Programmation 3D

Compte-rendu TP2

GLSL

Louis Jean
Master 1 IMAGINE
Université de Montpellier

28 septembre 2023

Table des matières

1 Introduction

Le but de ce TP était d'afficher une scène simple, en l'occurrence, un simple triangle) en paramétrant le pipeline OpenGL, c'est-à-dire configurer la manière dont OpenGL affiche les objets en chargeant et paramétrant les programmes appelés shaders (fichiers .glsl).

2 Affichage d'un simple triangle

Pour cette partie, nous avons uniquement utilisé la structure TriangleVArray et l'avons remplie de manière à pouvoir envoyer les données de notre triangle directement sur le GPU, pour que les shaders le traitent et qu'il soit affiché. Ces lignes de code ci-dessous permettent la création et la liasion de deux buffers : un qui contient les sommets du triangle et un autre qui contient ses couleurs.

```
// 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(g_vertex_buffer_data),g_vertex_buffer_data,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(g_color_buffer_data), g_color_buffer_data, GL_STATIC_DRAW);
```

Figure 1: Création et liaison des buffers dans la struct TriangleVArray

Cette fonction permet la suppression des buffers (sur le CPU) après leur envoi.

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

Figure 2: Gestion de la suppression des buffers

Ici, on a écrit une fonction draw, qui explique au GPU quoi faire des buffers que l'on lui a envoyé. Les deux premières lignes permettent d'activer les AttributArray sur les layouts 0 et 1. Les appels à glVertexAttribPointer précisent au GPU comment nos buffers sont constitués. Ensuite, on appelle glDrawArrays pour dessiner des triangles, puis on désactive les AttributArray que l'on a activé au début.

```
void draw (){
    // 1rst attribute buffer : vertices
    // Activer les AttributArray
    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    //A faire
    //Utiliser glVertexAttribPointer
    glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,3*sizeof(float),(void*)0);

    //Ajouter un attribut dans un color buffer à envoyé au GPU
    //Utiliser glVertexAttribPointer
    // 2nd attribute buffer : colors
    glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,3*sizeof(float),(void*)0);

    // Draw the triangle !
    // Utiliser glDrawArrays
    glDrawArrays(GL_TRIANGLES,0,3);

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

Figure 3: Code de la fonction draw dans la struct TriangleVArray

Voici le résultat.

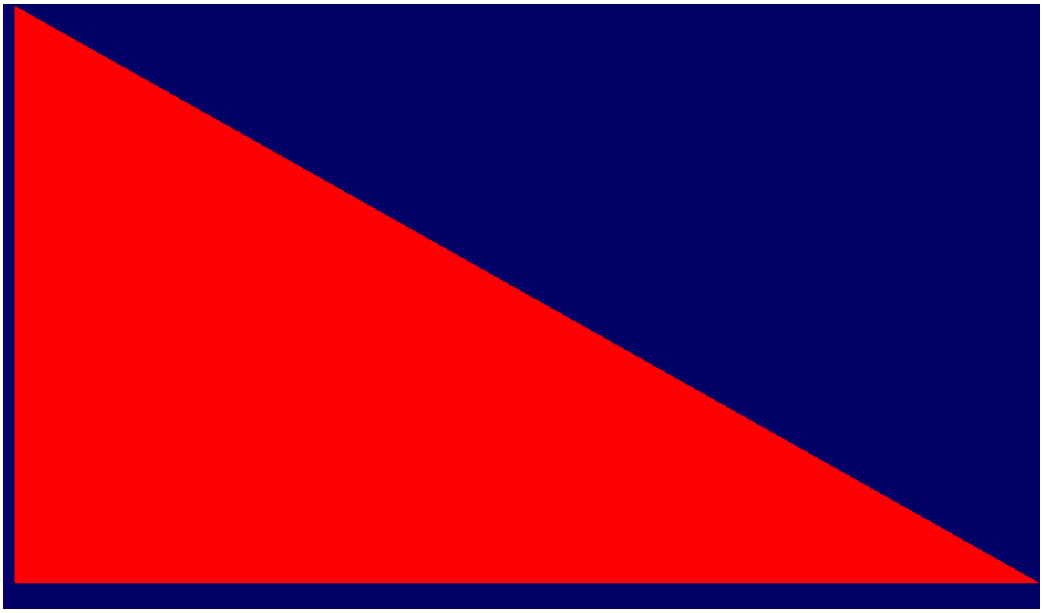

Figure 4: Affichage du triangle suite à la complétion de la struct TriangleVArray

3 Mise à l'échelle et translation du triangle

Dans cette question, il était demandé d'appliquer au triangle une mise à l'échelle uniforme et une translation, le tout en utilisant des variables uniform à passer au shader. Il fallait pouvoir augmenter et diminuer la mise à l'échelle en appuyant respectivement sur les touches + et -. Aussi, la translation en X et en Y devait pouvoir être contrôlée respectivement par les touches q,d,z,s.

3.1 Mise à l'échelle uniforme

Nous avons choisi de commencer par la mise à l'échelle uniforme. Pour cela, la déclaration de deux variables globales était nécessaire.

