# 4

# Premiers programmes OpenGL

Dans ce chapitre, je vais vous présenter les aspects *bas niveau* d'OpenGL qu'il faut connaître pour écrire tout programme graphique. Le but n'est pas encore de faire de belles images, mais seulement de découvrir deux mécanismes essentiels d'OpenGL : les *buffers* et les *shaders*.

## 4.1. Dessin en 2D

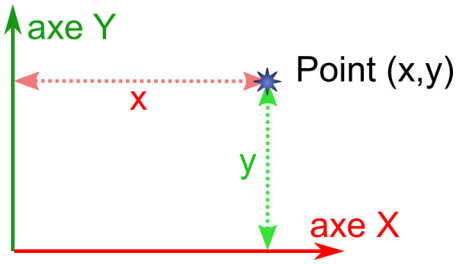
Pour commencer, je vais vous montrer comment dessiner seulement en deux dimensions avec OpenGL. C'est-à-dire que ce seront des dessins faits à plat. C'est plus simple à comprendre pour un début. Ce qui concerne la profondeur pour faire de la 3D sera expliqué à la section suivante.

On peut objecter qu'OpenGL n'est pas l'outil le plus adapté pour la 2D. Dessiner des figures 2D très complexes est beaucoup plus aisé avec l'API Cairo par exemple. L'intérêt ici est de comprendre progressivement comment OpenGL dessine, comment fonctionne le pipeline graphique présenté au chapitre Principes généraux.

### Dessiner un simple triangle

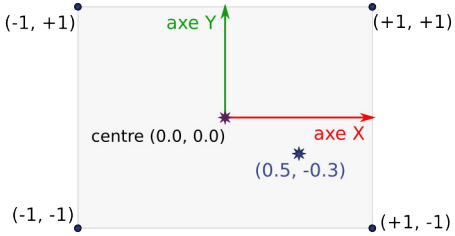
Pour dessiner un triangle, il faut fournir trois points : les trois sommets du triangle. Ces points sont spécifiés par leurs coordonnées. Les coordonnées représentent d'une certaine manière la distance entre le point qu'elles définissent et le coin inférieur gauche de la zone de dessin. La Figure 4.1 illustre cette idée.

***Figure 4.1 :*** *Coordonnées 2D*



En général, dans de nombreuses API de dessin, les coordonnées sont mesurées en nombre de pixels. Par exemple, pour une fenêtre de taille (320,240), les coordonnées varient respectivement de 0 à 319 et de 0 à 239. Ce n'est pas le cas en OpenGL, les coordonnées d'un pixel écran sont des nombres réels qui doivent être dans l'intervalle [−1, 1]. C'est-à-dire que le centre de l'image est en (0,0) et le coin haut-droit de l'image est en (+1,+1). Ce système de coordonnées s'appelle l'*espace normalisé écran* (*normalized device space* en anglais), abrégé en NDC. La Figure 4.2 résume la situation en OpenGL.

***Figure 4.2 :*** *Coordonnées 2D OpenGL*




Ce schéma vous surprend peut-être : des coordonnées toutes deux comprises entre −1 et 1 devraient donner un contour bien carré au lieu de ce rectangle un peu aplati. En fait, les coordonnées OpenGL sont normalisées par rapport à la taille de l'écran. Dans tous les cas, le côté gauche de la vue OpenGL est associé à $x = -1$ et le haut de la vue est en $y = +1$ même si l'écran est en format 16/9$^{e}$ et quelle que soit sa définition : qu'il soit HD paysage en 1920 × 1280 ou SD portrait en 640 × 800.

Comment faire alors pour dessiner un vrai carré dans ces conditions ? Il vous faudra impérativement des informations sur les proportions réelles de la vue OpenGL : soit la largeur et la hauteur, soit le rapport entre les deux qu'on appelle *aspect ratio* en anglais. Et en plus, ce n'est pas forcément suffisant, certains écrans ont des pixels rectangulaires, c'est-à-dire que sur ces écrans, une image 100 × 100 n'est pas carrée... Cela semble alors ingérable, mais heureusement OpenGL fournit des fonctions pour dessiner des figures correctement proportionnées, nous les verrons à la section Projection caméra qui explique comment obtenir la vue d'une caméra virtuelle.

