

# Programmation 3D

## Compte-rendu TP1

Louis Jean  
Master 1 IMAGINE  
Université de Montpellier

21 septembre 2023

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Mise à jour du code</b>	<b>3</b>
2.1	Première étape : sommets . . . . .	3
2.2	Deuxième étape : triangles et premier affichage . . . . .	4
2.3	Troisième étape : normales . . . . .	7
2.4	Quatrième étape : ajout de couleur et affichage final . . . . .	8
<b>3</b>	<b>Compteur de FPS</b>	<b>10</b>
<b>4</b>	<b>Modification en temps réel de la taille du maillage</b>	<b>12</b>
<b>5</b>	<b>Animation</b>	<b>15</b>
<b>6</b>	<b>Bonus : bugs et rendus inattendus</b>	<b>16</b>

# 1 Introduction

Le but de ce TP était de se familiariser avec la programmation OpenGL (avec GLUT) sur GPU, mais sans shaders. Pour cela, il est proposé de réadapter du code OpenGL CPU en code OpenGL GPU. Dans le code qui nous a été donné, il y avait une struct Mesh implémentée "à l'ancienne" (comprenez qui était optimisée pour OpenGL 1.x) et notre but était de la moderniser pour pouvoir envoyer les données du maillage directement sur le GPU. Le maillage était ouvert dans la fonction main, et ses informations stockées dans les attributs de la struct Mesh. À nous d'en transférer le contenu pour l'emmener sur le GPU.

```
struct Mesh {  
    std::vector<Vec3> vertices;  
    std::vector<Vec3> normals;  
    std::vector<Triangle> triangles;  
}
```

Figure 1: struct Mesh donnée dans le code de base

## 2 Mise à jour du code

### 2.1 Première étape : sommets

Le premier exercice consistait à mettre à jour la manière d'implémenter les sommets. Pour ce faire, il fallait rajouter un attribut `vertexArray` à la struct, et copier le contenu de `vertices` à l'intérieur, grâce à une fonction `buildVertexArray`. Enfin, on pouvait, grâce à la fonction `glVertexPointer` envoyer le contenu de `vertexArray` directement sur le GPU, pour être bien plus efficace.

```
std::vector<float> vertexArray;
```

Figure 2: Ajout de l'attribut `vertexArray` dans la struct `Mesh`

```
void buildVertexArray() {  
    // Passer les données de "vertices" à "vertexArray"  
    this->vertexArray.resize(3*this->vertices.size());  
    for(int i = 0; i < (int)this->vertices.size(); i++) {  
        this->vertexArray[3*i] = this->vertices[i][0];  
        this->vertexArray[3*i+1] = this->vertices[i][1];  
        this->vertexArray[3*i+2] = this->vertices[i][2];  
    }  
    glVertexPointer(3, GL_FLOAT, 3*sizeof(float), (GLvoid*)&this->vertexArray[0]);  
}
```

Figure 3: Fonction `buildVertexArray`

## 2.2 Deuxième étape : triangles et premier affichage

D'une manière tout à fait similaire, j'ai pu copier la totalité des informations contenues dans `triangle` vers un nouvel attribut `triangleArray`, c'est-à-dire les indices des sommets qui sont reliés pour faire des triangles.

```
std::vector<unsigned int> triangleArray;
```

Figure 4: Ajout de l'attribut `vertexArray` dans la struct `Mesh`

```
void buildTriangleArray() {  
    // Passer les données de "triangles" à "triangleArray"  
    this->triangleArray.resize(3*this->triangles.size());  
    for(int i = 0; i < (int)this->triangles.size(); i++) {  
        this->triangleArray[3*i] = this->triangles[i][0];  
        this->triangleArray[3*i+1] = this->triangles[i][1];  
        this->triangleArray[3*i+2] = this->triangles[i][2];  
    }  
}
```

Figure 5: Fonction `buildTriangleArray`

Passé cette étape, j'ai voulu apercevoir le fruit de ces changements. Il a donc aussi fallu modifier la fonction drawTriangleMesh. Terminé le gros bloc de code, il suffit désormais d'utiliser la fonction glDrawElements pour que le GPU se charge d'afficher les triangles à l'écran.

```
void drawTriangleMesh( Mesh const & i_mesh ) {  
    glDrawElements(GL_TRIANGLES, i_mesh.triangleArray.size(), GL_UNSIGNED_INT, (GLvoid*)&i_mesh.triangleArray[0]);  
}
```