```
float scale;
GLfloat scaleLocation;
```

Figure 5: Déclaration de deux variables globales

```
uniform float scale;
```

Figure 6: Déclaration d'une variable unifom dans le vertex shader

Les deux lignes de code ci-dessous, placées dans le main, permettent la liaison de la variable scaleLocation à la position de la variable scale dans le vertex shader et l'initialisation de cette dernière. Ainsi, on pourra modifier la variable du vertex shader directement depuis le code C++ de notre application. programID contient l'identifiant du programme vertex_shader.glsl, retourné par la fonction load_shaders (donnée dans la base de code fournie).

```
GLfloat scaleLocation = glGetUniformLocation(programID,"scale");
glUniform1f(scaleLocation,1);
```

Figure 7: Code dans le main

Pour permettre l'ajustement de la mise à l'échelle selon l'appui sur certaines touches, voici comment nous avons adapté le switch de la fonction key déjà fourni dans l'archive de code. glUniform1f permet d'aller modifier une variable uniform float présente dans un shader.

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

case '-': //Press - key to decrease scale
    //Completer
    scale -= 0.005;
    glUniform1f(scaleLocation,scale);
    break;
```

Figure 8: Ajout dans le switch de la fonction key

Enfin, voici la modification que nous avons apporté dans le vertex shader, afin de mettre à l'échelle chaque sommet du triangle.

```
gl_Position =  vec4(scale * vertexPosition_modelspace,1);
```

Figure 9: Mise à l'échelle de chaque sommet dans le vertex shader

Après toutes ces étapes, voilà les résultats obtenus à l'affichage.

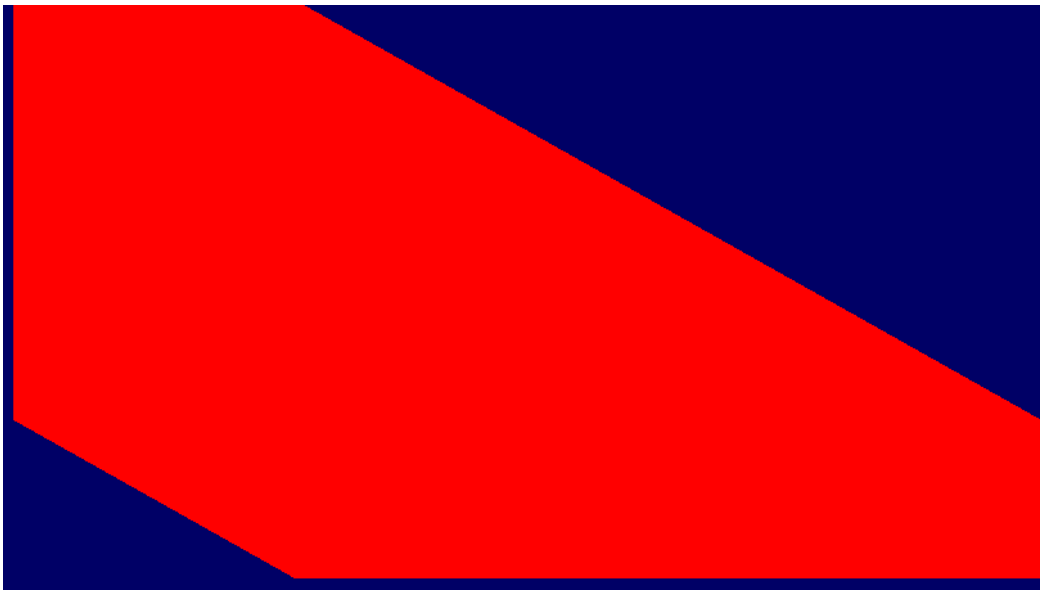

Figure 10: Affichage du triangle après plusieurs appuis sur la touche +

Ci-dessus, on obtient ce rendu à cause de l'implémentation de la caméra dans le code. En effet, il est possible que le triangle sorte des deux plans entre lesquels les éléments de la scène sont rendus, le tronquant ainsi.

