INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES RENNES

Informatique Graphique - TP 1

Démarrage

GRAINDORGE Amance

VERNET Hector

Université de Rennes

I. INTRODUCTION

Le but de ce premier TP est d'installer la pipeline fournie par les encadrants et de commencer à s'en servir. Cette pipeline utilise OpenGL comme API, et SFML pour gérer les interactions avec l'utilisateur (fenêtre, entrées clavier). On ira jusqu'à afficher un cube coloré et lui appliquer des transformations.

II. TUTORIELS

II.1. Tutoriels 1 et 2

Les tutoriels 1 et 2 ont simplement pour but de pouvoir build et exécuter le code fourni. Étant tous les deux sur Arch Linux, cela nous a demandé d'apporter quelques modifications au projet de base.

Nous avons dû passer la version de CMake à 3.28 (la dernière) dans tous les `CMakeLists.txt` du projet. Il a également fallu remplacer toutes les directives ayant pour but de trouver les librairies sur le système par des one-liners comme :

```
find_package(SFML 2.6 REQUIRED COMPONENTS graphics) # Exemple pour SFML
```

Et quelques autres changements de syntaxe que vous pouvez retrouver dans ce commit : https://github.com/pixup1/tpinfographique/commit/719ea31629623d5dd193b0f1222b5710582b9d3c.

Après avoir apporté ces modifications, nous pouvons lancer l'exécutable `practical1` et obtenir une fenêtre vide :

Figure 1 : Fenêtre vide

II.2. Tutoriel 3

On va commencer par afficher une « frame », trois lignes dans l'espace 3d. On écrit notre code dans `sampleProject/practical1.cpp` :

```
// Ajout des shaders defaultVertex et defaultFragment
std::string vShader = "./../../sfmlGraphicsPipeline/shaders/defaultVertex.glsl";
std::string fShader = "./../../sfmlGraphicsPipeline/shaders/defaultFragment.glsl";
ShaderProgramPtr defaultShader = std::make_shared<ShaderProgram>(vShader, fShader);
viewer.addShaderProgram(defaultShader);

// Ajout du Renderable "frame"
FrameRenderablePtr frame = std::make_shared<FrameRenderable>(defaultShader);
viewer.addRenderable(frame);
```

Et voici le résultat :

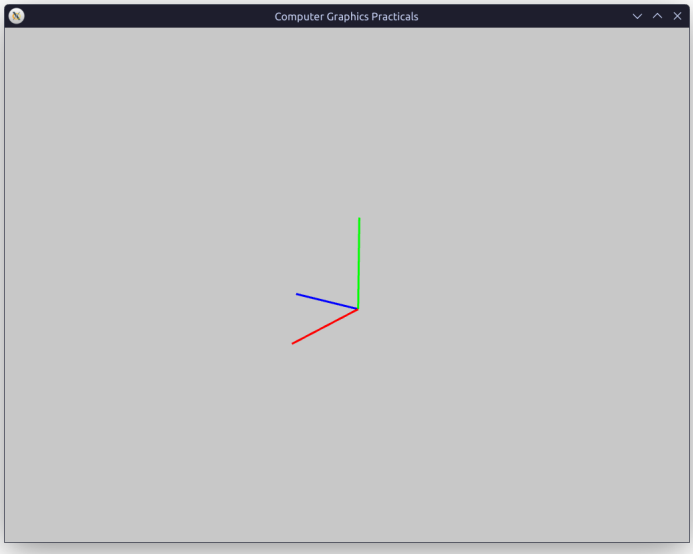

Figure 2 : Lignes

II.3. Tutoriel 4

Dans ce tutoriel, on ajoute un triangle à la scène de la même manière que pour la `frame` :

```
// Ajout des shaders flatVertex et flatFragment
vShader = "./../../sfmlGraphicsPipeline/shaders/flatVertex.glsl";
fShader = "./../../sfmlGraphicsPipeline/shaders/flatFragment.glsl";
ShaderProgramPtr flatShader = std::make_shared<ShaderProgram>(vShader, fShader);
viewer.addShaderProgram(flatShader);

// Ajout du Renderable "cube" (qui sera pour le moment un simple triangle)
CubeRenderablePtr cube = std::make_shared<CubeRenderable>(flatShader);
viewer.addRenderable(cube);
```

On décommente aussi les lignes suivantes :

```
m_positions.push_back( glm::vec3 (-1 ,0 ,0) );
m_positions.push_back( glm::vec3 (1 ,0 ,0) );
m_positions.push_back( glm::vec3 (0 ,1 ,0) );
```

de `sfmlGraphicsPipeline/src/CubeRenderable.cpp` pour ajouter les trois points du triangle. On obtient bien un triangle :