Figure 6: Fonction drawTriangleMesh

Il fallait aussi expliquer à OpenGL que je souhaitais dessiner des points et des triangles en utilisant ce que j'avais envoyé sur le GPU, en activant des états bien définis (car OpenGL fonctionne comme une machine à états). J'ai mis cela en place dans la fonction init déjà donnée.

```
glEnableClientState(GL_VERTEX_ARRAY);
```

Figure 7: Activation de l'état pour utiliser le tableau de sommets envoyé sur le GPU

Il fallait aussi appeler les fonctions (sur le maillage chargé) que je venais de créer dans le main du programme.

```
mesh.buildVertexArray();  
mesh.buildTriangleArray();
```

Figure 8: Appels aux fonctions buildVertexArray et buildTriangleArray dans le main du programme

Voici le résultat.



Figure 9: Affichage des points et triangles du maillage en utilisant la programmation GPU

## 2.3 Troisième étape : normales

Ici encore, j'ai utilisé le même procédé pour copier les normales dans un nouvel attribut normalArray. Il fallait encore une fois envoyer le tableau de normales nouvellement créé vers le GPU, cette fois-ci en utilisant la fonction glNormalPointer.

```
std::vector<float> normalArray;
```

Figure 10: Ajout de l'attribut normalArray dans la struct Mesh

```
void buildNormalArray() {  
    // Passer les données de "normals" à "normalArray"  
    this->normalArray.resize(3*this->normals.size());  
    for(int i = 0; i < (int)this->normals.size(); i++) {  
        this->normalArray[3*i] = this->normals[i][0];  
        this->normalArray[3*i+1] = this->normals[i][1];  
        this->normalArray[3*i+2] = this->normals[i][2];  
    }  
    glNormalPointer(GL_FLOAT, 3*sizeof(float), (GLvoid*)&this->normalArray[0]);  
}
```

Figure 11: Fonction buildNormalArray

```
glEnableClientState(GL_NORMAL_ARRAY);
```

Figure 12: Activation de l'état pour utiliser le tableau de normales envoyé sur le GPU

```
mesh.buildNormalArray();
```

Figure 13: Appel à la fonction buildNormalArray dans le main du programme

## 2.4 Quatrième étape : ajout de couleur et affichage final

Pour ajouter de la couleur, il a fallu choisir arbitrairement des couleurs à associer à chaque sommet, car l'ancienne structure Mesh ne stockait pas de couleurs.

J'ai choisi d'utiliser les positions des sommets pour définir les couleurs. Autrement, la méthode est la même que pour les autres attributs.

```
std::vector<float> colorArray;
```

Figure 14: Ajout de l'attribut colorArray dans la struct Mesh

```
void buildColorArray() {  
    // Ajouter des couleurs (pas de transfert de données car pas d'ancien attribut couleur)  
    // On va utiliser les positions des sommets pour définir les couleurs  
    this->colorArray.resize(this->vertexArray.size());  
    for(int i = 0; i < (int)this->vertexArray.size(); i++) {  
        this->colorArray[i] = this->vertexArray[i];  
    }  
    glColorPointer(3, GL_FLOAT, 3*sizeof(float), (GLvoid*)&this->colorArray[0]);  
}
```

Figure 15: Fonction buildColorArray

```
glEnableClientState(GL_COLOR_ARRAY);  
glEnable(GL_COLOR_MATERIAL);
```

Figure 16: Activation des états pour utiliser le tableau de couleurs envoyé sur le GPU

```
mesh.buildColorArray();
```

Figure 17: Appel à la fonction buildColorArray dans le main du programme



Après ces étapes, voici le rendu.



Figure 18: Affichage des points, des triangles, des normales et des couleurs du maillage en utilisant la programmation GPU

### 3 Compteur de FPS

Avec le temps restant, j'ai pu implémenter un simple compteur de FPS (Frames Per Seconds) qui imprime le nombre d'images par seconde dans la console. À vrai dire, ce compteur affiche le nombre de FPS toutes les demi-secondes, en multipliant le nombre de FPS contenus dans une demi-seconde par 2.

