   if (vertex_normals)
10   {
11     // Per vertex normals
12     normals[voffset + 3] = glm::vec3(cos, sin, 0.0);
13     normals[voffset + 4] = glm::vec3(previous_cos, previous_sin, 0.0);
14     normals[voffset + 5] = glm::vec3(previous_cos, previous_sin, 0.0);
15     normals[voffset + 6] = glm::vec3(previous_cos, previous_sin, 0.0);
16     normals[voffset + 7] = glm::vec3(cos, sin, 1.0);
17     normals[voffset + 8] = glm::vec3(cos, sin, 1.0);
18   }
19   else
20   {
21     // Per triangle normals
22     glm::vec3 n = glm::normalize(glm::vec3((cos + previous_cos) / 2, (sin +
previous_sin) / 2, 0.0));
23     normals[voffset + 3] = n;
24     normals[voffset + 4] = n;
25     normals[voffset + 5] = n;
26     normals[voffset + 6] = n;
27     normals[voffset + 7] = n;
28     normals[voffset + 8] = n;
29   }
30
```

```
31    normals[voffset + 9] = glm::vec3(0.0, -1.0, 0.0);
32    normals[voffset + 10] = glm::vec3(0.0, -1.0, 0.0);
33    normals[voffset + 11] = glm::vec3(0.0, -1.0, 0.0);
```

On change également le vertex shader (`flatVertex.glsl`) pour changer la couleur du point en fonction de sa normale :

```
1    uniform mat4 projMat, viewMat, modelMat;
2
3    in vec3 vPosition;
4    in vec3 vNormal;
5    out vec4 surfel_color;
6
7    void main()
8    {
9        gl_Position = projMat*viewMat*modelMat*vec4(vPosition, 1.0f);
10       int w = 1;
11       vec3 cNormal = vec3(vNormal[0] * 0.5 + 0.5, vNormal[1] * 0.5 + 0.5, vNormal[2] *
0.5 + 0.5); // Change normal vector to stay in [0,1]
12       surfel_color = vec4(cNormal, w);
13    }
```

Cela nous donne le résultat suivant :

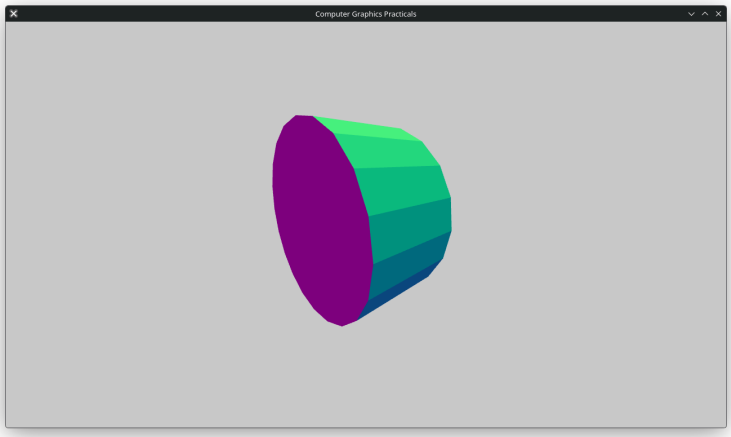

Figure 2 : Cylindre avec normales dans le vertex shader

IV. EXERCICE 3

Pour ce troisième exercice, nous devons importer des mesh stockés dans des fichiers : `cat.obj` et `pillar.obj`. Nous devons également utiliser les transformations pour que l'objet `cat` soit positionné sur l'objet `pillar`. On modifie donc le fichier `practical2.cpp` :

```
1    const std::string cat_path = "../../sfmlGraphicsPipeline/meshes/cat.obj";
2    const std::string pillar_path = "../../sfmlGraphicsPipeline/meshes/pillar.obj";
3    // Import both objects
4    MeshRenderablePtr cat = std::make_shared<MeshRenderable>(flatShader, cat_path);
5    MeshRenderablePtr pillar = std::make_shared<MeshRenderable>(flatShader, pillar_path);
6
7    // Move the cat up
8    cat->setModelMatrix(getTranslationMatrix(0.5,3.2,0));
```

```
9    // Rotate the pillar
10    pillar->setModelMatrix(getRotationMatrix(glm::radians(90.0f), glm::vec3(0.0, 0.0,
1.0)));
11
12    viewer.addRenderable(cat);
13    viewer.addRenderable(pillar);
```

Pour que les couleurs données par les normales soient cohérentes entre les deux objets, il faut modifier le vertex shader pour utiliser les coordonnées globales pour les normales.