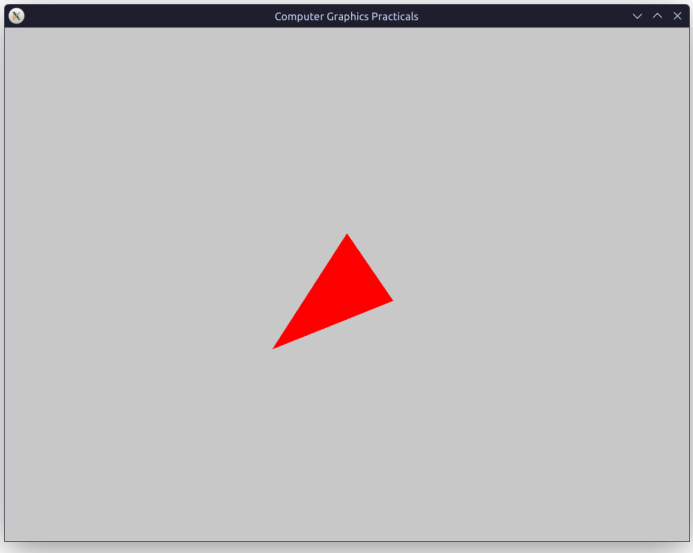

Figure 3 : Triangle

III. EXERCICE 1

Dans cet exercice, on cherche à ajouter un attribut de couleur au vertex shader. Pour cela, on modifie le fichier `sfmlGraphicsPipeline/shaders/flatVertex.glsl` :

```
#version 400

uniform mat4 projMat, viewMat, modelMat;

in vec3 vPosition;
// Ajout d'un nouvel attribut "vColor"
in vec3 vColor;
out vec4 surfel_color;

 void main()
 {
     gl_Position = projMat * viewMat * modelMat * vec4(vPosition, 1.0f);
     int w = 1;
     // La couleur n'est plus fixée à vec4(1, 0, 0, w)
     surfel_color = vec4(vColor, w);
 }
```

On change également la classe `CubeRenderable` pour ajouter des couleurs.

Dans le constructeur :

```
// Ajout de couleurs
m_colors.push_back(glm::vec4(1, 0, 0, 1));
m_colors.push_back(glm::vec4(0, 1, 0, 1));
m_colors.push_back(glm::vec4(0, 0, 1, 1));

// Création d'un buffer pour les couleurs
glGenBuffers(1, &m_cBuffer);
glBindBuffer(GL_ARRAY_BUFFER, m_cBuffer);
```

```
9    glBufferData(GL_ARRAY_BUFFER, m_colors.size() * sizeof(glm::vec4), m_colors.data(),
GL_STATIC_DRAW);
```

Dans la fonction do_draw() :

```
1    // Utilisation de l'attribut "vColor" dans le rendu
2    int colorLocation = m_shaderProgram->getAttributeLocation("vColor");
3    glEnableVertexAttribArray(colorLocation);
4    glBindBuffer(GL_ARRAY_BUFFER, m_cBuffer);
5    glVertexAttribPointer(colorLocation, 4, GL_FLOAT, GL_FALSE, 0, (void *)0);
```

On obtient le rendu suivant :