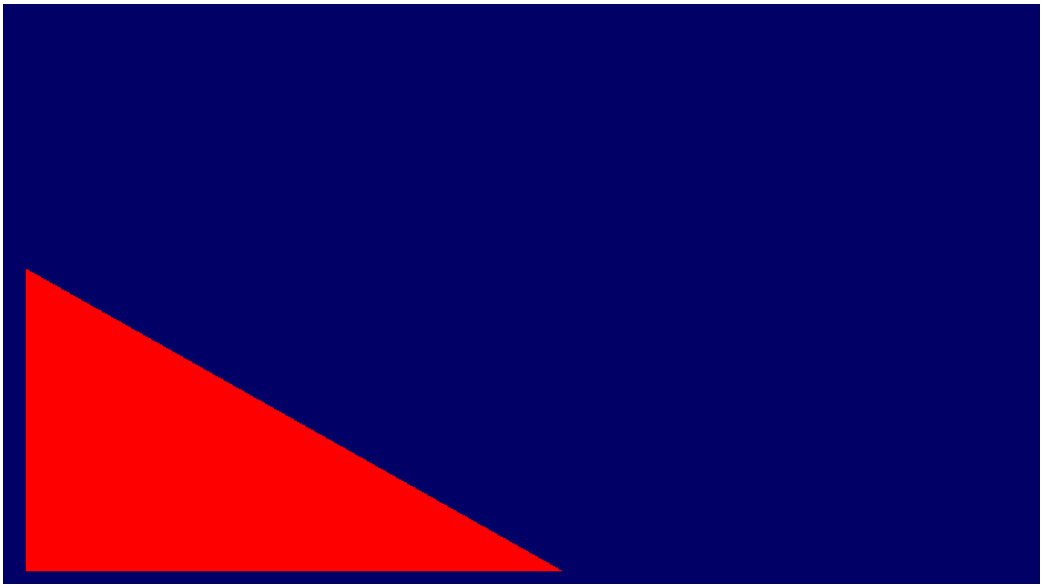

Figure 11: Affichage du triangle après plusieurs appuis sur la touche -

3.2 Translation

Pour ce qui est de la translation sur X et sur Y, le procédé est très similaire.

```
Vec3 translation;
GLint translationLocation;
```

Figure 12: Déclaration de deux variables globales

```
uniform vec3 translation;
```

Figure 13: Déclaration d'une variable unifom dans le vertex shader

```
translationLocation = glGetUniformLocation(programID,"translation");
glUniform3fv(translationLocation,1,&translation[0]);
```

Figure 14: Code dans le main

```
case 'd': //Press d key to translate on x positive
    translation[0] += 0.05;
    glUniform3fv(translationLocation,1,&translation[0]);
    break;

case 'q': //Press q key to translate on x negative
    translation[0] -= 0.05;
    glUniform3fv(translationLocation,1,&translation[0]);
    break;

case 'z': //Press z key to translate on y positive
    translation[1] += 0.05;
    glUniform3fv(translationLocation,1,&translation[0]);
    break;

case 's': //Press s key to translate on y negative
    translation[1] -= 0.05;
    glUniform3fv(translationLocation,1,&translation[0]);
    break;
```

Figure 15: Ajout dans le switch de la fonction key

```
gl_Position =  vec4(scale * vertexPosition_modelspace + translation,1);
```

Figure 16: Application de la translation à chaque sommet dans le vertex shader

Après quelques tests, voici ce que nous avons obtenu.