En créant trois variables globales, il m'a été possible de modifier la fonction display (qui est appelée à chaque frame) afin de calculer le nombre d'images par seconde.

```
double lastTime = 0.0;
int frameCount = 0;
int fps = 0;
```

Figure 19: Variables globales nécessaires pour le compteur

```
double current_time = glutGet(GLUT_ELAPSED_TIME) / 1000.0;
double deltaTime = current_time - lastTime;
frameCount++;
if(deltaTime >= 0.5) {
    fps = frameCount * 2;
    frameCount = 0;
    lastTime = current_time;
    std::cout<<"FPS : "<<fps<<std::endl;
}
```

Figure 20: Modification de la fonction display

```
FPS : 240
FPS : 240
FPS : 240
FPS : 240
FPS : 240
FPS : 242
FPS : 240
FPS : 240
FPS : 240
FPS : 238
FPS : 240
FPS : 236
```


Figure 21: Affichage des fps dans la console lors de l'exécution du programme

## 4 Modification en temps réel de la taille du maillage

Par la suite, j'ai choisi d'ajouter la possibilité de modifier en temps réel la taille du maillage couramment chargé.

En appuyant sur la touche +, le maillage grandit, et il diminue lors de l'appui sur la touche -.

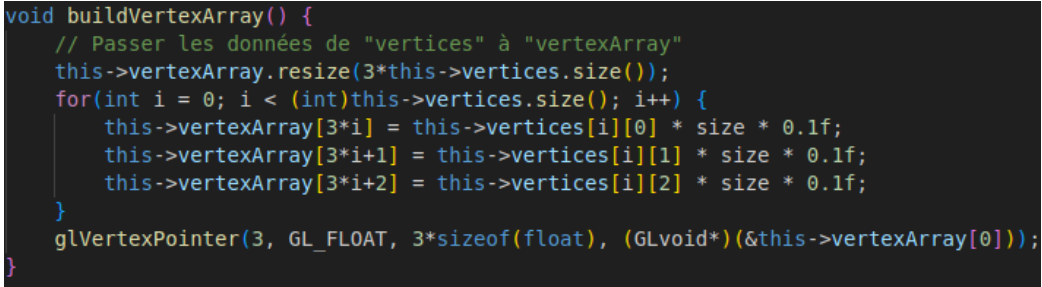
Je suis parvenu à implémenter cette fonctionnalité grâce à une petite extension de la fonction key, déjà présente dans le code donné, et une modification de la fonction buildVertexArray.



```
int size = 10;
```

Figure 22: Déclaration de la variable globale nécessaire pour cette fonctionnalité

J'ai multiplié les coordonnées des sommets par  $\text{size} * 0.1f$  pour avoir une augmentation/diminution correcte et pas trop abrupte. Cela a pour effet d'écartier ou de rapprocher les sommets dans l'espace, rendant donc les triangles qui les relient plus grands ou plus petits, respectivement.



```
void buildVertexArray() {  
    // Passer les données de "vertices" à "vertexArray"  
    this->vertexArray.resize(3*this->vertices.size());  
    for(int i = 0; i < (int)this->vertices.size(); i++) {  
        this->vertexArray[3*i] = this->vertices[i][0] * size * 0.1f;  
        this->vertexArray[3*i+1] = this->vertices[i][1] * size * 0.1f;  
        this->vertexArray[3*i+2] = this->vertices[i][2] * size * 0.1f;  
    }  
    glVertexPointer(3, GL_FLOAT, 3*sizeof(float), (GLvoid*)&this->vertexArray[0]);  
}
```

Figure 23: Modification de la fonction buildVertexArray

Selon la touche appuyée, je mets à jour size et rappelle la fonction buildVertexArray pour actualiser les positions des sommets.

```
case '+':  
    size++;  
    mesh.buildVertexArray();  
    break;  
  
case '-':  
    size--;  
    mesh.buildVertexArray();  
    break;
```

