TP 2 Programmation 3D : GLSL

Mateusz Birembaut

September 30, 2024

Contents

1 Exercice 1 : Compléter la structure TriangleVArray

Dans cet exercice, on va ajouter le code manquant à la structure TriangleVArray, on commence par la fonction initBuffer() : on créer le buffer avec la fonction "glGenBuffer" :

- 1er paramètre : nombre de buffers à créer.
- 2ème : pointeur vers un GLuint pour stocker l'identifiant du buffer.

Ensuite BindBuffer comme openGL est une "machine à états", on dit qu'on va travailler sur le buffer "vertexbuffer". Si on passe cette étape, la ligne d'après va remplir le dernier buffer bind, donc pas forcément ce que l'on veut.

Enfin, pour finir la création du buffer, on va remplir ce buffer avec "glBufferData" :

- 1er paramètre : type du buffer.
- 2ème : la taille en octets du tampon à allouer.
- 3ème : Pointeur vers les données à copier dans le buffer.
- 4ème : "usage", ici "GL STATIC DRAW", car le contenu buffer ne sera pas modifié et utilisé fréquemment.

```
// Creer un premier buffer contenant les positions
// a mettre dans le layout 0
// Utiliser
// glGenBuffers(...);
// glBindBuffer(...);
// glBufferData(...);
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer); // si on met pas cette ligne on utilise le dernier buffer "activé".
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL_STATIC_DRAW);

// Creer un deuxieme buffer contenant les couleurs
// a mettre dans le layout 1
// Utiliser
// glGenBuffers(...);
// glBindBuffer(...);
// glBufferData(...);
glGenBuffers(1, &colorbuffer);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data) , g_color_buffer_data, GL_STATIC_DRAW);
```

Figure 1: Initialisation des buffers

```
void clearBuffers(){
    //Liberer la memoire, utiliser glDeleteBuffers
    glDeleteBuffers(1, &vertexbuffer);
    glDeleteBuffers(1, &colorbuffer);
}
```

Figure 2: Déclaration des variables uniformes

Dans le draw, le plus important est la fonction "glVertexAttribPointer" est utilisée pour spécifier l'emplacement et le format des données de sommets dans un buffer :

- 1er paramètre : le layout, l'indice de l'attribut dans le shader (par exemple : layout(location = 0) in vec3 vertexPosition modelspace;).
- 2ème : le nombre de composantes par attribut (même exemple qu'avant, c'est une position avec vec3, donc 3 pour x, y, z).
- 3ème : Type des composantes (ici floats).
- 4ème : Si vrai : Cela indique que les valeurs stockées dans un format entier doivent être mappées à la plage [-1,1] (pour les valeurs signées) ou [0,1] (pour les valeurs non signées).
- 5ème : "stride", Définit un décalage en octets entre deux lectures. (par exemple si on a un tableau de vec3, et que stride = sizeof(vec3), on va lire un vec3 sur deux).
- 6ème : Un pointeur vers le premier composant de l'attribut dans le tampon. Cela peut servir de décalage dans le buffer (par exemple si à la place de (void*)0, on met (void*)(sizeof(Vec3)), on indique qu'on va commencer la lecture au 2ème vec3).

Il faut aussi ne pas oublier d'activer l'attribut créé avec "glEnableVertexAttribArray(layout)" (le layout = au layout défini dans le shader, ex : si 0 alors ce sera un attribut pour les sommets).

```
void draw (){
    // 1rst attribute buffer : vertices
    //A faire
    //Utiliser glVertexAttribPointer
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
    glEnableVertexAttribArray(0);

    //Ajouter un attribut dans un color buffer à envoyé au GPU
    //Utiliser glVertexAttribPointer
    // 2nd attribute buffer : normals
    glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);

    //1er argument : layout dans shader vertex, on a les 2 layout 0 et 1,  0 pour la position et 1 pour la couleur,
    //2eme : nombre de composantes par attribut de sommet comme on a des vec 3 on a 3,
    //3eme : type de chaque composantes
    //4eme : normalisé, si vrai le gpu normalise entre 0 et 1 ,
    //5eme : stride, 6eme : offset
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (void*) 0);
    // si on avait un seul array avec pos , couleur, pos, couleur, ect... c'est dans le 5eme param qu'on met sizeof
    glEnableVertexAttribArray(1);

    // Draw the triangle !
    // Utiliser glDrawArrays
    glDrawArrays(GL_TRIANGLES, 0, sizeof(vertexbuffer));

    //Pensez à desactive les AttributArray
    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
}
```