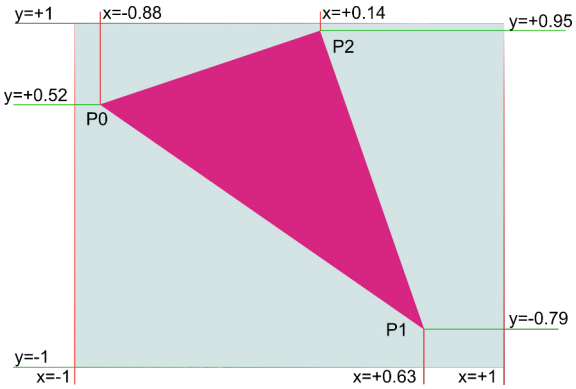
Autre point à considérer : si vous voulez que le triangle soit visible, vous devez fournir des coordonnées dans la plage [−1, +1]. Avec des coordonnées sortant de cet intervalle, le triangle sera partiellement visible, voire invisible. C'est un souci constant pour mettre au point les programmes en OpenGL : il arrive que les objets qu'on dessine soient

invisibles, non pas à cause d'un mauvais algorithme de dessin, mais simplement à cause des coordonnées effectives des points, par exemple suite à des déplacements incorrects ou imprévus des objets. Cette plage [−1, +1] est une sorte de champ de vision – OpenGL ne dessine que ce qui est dedans.

Je vous propose de dessiner le triangle de la Figure 4.3 .

***Note*** *› Les sources complètes de cet exemple, dans ses versions C++, Android et WebGL sont disponibles sur GitHub dans le dossier du projet. Tous les exemples de ce livre sont ainsi disponibles afin que vous puissiez les essayer et les modifier à volonté.*

***Figure 4.3 :*** *Triangle à dessiner*




Ses sommets sont P0(−0.88, +0.52), P1(+0.63, −0.79) et P2(+0.14, +0.95). On aurait pu les nommer A, B et C comme habituellement en mathématiques, et aussi donner un autre ordre pour les points, soit P1(+0.63, −0.79) comme premier point, P2(+0.14, +0.95) comme deuxième et P0(−0.88, +0.52) comme dernier ou encore P2(+0.14, +0.95) comme premier point, P1(+0.63, −0.79) comme deuxième et P0(−0.88, +0.52) comme dernier.

Passons maintenant à la mise en œuvre. Il y a trois étapes : 1) définition des coordonnées du triangle dans un Vertex Buffer, 2) création de shaders pour le dessiner et 3) dessin.

## Étape 1 : Définition de la structure du triangle

La première chose à faire est de créer un tableau de `float` contenant les coordonnées. C'est à peine différent d'une plate-forme à l'autre. Dans tous les cas, on trouve l'énumération des nombres réels, $x$ et $y$ pour chaque point, dans l'ordre de parcours du triangle.

- En C++ :

```
std::vector<GLfloat> vertices {
    -0.88f, +0.52f,     // P0
    +0.63f, -0.79f,     // P1
    +0.14f, +0.95f,     // P2
};
```

- En Java sur Android :

```
float vertices[] = {
    -0.88f, +0.52f,     // P0
    +0.63f, -0.79f,     // P1
    +0.14f, +0.95f,     // P2
};
```

- En JavaScript :

```
let vertices = [
    -0.88, +0.52,       // P0
    +0.63, -0.79,       // P1
    +0.14, +0.95,       // P2
];
```

On peut remarquer que ces nombres sont à la suite, sans séparation entre les sommets. On peut craindre les effets de l'oubli d'un nombre, ou pire : le remplacement involontaire d'un point `.` par une virgule `,` . Cela échangerait toutes les coordonnées : $y$ deviendrait $x$ à partir d'un certain point. C'est un risque très sérieux ; c'est pour le réduire que j'ai indenté les valeurs et je vous conseille d'en faire autant. Vous remarquerez que j'ai aussi ajouté tous les signes. C'est pour avoir les données bien alignées, ce qui permet d'être sûr de ne pas avoir oublié quelque chose – on a déjà assez de soucis avec les valeurs à donner aux coordonnées.

Ensuite, il faut envoyer cette énumération à OpenGL qui va la stocker sur la carte graphique dans le Vertex Buffer Object (VBO). Cela fait appel à plusieurs fonctions OpenGL assez compliquées. Je les ai regroupées dans une fonction appelée `makeFloatVBO` dans la bibliothèque `Utils` que vous avez téléchargée à la section Bibliothèques de fonctions supplémentaires. Voici son code source en JavaScript, c'est le plus simple à lire. En C++ et Java, c'est la fonction `glGenBuffers` qui est appelée au lieu de `gl.createBuffer` (voir Exemple 5.29 pour la version C++ de `makeFloatVBO`).

***Note*** *› La fonction `gl.createBuffer` appelle `glGenBuffers` en interne, mais au lieu de retourner un simple identifiant comme en Java et C++, elle retourne un objet JavaScript, voir les explications plus loin.*

***Exemple 4.1 :*** *Fonction Utils.makeFloatVBO*

```
function makeFloatVBO(values, vbo_type, usage)
{
    let buffer = gl.createBuffer(); ❶
    gl.bindBuffer(vbo_type, buffer); ❷
    gl.bufferData(vbo_type, new Float32Array(values), usage); ❸
    gl.bindBuffer(vbo_type, null); ❹
    return buffer;
}
```

❶ Cette instruction demande à OpenGL de créer un VBO.
❷ Activation du VBO : toutes les fonctions suivantes s'adresseront à ce VBO.
❸ Fourniture du tableau au VBO : les données sont transférées sur la carte graphique.
❹ Désactivation du VBO.