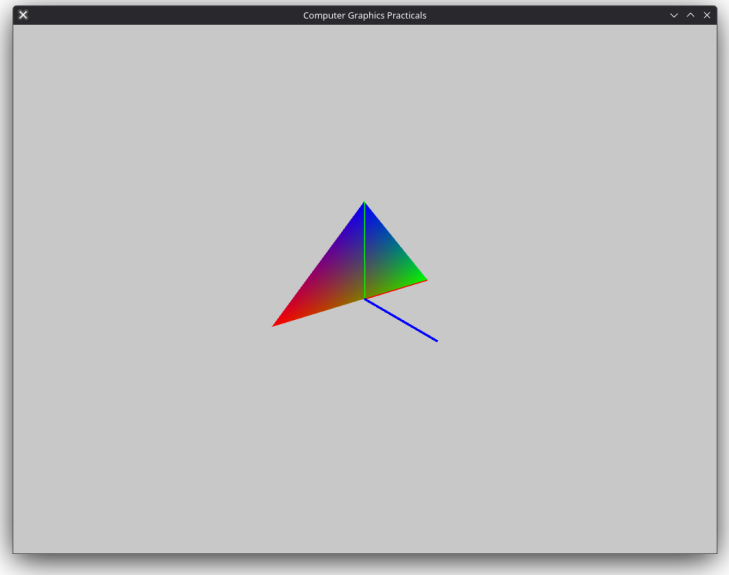

Figure 4 : Triangle coloré

IV. EXERCICE 2

Cet exercice a pour but de créer un cube, mais sans utiliser l'indexation. On change encore une fois le constructeur de `CubeRenderable`. Pour obtenir les points qui vont composer chaque triangle du cube, on utilise la fonction `getUnitCube`. Ensuite, pour que chaque triangle aie une couleur différente, on ajoute trois fois la même couleur dans la liste de couleur (une fois par point du triangle).

```
1    // Récupération de la liste des points
2    std::vector<glm::vec3> normals;
3    std::vector<glm::vec2> uvs;
4    getUnitCube(m_positions, normals, uvs);
5
6    // Couleurs pour chaque face du cube (12 triangles, 3 sommets par triangle)
7    m_colors.push_back(glm::vec4(1, 0, 0, 1)); // Rouge
8    m_colors.push_back(glm::vec4(1, 0, 0, 1));
9    m_colors.push_back(glm::vec4(1, 0, 0, 1));
10    // Même chose pour les 11 autres couleurs
```

On obtient le rendu suivant :

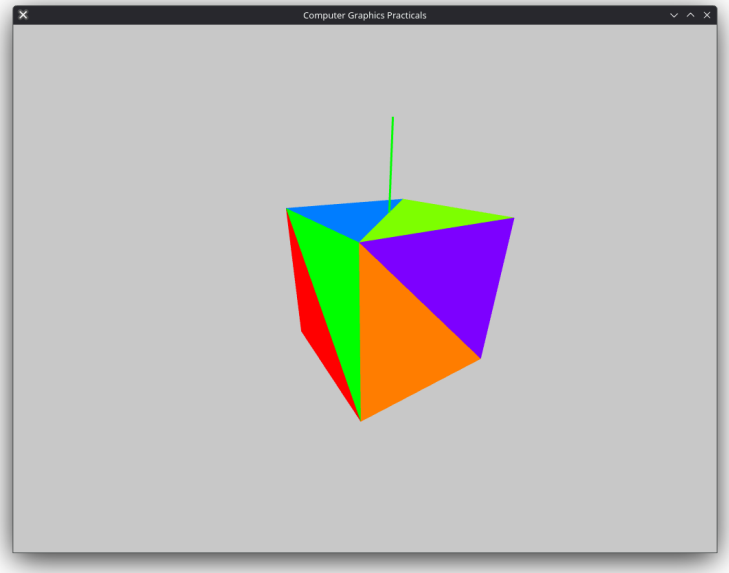

Figure 5 : Cube sans indexation

V. EXERCICE 3

Ce nouvel exercice vise à obtenir un cube en utilisant l'indexation. Les points mentionné dans la liste `m_positions` seront donc utilisables par plusieurs triangles à la fois. Les triangles sont décrits dans la liste `m_indices` par un vecteur contenant les indices des ses trois points dans `m_positions`. On crée donc une nouvelle classe, `IndexedCubeRenderable` :

```
1   IndexedCubeRenderable::IndexedCubeRenderable(ShaderProgramPtr shaderProgram)
2       : Renderable(shaderProgram), m_vBuffer(0), m_cBuffer(0), m_iBuffer(0)
3   {
4     // Positions des sommets du cube
5     m_positions.push_back(glm::vec3(-0.5, -0.5, -0.5));
6     // Idem pour les 7 autres sommets
7
8     // Couleurs pour chaque sommet du cube
9     m_colors.push_back(glm::vec4(0, 0, 0, 1));  // Noir
10     // Idem pour les 7 autres couleurs
11
12     // Indices des sommets pour chaque triangle du cube
13     m_indices.push_back(glm::uvec3(0, 2, 1));
14     // Idem pour les 11 autres triangles
15
16     // Création des buffers contenant les points et les couleurs
17
18     // Création du buffer contenant les indices
19     glGenBuffers(1, &m_iBuffer);
20     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_iBuffer);
21     glBufferData(GL_ELEMENT_ARRAY_BUFFER, m_indices.size() * sizeof(glm::uvec3),
m_indices.data(), GL_STATIC_DRAW);
22   }
23
24   void IndexedCubeRenderable::do_draw()
25   {
```

```
26      // Reste du code, identique à CubeRenderable
27
28      // Rendu des triangles à partir des indices
29      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, m_iBuffer);
30      glDrawElements(GL_TRIANGLES, m_indices.size() * 3, GL_UNSIGNED_INT, (void*)0);
31      glDrawArrays(GL_TRIANGLES, 0, m_positions.size());
32    }
```

On obtient le rendu suivant :