Figure 3: Déclaration des variables uniformes

2 Exercice 2 : Variables uniformes

Dans cet exercice, on va utiliser des variables uniformes pour contrôler la mise à l'échelle et la translation d'un modèle 3D. Les variables uniformes sont des variables à définir dans la boucle draw du programme principal et qui sont ensuite transmises au shader à chaque frames.

On commence par définir des variables "loc scale" et "loc translation" de type GLint. Cela permettra de stocker l'emplacement des variables uniformes puis de les envoyer avec la fonction glUniform1f au shader.

```
// Definition des parametre pour le rendu : uniforms etc...
// ajouter une variable uniform pour tous les sommets de type float permettant la mise à l'échelle

// Utiliser glGetUniformLocation pour récuperer l'identifiant GLuint
GLint loc_scale = glGetUniformLocation(programID, "u_scale");

// Ensuite glUniform1f( id_recuperer , valeur );
glUniform1f(loc_scale, scale);

// ajouter une variable uniform pour tous les sommets de type vec3 permettant d'appliquer une translation au modèle
GLint loc_translation = glGetUniformLocation(programID, "u_translation");

glUniform3fv(loc_translation, 1, &translate[0]);
```

Figure 4: Déclaration des variables uniformes

Pour utiliser les variables uniformes précédemment envoyés, on a défini dans le shader deux variables "uniform" avec le même nom que le 2ème paramètre de glUniform1f.

```
#version 330 core
//A faire
// ajouter une variable uniform pour tous les sommets de type float permettant la mise à l'échelle
uniform float u_scale;
// ajouter une variable uniform pour tous les sommets de type vec3 permettant d'appliquer une translation au modèle
uniform vec3 u_translation;

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;


void main(){
    //Mettre à jour ce code pour appliquer la translation et la mise à l'échelle
    gl_Position =  vec4(vertexPosition_modelspace * u_scale + u_translation,1);
}
```

Figure 5: Utilisation variables uniformes dans un shader

Pour modifier les valeurs uniformes, on a juste à modifier les variables globales associées aux variables uniformes (exemple : glUniform1f(loc scale, scale). Si on modifie scale, comme glUniform1f est appelé à chaque frame dans le draw du programme principal, cela va mettre à jour "u scale" en continu. À chaque pression des touches, on effectue une addition ou soustraction sur scale ou translation x, y ou z.

```
case '+': //Press + key to increase scale
    //Completer augmenter la valeur de la variable scale e.g. +0.005
    scale += 0.01;
    //glUniform1f(loc_scale, scale);
    break;

case '-': //Press - key to decrease scale
    //Completer
    scale -= 0.01;
    break;

case 'd': //Press d key to translate on x positive
    //Completer : mettre à jour le x du Vec3 translate
    translate[0] += 0.05;
    break;

case 'q': //Press q key to translate on x negative
    translate[0] -= 0.05;
    break;

case 'z': //Press z key to translate on y positive
    translate[1] += 0.05;
    break;

case 's': //Press s key to translate on y negative
    translate[1] -= 0.05;
    break;
```

Figure 6: Modification des variables uniformes

3 Exercice 3 : Ajouter un attribut couleur

Dans le vertex shader, on définit une variable vec3 color pour stocker la couleur du sommet, le mot clé "out" permet de déclarer une variable de sortie, de "transférer" ce paramètre à la prochaine couche du pipeline graphique, le fragment shader.