Avec cette fonction, il suffit de faire ceci pour créer un VBO :

- En C++ :

```
m_VertexBufferId = Utils::makeFloatVBO(vertices,
    GL_ARRAY_BUFFER, GL_STATIC_DRAW);
```

- En Android :

```
m_VertexBufferId = Utils.makeFloatVBO(vertices,
    GL_ARRAY_BUFFER, GL_STATIC_DRAW);
```

- En JavaScript :

```
this.m_VertexBufferId = Utils.makeFloatVBO(vertices,
    gl.ARRAY_BUFFER, gl.STATIC_DRAW);
```

Ces instructions affectent la variable membre `m_VertexBufferId` de la classe Triangle. Cette variable est de type entier en C++ et sur Android. OpenGL gère en effet de nombreuses structures de données internes : shaders, buffers, textures, etc. à l'aide d'identifiants. Ce sont des nombres entiers simples : 1,2,3... Pour certaines ressources, l'identifiant est strictement positif (shaders, textures, VBO), pour d'autres il peut être nul (emplacement de variable). Pour utiliser une ressource, par exemple un VBO, vous devez fournir son identifiant à la fonction `glBindBuffer` et pour cesser d'utiliser cette ressource, vous fournissez l'identifiant 0 à cette même fonction. C'est ce que j'ai fait dans la fonction `makeFloatVBO`.

En WebGL, les identifiants ne sont pas des entiers, mais des objets. C'est à cause du chargement asynchrone des documents, par des *XMLHttpRequest*. C'est un mécanisme pour recevoir des données comme des images ou des fichiers, après chargement de la page, sans bloquer le navigateur. Ce mécanisme est utilisé pour charger les textures, les modèles 3D, et même les sources des shaders si on veut. WebGL permet de ne pas bloquer le navigateur en créant un objet JavaScript qui se complétera de lui-même quand le document sera reçu du serveur. Pour désactiver une ressource, vous devez fournir la constante `null` et non pas 0.

Il ne faut employer que les valeurs retournées par OpenGL et ne jamais faire de suppositions sur les prochaines valeurs. Lorsqu'on se trompe d'identifiant, OpenGL positionne un code d'erreur, qui est le plus fréquemment `GL_INVALID_ENUM`. Des fonctions permettent d'interroger ces codes et de savoir ce qui ne va pas. Je vous invite à consulter l'annexe Mise au point d'un logiciel OpenGL pour avoir toutes les explications concernant la mise au point de programmes.

Dans l'exemple, les coordonnées ne changent pas au cours du temps. C'est indiqué par la valeur `GL_STATIC_DRAW` fournie en paramètre à `makeFloatVBO`. Si le triangle devait changer de forme, nous serions amenés à modifier le buffer dans la fonction de dessin `onDraw()`. Il y a plusieurs techniques qui sont expliquées au chapitre Animation d'un maillage.

L'autre paramètre, `GL_ARRAY_BUFFER`, indique le type de VBO qu'on souhaite créer. Un *Array Buffer* est un tableau de valeurs numériques. Ici ce sont des coordonnées, et dans le prochain programme ce seront des couleurs. L'autre catégorie de VBO qui existe est `GL_ELEMENT_ARRAY_BUFFER`. Les valeurs qu'ils contiennent sont des entiers qui servent d'indices dans d'autres VBO. La section Structure du tétraèdre montre comment ils fonctionnent.

À l'issue de cette étape, les coordonnées du triangle sont stockées sur la carte graphique, et on peut libérer le tableau de `float` qui a servi à la création du VBO. Ici, c'est fait automatiquement à la sortie du constructeur, puisque c'est une variable locale.

### Étape 2 : Définition des shaders

La seconde étape du travail consiste à écrire un programme de shaders. C'est ce programme qui effectue les dessins. Il est constitué de deux parties : un programme qui gère les sommets (coins des triangles) et un programme qui colore les pixels. Ces deux parties sont liées car nous verrons plus loin que le Vertex Shader peut envoyer des informations au Fragment Shader. Dans la suite, j'écrirai simplement "shader" à la place de "programme de shaders" et cela désignera le couple Vertex Shader, Fragment Shader.