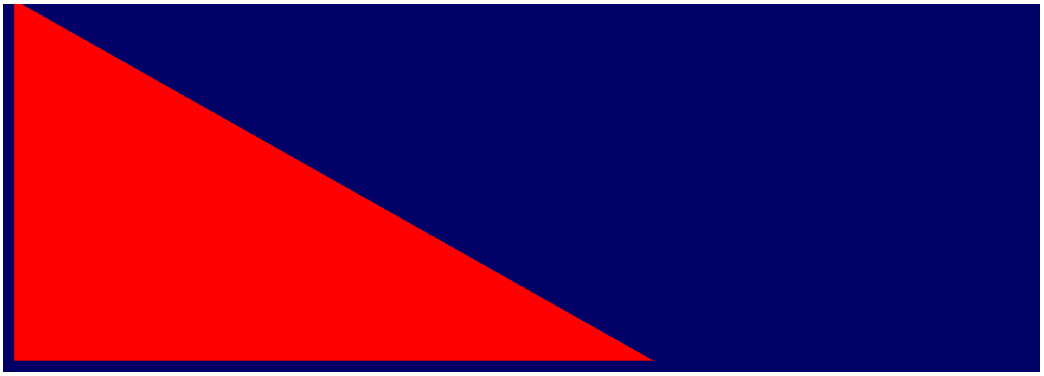

Figure 17: Affichage du triangle après plusieurs appuis sur la touche q (déplacement vers la gauche)

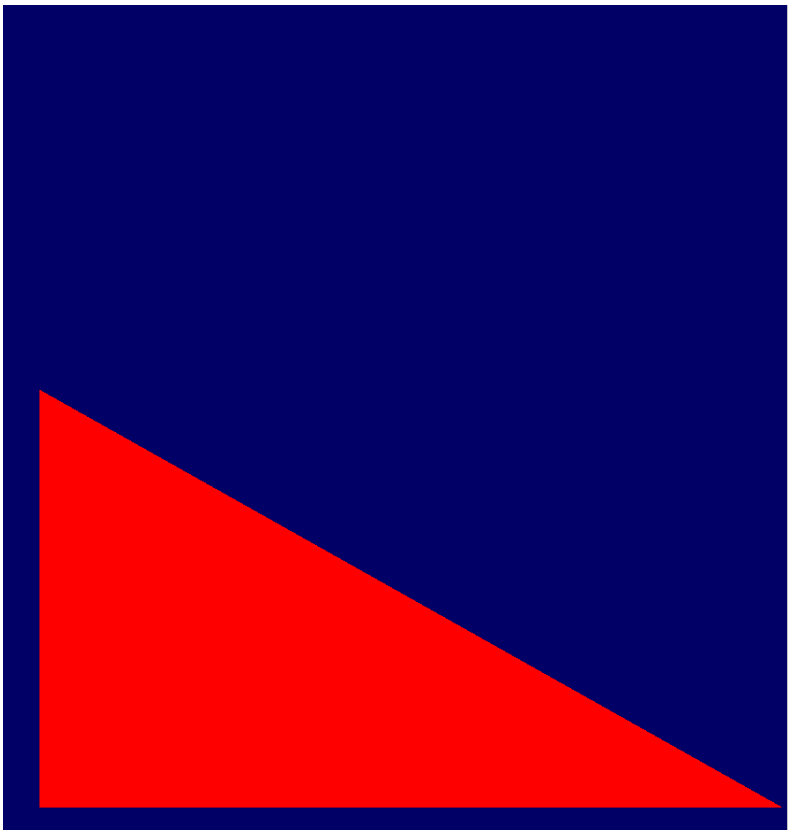

Figure 18: Affichage du triangle après plusieurs appuis sur la touche s (déplacement vers le bas)

4 Ajout de couleurs

Jusqu'ici, la couleur (rouge) du triangle était déterminée directement en dur dans le fragment shader. Autrement dit, la liaison du color_buffer que nous avons effectué plus haut était inutile. Mais elle allait désormais nous servir ! Il fallait d'abord ajouter un attribut couleur par sommet à la location 1 dans le vertex shader.

```
layout(location = 1) in vec3 colorValues_modelspace;
```

Figure 19: Ajout d'un attribut de couleur par sommet

Ensuite, il a fallu créer une variable out dans le vertex shader, qui servirait à passer la couleur de chaque sommet dans le fragment shader.

```
out vec3 o_color;
```

Figure 20: Création d'une variable à passer au fragment shader

Pour chaque sommet, on a initialisé la valeur de sa couleur par les valeurs correspondantes dans le buffer déjà lié.

```
o_color = colorValues_modelspace;
```

Figure 21: Initialisation de la variable out avec les valeurs du color_buffer

Bien sûr, nous avons créé une variable in dans le fragment shader, correspondant à la variable out du vertex shader.

```
// Ajouter une variable interpolée o_color venant du vertex shader
in vec3 o_color;
```

Figure 22: Création d'une variable in dans le fragment shader

Pour finir, nous avons mis à jour FragColor avec les valeurs de o_color.