"layout(location = 1) in vec3 vertexColor modelspace;" : Cette ligne déclare une variable qui recevra la couleur de chaque sommet.

```
#version 330 core
//A faire
// ajouter une variable uniform pour tous les sommets de type float permettant la mise à l'échelle
uniform float u_scale;

// ajouter une variable uniform pour tous les sommets de type vec3 permettant d'appliquer une translation au modèle
uniform vec3 u_translation;

// ajouter une variable o_color de type vec3 interpolée a envoyer au fragment shader
out vec3 color;

// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;

// A faire : ajouter un attribut de sommet color, contenant les couleurs pour chaque sommet à ( location = 1 )
//------
layout(location = 1) in vec3 vertexColor_modelspace;

void main(){

    //Mettre à jour ce code pour appliquer la translation et la mise à l'échelle
    gl_Position =  vec4(vertexPosition_modelspace * u_scale + u_translation,1);

    //Assigner la normale à la variable interpolée color
    color = vertexColor_modelspace;
}
```

Figure 7: Code Vertex Shader

Dans le fragment shader, on récupère la couleur qui a été passé par le vertex shader avec le mot clé "in".

```
#version 330 core


// Output data
out vec4 FragColor;
// Ajouter une variable interpolée o_color venant du vertex shader
in vec3 color;

void main()
{
    // Mettre à jour la couleur avec la variable interpolée venant du vertex shader
    FragColor = vec4(color, 1.);// Output color = red
}
```

Figure 8: Code Fragment Shader

4 Exercice 4 : Mise à jour de Mesh pour avoir une liste indexée de sommets

On ajoute un buffer de type "GL ELEMENT ARRAY BUFFER" qui stockera les indices des sommets pour chaque triangle du mesh, on parcourt donc tous les triangles et on ajoute leurs sommets dans "indices".

```
void initBuffers(){
    // Creer un premier buffer contenant les positions
    // a mettre dans le layout 0
    glGenBuffers(1, &vertexbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vec3) * vertices.size(), &vertices[0], GL_STATIC_DRAW);

    // Creer un deuxieme buffer contenant les couleurs
    // a mettre dans le layout 1
    glGenBuffers(1, &colorbuffer);
    glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vec3) * normals.size(), &normals[0], GL_STATIC_DRAW);

    //Remplir indices avec la liste des indices des triangles concatenes
    std::vector<unsigned int> indices;
    for (const Triangle& triangle : triangles) {
        indices.push_back(triangle.v[0]);
        indices.push_back(triangle.v[1]);
        indices.push_back(triangle.v[2]);
    }

    // Creer un element buffer contenant les indices des sommets
    glGenBuffers(1, &elementbuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int)*indices.size(), &indices[0], GL_STATIC_DRAW);

}
```

Figure 9: Code modifié Mesh.initBuffer()

Ensuite, dans le draw, on remplace glDrawArray par glDrawElement. on aura au total besoin d'afficher triangles.size()*3 sommets (1 triangle = 3 sommets).

```
void draw (){
    // 1rst attribute buffer : vertices
    glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);
    glEnableVertexAttribArray(0);

    //Ajouter un attribut dans un color buffer à envoyé au GPU
    glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, (void*) 0);
    glEnableVertexAttribArray(1);

    // Draw the triangles !
    // Utiliser l'index buffer
    // glBindBuffer
    // glDrawElements
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
    glDrawElements(GL_TRIANGLES, triangles.size()*3, GL_UNSIGNED_INT, (void*)0);

    //Pensez à desactive les AttributArray
    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);

}
```

Figure 10: Code modifié Mesh.draw()