Dans notre cas, les shaders sont très simples. Ils sont écrits dans un langage de programmation spécifique d'OpenGL qui s'appelle GLSL : *OpenGL Shading Language*. Sa syntaxe ressemble à celles des langages C, Java et JavaScript, mais en beaucoup plus simple car c'est un langage destiné à faire des calculs sur des points, des vecteurs et des matrices. Ces programmes sont mis en place sur la carte graphique par un compilateur intégré à OpenGL.

### Vertex Shader

Le premier shader est le Vertex Shader. Son rôle général est de calculer les coordonnées écran des sommets qu'on lui envoie. Dans cet exemple, les coordonnées écran sont déjà calculées et elles sont placées dans le VBO `m_VertexBufferId`. Le Vertex Shader n'a donc qu'à les recopier en sortie.

Ces coordonnées du VBO arrivent par l'intermédiaire d'une variable appelée `glVertex`[1]. Comme le VBO contient des couples $(x,y)$, et qu'il faut générer des coordonnées à quatre composantes, nous verrons pourquoi ultérieurement, le Vertex Shader doit rajouter les composantes manquantes : 0 et 1. Ces coordonnées en sortie doivent être mises dans une variable globale spéciale appelée `gl_Position`. Cela donne la source suivante.

```
#version 300 es ❶
in ❷ vec2 ❸ glVertex;
void main(void)
{
    gl_Position ❹ = vec4(glVertex ❺, 0.0, 1.0) ❻;
}
```

❶ Le mot clé `#version` suivi d'un nombre indique quelle version de GLSL on souhaite employer, à écrire sans point décimal. Pour ce livre, j'ai choisi la version 3.00 pour WebGL et C++ et la 3.10 pour Android pour être certain que les exemples puissent fonctionner partout. Voir l'annexe Différences entre les versions d'OpenGL pour plus de détails sur ces différences. Elles sont tout à fait minimes au regard de l'objectif de ce livre.

❷ Le mot clé `in` définit une variable liée à un Vertex Buffer Object (VBO). C'est-à-dire que la variable `glVertex` contiendra à tour de rôle les coordonnées de l'un des sommets du triangle à dessiner (revoir la Figure 2.4). Une telle variable est nommée *variable attribute*.

[1] On aurait pu nommer cette variable autrement. C'est seulement une habitude résultant des versions précédentes d'OpenGL.

À noter que je ne peux pas utiliser le préfixe `gl_` pour nommer mes propres variables, par exemple `gl_Vertex`, car ce préfixe est réservé pour les variables prédéfinies de GLSL.

❸ Le type `vec2` est un couple de `float` $(x,y)$. Il n'y a que deux coordonnées car nous avons défini le VBO des vertices avec deux coordonnées $x$ et $y$.

❹ La variable `gl_Position` est prédéfinie dans GLSL 3.10 ES et elle est de type `vec4 (x,y,z,w)`. C'est la sortie du shader.

❺ Comme la variable `glVertex` est de type `vec2` pour correspondre à ce qu'on a mis dans le VBO, on ne peut pas l'affecter directement à `gl_Position` qui est du type `vec4`, il faut spécifier la valeur des deux champs, $z$ et $w$ manquants. GLSL ne tolère pas les conversions implicites (*type casting* en anglais).

❻ La notation `vec4(a,b,c,d)` crée un quadruplet de `float`. Ça peut être une couleur, une position, une direction ou autre chose. Ici, on définit la position `(x,y,z,w)` du vertex. J'expliquerai pourquoi il y a quatre coordonnées au chapitre Transformations géométriques. Ici, il suffit de savoir que la quatrième doit toujours valoir 1.

### Fragment Shader

Voici maintenant le Fragment Shader : on lui fait appliquer une sorte de rouge fuschia. Cette teinte est représentée par un mélange de rouge, de vert et de bleu, défini par trois coefficients d'intensité compris entre 0 et 1. Pour déterminer ces coefficients, j'ai utilisé un nuancier dans un logiciel de dessin ou de retouche d'image quelconque. Mais, en général, ces logiciels affichent les composantes sous forme de nombres dans l'intervalle [0, 255], par exemple (214, 38, 130). Il suffit de les diviser par 255 pour tomber dans l'intervalle [0, +1] qu'attend OpenGL.

```
#version 300 es ❶
precision mediump float; ❷
out vec4 glFragColor; ❸
void main()
{
    glFragColor = vec4(0.84, 0.15, 0.51, 1.0) ❹;
}
```

❶ Le numéro de version doit être exactement le même entre le Vertex Shader et le Fragment Shader.

❷ Cette directive est nécessaire quand il y a une variable en entrée ou en sortie du Fragment Shader. Elle spécifie l'amplitude des nombres réels par l'un des mot clés : `highp`, `mediump` et `lowp`. Avec `highp` les réels peuvent aller de $-2^{62}$ à