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

Figure 23: Mise à jour de la couleur grâce à o_color

Les variables in (et out) étant interpolées, cela s'est traduit ici par une interpolation des couleurs entre les différents sommets. Voici les résultats en découlant.

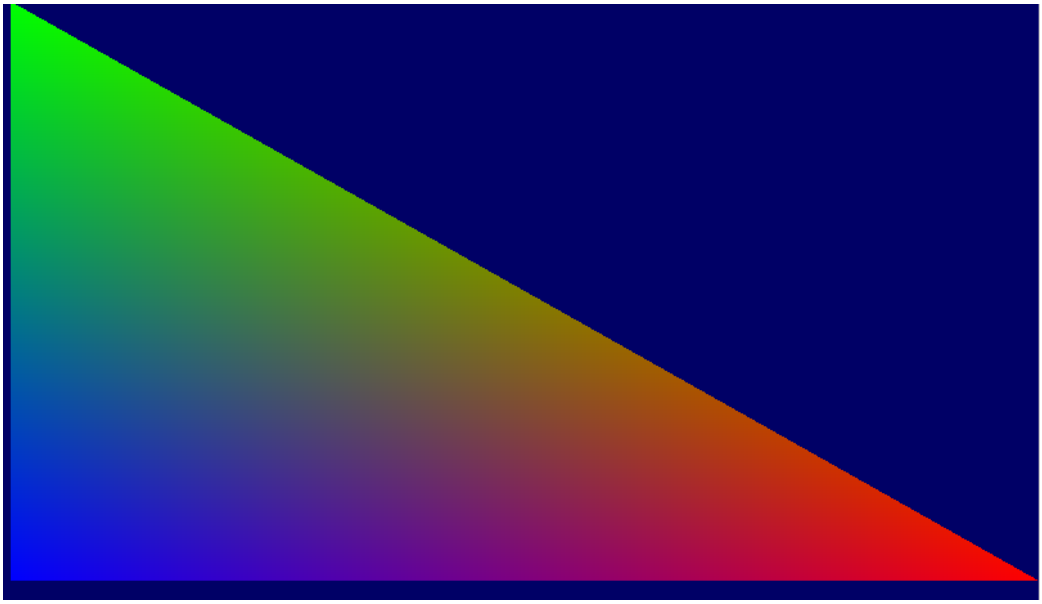

Figure 24: Résultat de la mise à jour des couleurs des sommets du triangle

5 Mise à jour de la structure Mesh

Dans cette partie, l'objectif était d'adapter la structure Mesh pour pouvoir envoyer les données de notre triangle au GPU sous forme de liste indexée de sommets. De la même manière que dans la première partie, nous avons créé et lié les différents buffers : sommets, couleurs et indices de sommets, sans oublier la gestion de leur suppression. Ce qui différe vraiment de la première partie sont les arguments de taille des éléments dans les buffers.

```
glGenBuffers(1,&vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER,vertexbuffer);
glBufferData(GL_ARRAY_BUFFER,sizeof(Vec3)*vertices.size(),&vertices[0],GL_STATIC_DRAW);

glGenBuffers(1,&colorbuffer);
glBindBuffer(GL_ARRAY_BUFFER,colorbuffer);
glBufferData(GL_ARRAY_BUFFER,sizeof(Vec3)*normals.size(),&normals[0],GL_STATIC_DRAW);

std::vector<unsigned int> indices;
for(auto& t : triangles) {
    indices.push_back(t[0]);
    indices.push_back(t[1]);
    indices.push_back(t[2]);
}

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 25: Création et liaison des buffers dans la struct Mesh

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

Figure 26: Gestion de la suppression des buffers

Pour la fonction draw, dans l'appel à la fonction drawElements, il a fallu expliquer au GPU comment naviguer dans la liste indexée de sommets pour dessiner les triangles.