5 Exercice 5 : Utilisation VAO et attributs entrelacés

Le VAO va stocker une configuration de glVertexAttribPointer pour un maillage. Il améliore les performances en réduisant le nombre d'appels OpenGL nécessaires pour configurer les attributs de sommet.

```
struct Mesh_Position_Color {
    std::vector< Vec3 > vertices_color;
    std::vector< Triangle > triangles;


    GLuint vertices_color_buffer, elementbuffer, vao;

    void initTriangleMesh(){
        std::vector<Vec3> g_vertex_buffer_data {
            Vec3(-1.0f, -1.0f, 0.0f), Vec3(1.0f, 0.0f, 0.0f), //Position, couleur
            Vec3(1.0f, -1.0f, 0.0f), Vec3(0.0f, 1.0f, 0.0f),
            Vec3(1.0f, 1.0f, 0.0f), Vec3(0.0f, 0.0f, 1.0f),

            Vec3(-1.0f, -1.0f, 0.0f), Vec3(1.0f, 0.0f, 0.0f), //Position, couleur
            Vec3(1.0f, 1.0f, 0.0f), Vec3(0.0f, 0.0f, 1.0f),
            Vec3(-1.0f, 1.0f, 0.0f), Vec3(0.0f, 1.0f, 0.0f),
        };

        vertices_color = g_vertex_buffer_data;

        for (int i = 0; i < g_vertex_buffer_data.size(); i = i + 6){
            triangles.push_back(Triangle(i, i+2, i+4));
        }

    }
```

Figure 11: Code Struct Mesh color position

```
void initBuffers(){
    // comme avant mais je met 1 buffer pour les position et couleurs et pas deux buffers donc on stock tout dans vertices_color_buffer
    glGenBuffers(1, &vertices_color_buffer);
    glBindBuffer(GL_ARRAY_BUFFER, vertices_color_buffer);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vec3) * vertices_color.size(), &vertices_color[0], GL_STATIC_DRAW);

    // comme avant
    std::vector<unsigned int> indices;
    for (const Triangle& triangle : triangles) {
        indices.push_back(triangle.v[0]);
        indices.push_back(triangle.v[1]);
        indices.push_back(triangle.v[2]);
    }

    // comme avant
    glGenBuffers(1, &elementbuffer);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(unsigned int)*indices.size(), &indices[0], GL_STATIC_DRAW);
```

Figure 12: Code Mesh color position InitBuffer() Part 1

Dans les "glVertexAttribPointes", comme on utilise qu'un buffer pour stocker les positions et couleurs et qu'on alterne 1 vec3 "position" puis 1 vec3 "couleur", il faut préciser que quand on veut des positions, on commence à l'élément 0 et on "saute" 1 vec3 à chaque fois. Pour les couleurs, on commence à lire à partir de sizeof(vec3) = on commence à l'indice 1 et on "saute" 1 vec3 à chaque fois.

```
// generation et liaison du vao
// il va stocker les glVertexAttribPointer pour ce mesh
// parce que dans les anciennes verions d'open GL les glVertexAttribPointer sont globaux
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

// comme avant
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vertices_color_buffer);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vec3), (void*)0);

// je met "vertices_color_buffer" dans le 2eme param de bindbuffer et pas "colorbuffer" car j'ai tout mis dans un seul buffer
glEnableVertexAttribArray(1);
glBindBuffer(GL_ARRAY_BUFFER, vertices_color_buffer);
// je précise que je commence a lire a partir de sizeof(Vec3) pour lire les couleurs et je saute sizeof(Vec3) a chaque pour pas lire les positions
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vec3), (void*)(sizeof(Vec3)));

// delier le vao pour pas se soit modifié par erreur
glBindVertexArray(0);
```

Figure 13: Code Mesh color position InitBuffer() Part 2

```
void draw() {
    // lier le vao pour recup les glVertexAttribPointer de ce mesh
    glBindVertexArray(vao);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementbuffer);
    glDrawElements(GL_TRIANGLES, triangles.size() * 3, GL_UNSIGNED_INT, (void*)0);

    // delier le vao
    glBindVertexArray(0);
}
```

Figure 14: Code Mesh color position draw()