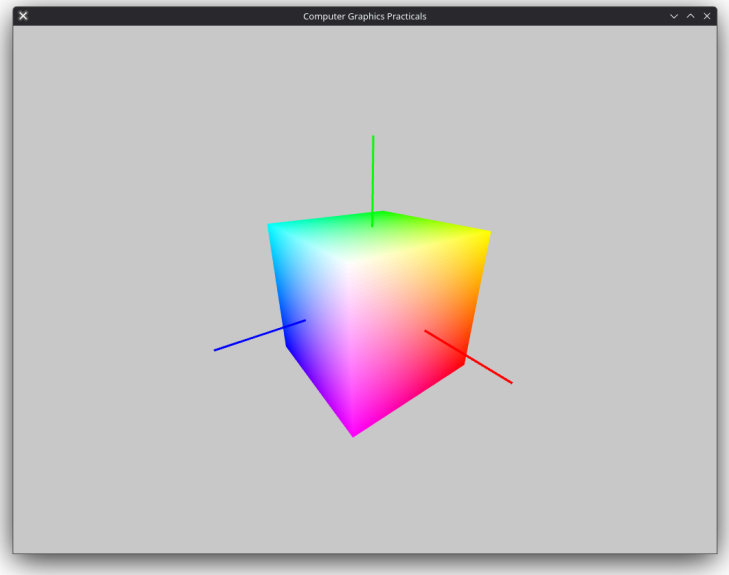

Figure 6 : Cube avec indexation

VI. EXERCICE 4

Ce dernier exercice introduit la transformation (rotation, translation et mise à l'échelle) des objets en montrant un `IndexedCubeRenderable` tourné et un `CubeRenderable` déformé. On applique également une translation aux deux cubes pour qu'ils ne soient pas superposés.

```
1    // IndexedCubeRenderable
2    IndexedCubeRenderablePtr indexedCube =
std::make_shared<IndexedCubeRenderable>(flatShader);
3    // Modèle par défaut du cube
4    glm::mat4 indexedCubeModel = glm::mat4(1.0);
5    // Translation
6    indexedCubeModel = glm::translate(indexedCubeModel, glm::vec3(0.5, 0.5, 0.0));
7    // Rotation
8    indexedCubeModel = glm::rotate(indexedCubeModel, glm::radians(30.0f), glm::vec3(1.0,
1.0, 0.0));
9    // Mise à jour du modèle
10    indexedCube->setModelMatrix(indexedCubeModel);
11
12    // CubeRenderable
13    CubeRenderablePtr cube = std::make_shared<CubeRenderable>(flatShader);
14    // Modèle par défaut du cube
15    glm::mat4 cubeModel = glm::mat4(1.0);
16    // Translation
```

```
17    cubeModel = glm::translate(cubeModel, glm::vec3(-0.5, -0.5, 0.0));
18    // Rotation
19    cubeModel = glm::scale(cubeModel, glm::vec3(0.5, 1, 0.75));
20    // Mise à jour du modèle
21    cube->setModelMatrix(cubeModel);
```

On obtient le rendu suivant :

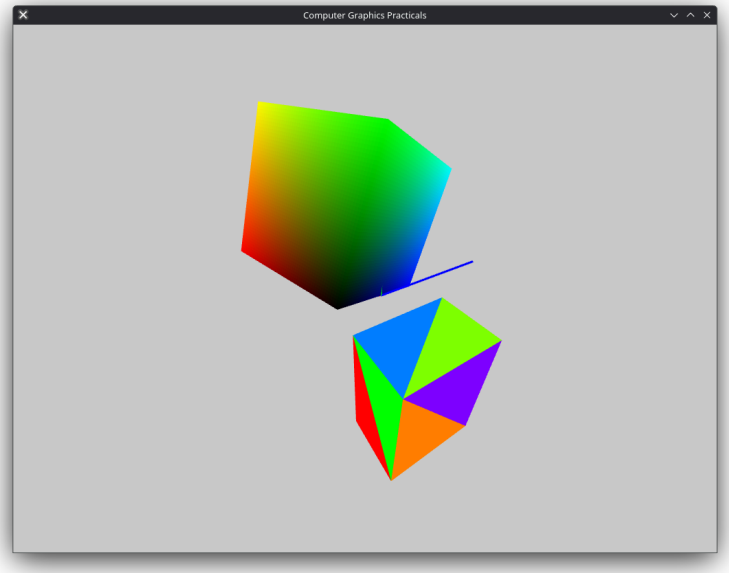

Figure 7 : Cubes transformés

VII. CONCLUSION

Ce premier TP nous a permis de mettre en place et de faire nos premiers pas avec la pipeline OpenGL. Même si nous n'avons pas démarré le projet de zéro, nous avons vu les aspects essentiels du rendu 3D : la modélisation de formes primitives, l'indexation, les shaders, les attibuts (position, couleur) et les transformations d'objets 3D. Ces différentes étapes de la pipeline nous serviront par la suite pour réaliser notre projet.