```
void draw () {

    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glEnableVertexAttribArray(2);

    glBindBuffer(GL_ARRAY_BUFFER,vertexbuffer);
    glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,sizeof(Vec3),(void*)0);

    glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
    glVertexAttribPointer(1,3,GL_FLOAT,GL_FALSE,sizeof(Vec3),(void*)0);

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

    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);
    glDisableVertexAttribArray(2);
}
```

Figure 27: Gestion de la suppression des buffers

Après avoir créé un Mesh triangle_mesh et avoir fait les bons appels aux bons endroits, voici ce que nous avons obtenu.

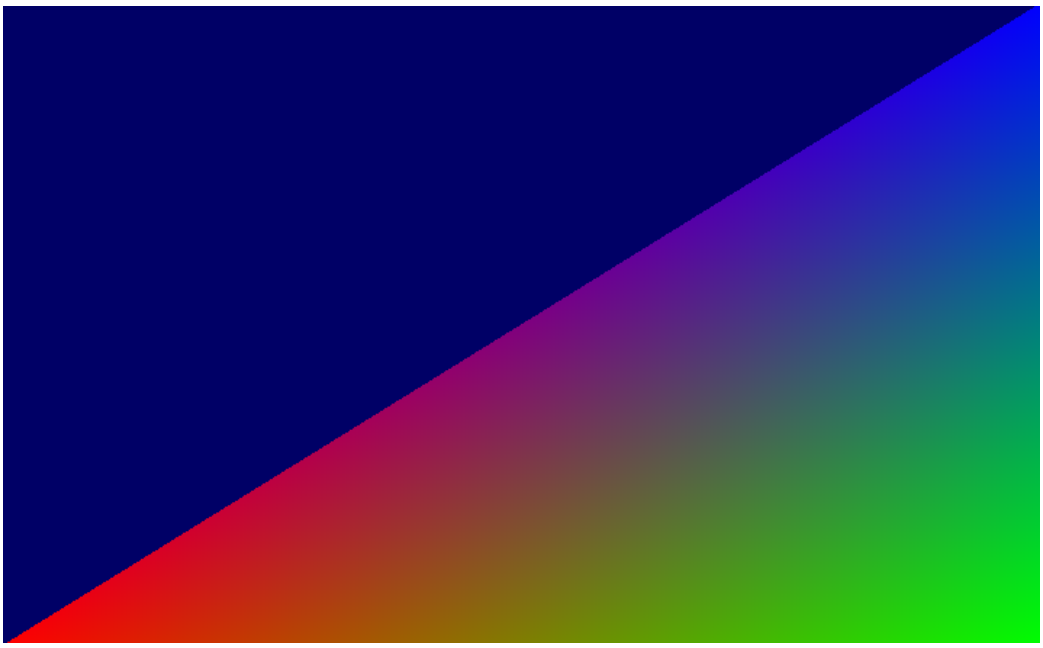

Figure 28: Affichage du triangle en tant que liste indexée de sommets

6 Bonus

6.1 Affichage des sommets et couleur avec un seul buffer

Cet exercice bonus demandait de dessiner un maillage en utilisant un seul buffer pour stocker et manipuler à la fois les données de position et de couleur des sommets. Nous avons donc créé une nouvelle structure MaillageVAO. Cette structure contient un buffer OpenGL et un tableau dynamique de Vec3, servant à stocker positions et couleurs en même temps.

```
struct MaillageVAO {

    GLuint VAO;
    std::vector<Vec3> poscolor;
```

Figure 29

Nous avons choisi d'un peu changer les couleurs des sommets pour obtenir quelque chose de différent.

```
void initMaillageVAO() {

    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, 1.0f, 1.0f),
    };

    poscolor = g_vertex_buffer_data;

}
```

Figure 30: Initialisation du maillage

Nous avons stocké les données de poscolor dans le buffer VAO.

```
void initBuffers() {

    glGenBuffers(1, &VAO);
    glBindBuffer(GL_ARRAY_BUFFER, VAO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(Vec3) * poscolor.size(), &poscolor[0], GL_STATIC_DRAW);

}
```

Figure 31: Initialisation du buffer

Sans oublier la gestion de la suppression du buffer après envoi sur le GPU.