6 Résultat

```
struct Mesh_Position_Color {
    std::vector< Vec3 > vertices_color;
    std::vector< Triangle > triangles;


    GLuint vertices_color_buffer, elementbuffer, vao;

    void initTriangleMesh(){
        std::vector<Vec3> g_vertex_buffer_data {
            Vec3(-1.0f, -1.0f, 0.8f), Vec3(1.0f, 0.0f, 0.0f), //Position, couleur
            Vec3(1.0f, -1.0f, 0.8f), Vec3(1.0f, 0.0f, 0.0f),
            Vec3(1.0f, 1.0f, 0.8f), Vec3(0.0f, 0.0f, 1.0f),

            Vec3(-1.0f, -1.0f, 0.8f), Vec3(1.0f, 0.0f, 0.0f), //Position, couleur
            Vec3(1.0f, 1.0f,0.8f), Vec3(0.0f, 0.0f, 1.0f),
            Vec3(-1.0f, 1.0f, 0.8f), Vec3(0.0f, 0.0f, 1.0f),
        };
```

Figure 15: 2 triangles pour le fond

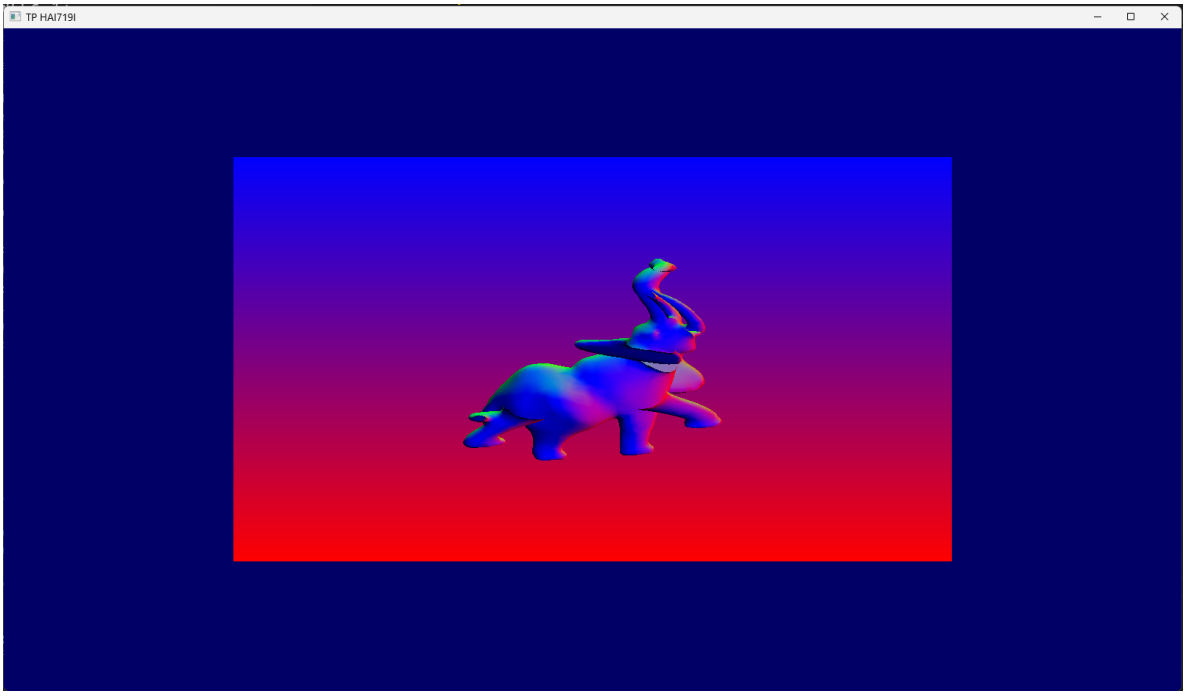

Figure 16: Affichage mesh.draw() et "triangle mesh position color".draw(); (avec scale modifié)