```
1    gl_Position = projMat*viewMat*modelMat*vec4(vPosition, 1.0f);
2    vec3 norm = normalize(transpose(inverse(mat3(modelMat))) * vNormal); // Change to
global coordinates
3    int w = 1;
4    vec3 cNormal = vec3(norm[0] * 0.5 + 0.5, norm[1] * 0.5 + 0.5, norm[2] * 0.5 + 0.5);
5    surfel_color = vec4(cNormal, w);
```

On obtient ainsi des couleurs uniforme sur les deux objets :

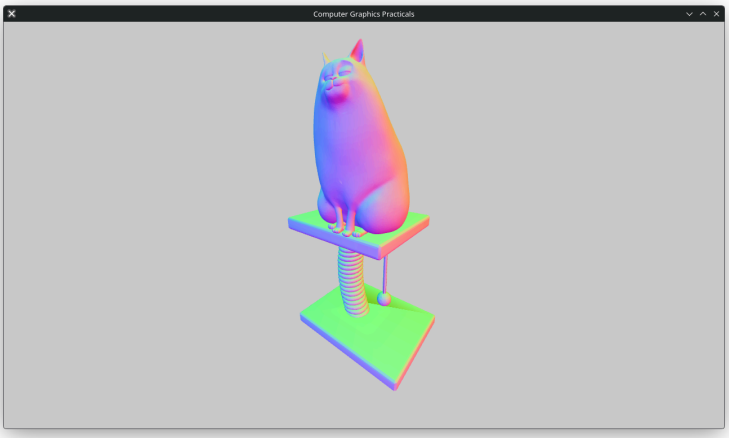

Figure 3 : Chat sur un arbre à chat avec normales dans le vertex shader

V. EXERCICE 4

Cet exercice consiste à commencer la modélisation 3D pour notre projet. Pour cela, nous modéliserons les différents éléments dans Blender. Nous pourrons plus tard les importer dans notre projet au format Wavefront (`.obj`) comme vu plus haut.

Notre scénario (nouvellement écrit) mettant en scène une tortue de mer ainsi qu'une plage, nous commencerons par modéliser cela.

V.1. Tortue

Pour modéliser la tortue, nous utiliserons le workflow dit « subsurf » (Subdivision Surface), qui permet de subdiviser des formes basiques composées de quadrilatères pour obtenir un résultat plus détaillé et organique.

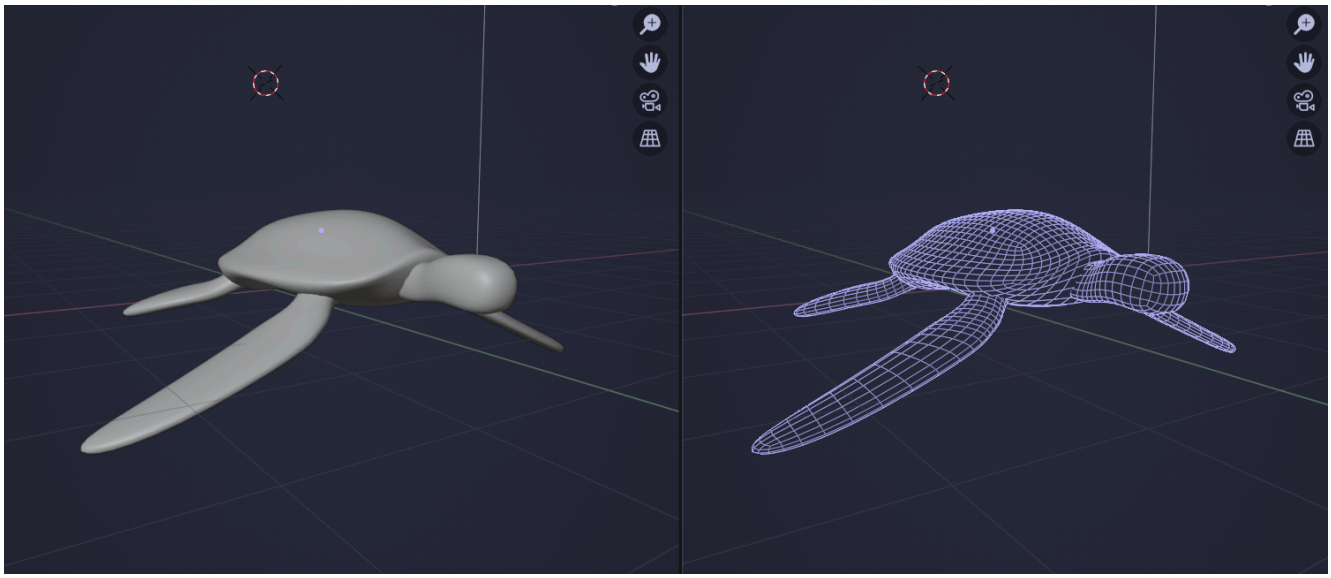

Figure 4 : Notre modèle de tortue dans Blender (mesh après subdivision)