```
void clearBuffers(){
    glDeleteBuffers(1,&VAO);
}
```

Figure 32: Gestion de la suppression du buffer

La plus grosse différence résidait dans la fonction draw. En effet, il faut expliquer à OpenGL la manière dont sont organisées les données dans notre buffer. Pour cela, il a fallu lui préciser qu'il trouvera chaque donnée en alternance, c'est-à-dire ici tous les deux Vec3 (d'où l'offset qui vaut 2*sizeof(Vec3)). De plus, comme la couleur se trouve après la position dans le buffer, il faut dire, au moment du deuxième appel à glVertexAttribPointer, où se trouve le premier vecteur de couleur dans le buffer. Il est décalé d'un Vec3 par rapport au début, donc on commence les couleurs à sizeof(Vec3). Enfin, au moment de dessiner, il ne faut pas oublier de préciser qu'il y a poscolor.size() / 2 sommets à relier.

```
void draw() {

    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);

    glBindBuffer(GL_ARRAY_BUFFER, VAO);

    // Position
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 2 * sizeof(Vec3), (void*)0);

    // Couleur
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 2 * sizeof(Vec3), (void*)(sizeof(Vec3)));

    glDrawArrays(GL_TRIANGLES, 0, poscolor.size() / 2);

    glDisableVertexAttribArray(0);
    glDisableVertexAttribArray(1);

}
```

Figure 33: Code de la fonction draw dans la struct MaillageVAO

Après avoir réalisé toutes ces étapes, voilà ce que nous avons observé.

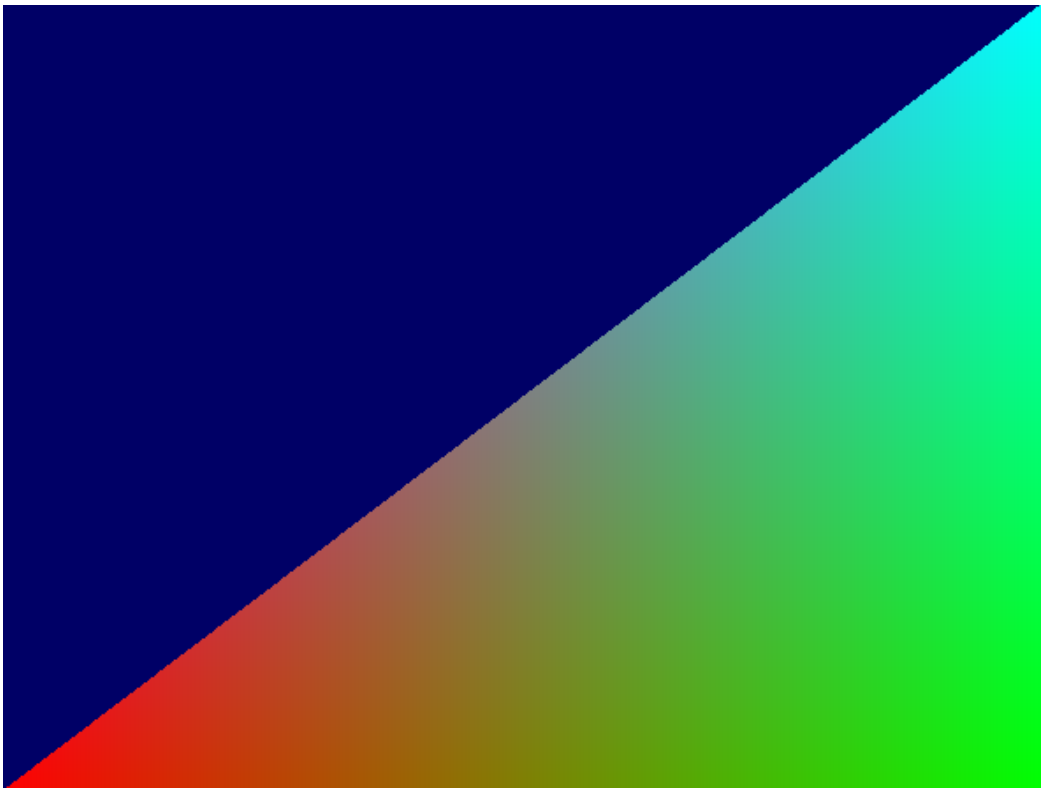

Figure 34: Affichage du triangle grâce à la structure MaillageVAO

6.2 Ajout d'un demi-cercle sur le triangle

Pour cet exercice bonus, nous avons pris l'initiative d'afficher un demi-cercle sur le triangle. À vrai dire, nous voulions afficher un cercle bien centré au milieu du triangle, mais nous n'avons pas réussi et par manque de temps nous nous sommes contentés d'un demi-cercle. Pour réaliser ceci, il a uniquement fallu modifier le code du vertex shader et du fragment shader. L'idée est qu'on passe les coordonnées de chaque sommet depuis le vertex shader vers le fragment shader et on regarde si le sommet est situé dans le rayon du cercle ou non. Si c'est le cas, on le colorie en noir, sinon on laisse la couleur de base.