Figure 24: Extension du switch dans la fonction key

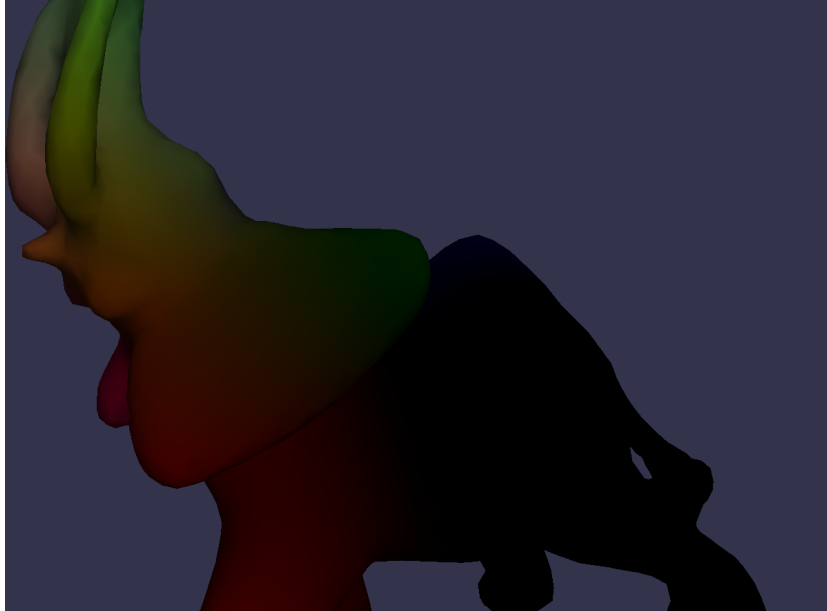


Figure 25: Maillage de l'éléphant grossi

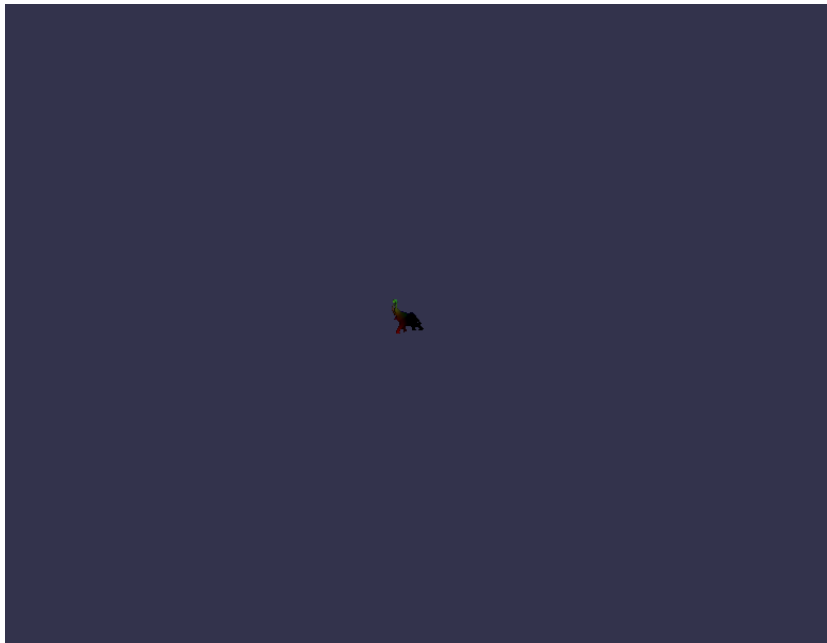


Figure 26: Maillage de l'éléphant rapetissé

## 5 Animation

Enfin, j'ai pris la liberté d'animer le maillage. Pour y parvenir, j'ai réutilisé la variable `lastTime`, créée à la base pour le compteur de FPS, car elle est déjà actualisée à chaque demi-seconde. En ajoutant du code dans la fonction `drawTriangleMesh`, j'ai donc pu faire bouger les sommets du maillage en fonction du temps.

Ici j'ai choisi d'utiliser le cosinus pour effectuer ma translation, mais on pourrait utiliser tout un tas d'autres fonctions pour aboutir à des résultats plus intéressants (par exemple cosinus et sinus simultanément pour décrire une translation sur un cercle).

```
for(int i = 0; i < (int)i_mesh.vertices.size(); i++) {  
    mesh.vertexArray[3*i] += cos(lastTime)/200;  
}
```

Figure 27: Bout de code rajouté dans la fonction `drawTriangleMesh`

Le maillage de l'éléphant suit la droite rouge en faisant des allers-retours.