Pour ce qui est des textures, nous verrons ça plus tard.

V.2. Plage

Pour ce qui est de la plage, on peut modéliser le terrain en sculptant un plan subdivisé. Pour l'océan, un modificateur « Océan » existe dans Blender, ce qui permet de générer des vagues automatiquement. Enfin, pour les palmiers, on utilise une texture de feuille trouvée sur internet. Nous verrons comment appliquer des textures dans notre projet à l'occasion du TP7, à priori.

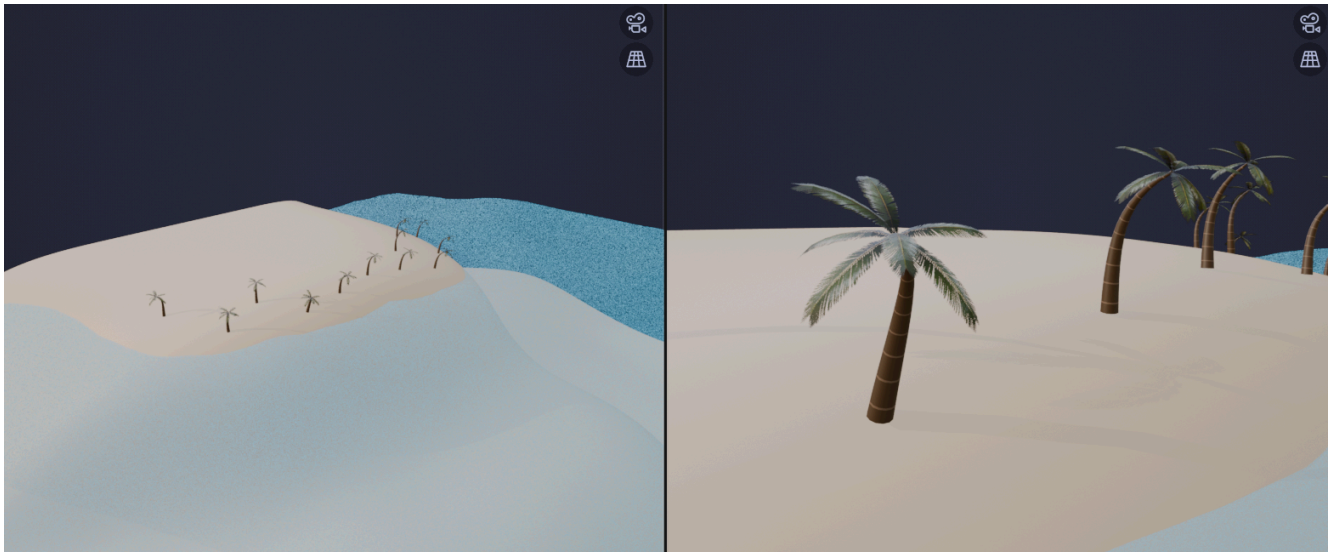

Figure 5 : Modélisation de la tortue dans Blender (mesh après subdivision)

Tout comme la tortue, ce modèle n'est évidemment pas terminé. On envisage notamment de rajouter du corail et des rochers sous l'océan afin de détailler le fond marin, puisqu'une partie de notre scénario s'y déroule.

VI. CONCLUSION

Dans ce TP, nous avons vu comment utiliser la modélisation paramétrique pour créer des primitives comme un cylindre, puis comment utiliser un logiciel comme Blender pour modéliser des objets plus complexes (modélisation descriptive). Nous avons également vu comment

différents logiciels peuvent se transmettre des modèles 3D via des formats standards comme le Wavefront. Nous sommes donc prêts à continuer la modélisation complète de notre scène pour le projet.