```
out vec2 FragPos;
```

Figure 35: Ajout d'une variable out dans le vertex shader

```
in vec2 FragPos;
```

Figure 36: Ajout d'une variable in dans le vertex shader

```
void main() {

    vec3 pos = scale * vertexPosition_modelspace + translation;
    //Mettre à jour ce code pour appliquer la translation et la mise à l'échelle
    gl_Position =  vec4(pos,1);
    o_color = colorValues_modelspace;

    FragPos = pos.xy;
}
```

Figure 37: Modification du vertex shader pour récupérer la position de chaque sommet et la transmettre au fragment shader

```
void main()
{

    float dist = length(FragPos); // Calcule la norme du vecteur

    if(dist < 0.5) {
        FragColor = vec4(0.0, 0.0, 0.0, 1.0);
    } else {
        FragColor = vec4(o_color, 1.0);
    }

}
```

Figure 38: Modification du fragment shader pour modifier la couleur de chaque sommet selon leur distance au centre de la scène

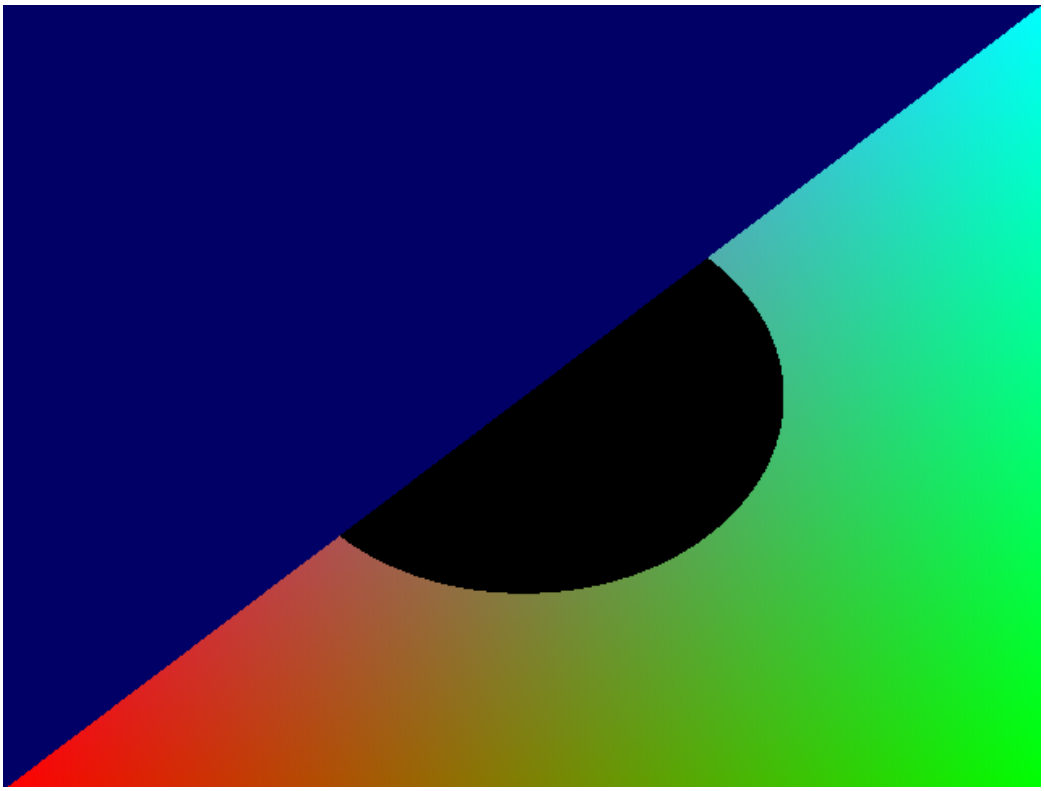

Figure 39: Rendu du triangle avec le demi-cercle noir

Merci pour le temps et l'attention que vous avez consacrés à la lecture de ce compte-rendu.