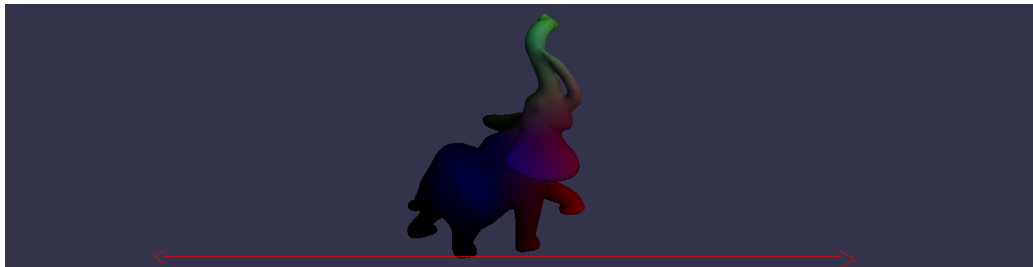


Figure 28: Représentation de l'animation du maillage

## 6 Bonus : bugs et rendus innatendus

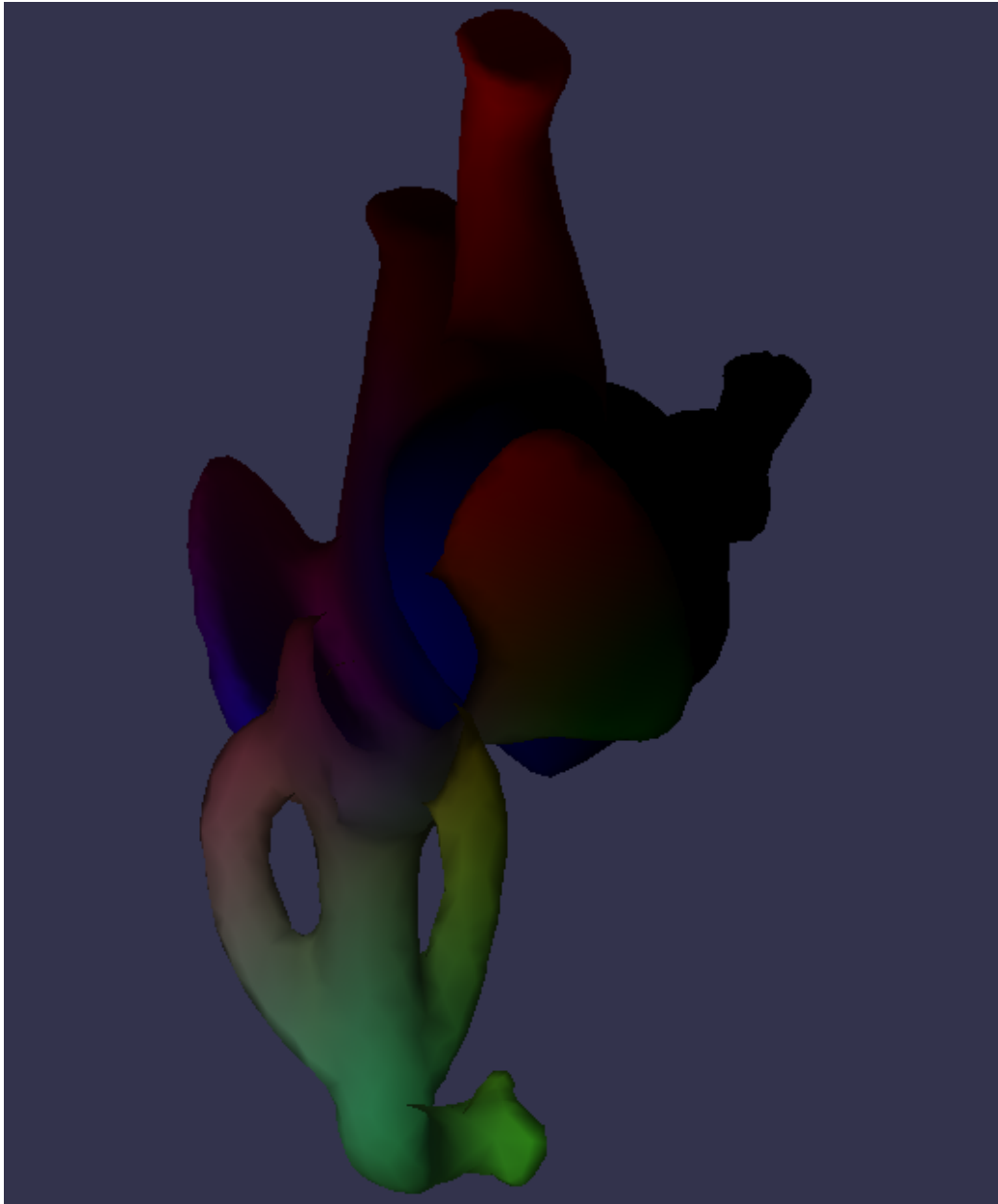


Figure 29: Éléphant de Moebius (mais où se situe la frontière entre l'intérieur et l'extérieur ?!)





Figure 30: En exclusivité pour vous, l'éléph'aplati !

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