Travaux dirigés OpenGL 3+ n°4

Cours de Synthèse d'Images

—IMAC 2—

3D

Ce TD a pour objectif de vous apprendre à "dessiner" en 3D avec OpenGL. Nous verrons ainsi les différents espaces de coordonnées, les transformations 3D et comment les appliquer aux objets via le Vertex Shader.

▲ Important :

Un nouveau template est disponible sur le site web. Celui ci contient en plus du code de base une bibliothèque mathématique appelée `glm` permettant la construction facile de matrices. En plus de cela plusieurs classes sont fournis pour construire les vertex de formes 3D usuelles comme le cube, la sphère, le cylindre ou le cone. Des fragments shaders qui seront utilisés pour le TD sont également disponibles dans le dossier "shaders". Durant ce TD vous n'aurez que le Vertex Shader à coder en plus du code de l'application.

▲ Cours : Les différents espaces et le passage de la 3D à la 2D

L'écran de votre ordinateur est un univers à 2 dimensions. Pour transformer une scène composé d'objets à 3 dimensions, il faut passer par une étape de *projection*. Ainsi on se donne une caméra placée dans la scène et on projette l'ensemble des triangles constituant la scène sur le plan 2D de la caméra. On ne conserve que les points visibles (si deux éléments de surface 3D sont projetés sur le même pixel, on affiche celui qui à la plus petite profondeur selon le point de vue de la caméra). Afin de faciliter tous ces calculs et le positionnement des éléments, on utilise différents espaces de coordonnées décrit ci-dessous.

Espace *Model*

Il existe un espace Model **pour chaque objet de la scène** (non pas chaque triangle, mais chaque groupe de triangles sensé constituer un seul et même objet, par exemple une sphère, un personnage ou encore un mur). Dans cet espace les coordonnées des vertex sont exprimées localement. Si par exemple on manipule une sphère, on pourra choisir le centre de la sphère comme origine de l'espace Model. En pratique ce choix n'est pas fait par le programmeur mais

par l'artiste qui modélise l'objet avec un logiciel spécialisé (3DS max ou Maya par exemple). Si l'artiste a décidé de modéliser un personnage avec comme origine le milieu des pieds du personnage, le programmeur doit s'arranger pour que cette origine coïncide avec le terrain afin que le modèle ne passe pas à travers le décors ! Au contraire si l'origine est placé sur la tête le programmeur doit prendre en compte la taille des personnages pour les placer.

L'utilisation de cet espace de coordonnées est primordial : dans l'espace local, les coordonnées des points ne changent jamais, quelque soit leur position dans la scène. Ainsi on peut les envoyer à la carte graphique dans un VBO et ne plus jamais y toucher. Le placement de l'objet dans la scène se fera alors en multipliant les coordonnées des points par la matrice Model notée M au sein du Vertex Shader. Si C_{Model} représente les coordonnées du point dans l'espace Model et C_{World} représente les coordonnées du meme point dans la scène, on à la relation suivante :

$$C_{World} = M \times C_{Model}$$

(attention à l'ordre de multiplication).

Mathématiquement, les colonnes de la matrice M sont les coordonnées du repère de l'objet exprimées dans la scène. La multiplication consiste à faire un changement de repère. En pratique on utilise peu cette propriété car des fonctions existantes nous permette de construire les matrices de transformation à partir de multiplications de trois matrices simples : translation, rotation et homothétie (changement d'échelle).

Espace *World*

Cet espace est en quelque sorte unique : il représente l'ensemble de la scène. Lorsque tous les points sont exprimés dans cet espace, on peut faire des calculs de distance entre eux et obtenir un résultat cohérent (en effet, on ne peut pas calculer la distance entre deux points de deux objets différents en utilisant leurs coordonnées locales, ça n'aurait aucun sens !). Bien qu'il soit importance, l'espace World est peu utilisé en OpenGL : ce qui nous intéresse, c'est de projeter la scène selon la caméra, on utilisera donc plutôt l'espace de la caméra, ou *View*.

Espace *View*

L'origine de cet espace est le centre de la caméra. Elle regarde dans la direction $-z$ de son repère local (convention OpenGL). Ainsi, un point (x, y, z) exprimé dans l'espace View avec z positif ne sera pas visible. Tout comme pour la matrice model M, on utilisera des fonctions pour construire la matrice view V et on n'aura donc pas à s'occuper des axes : la fonction orientera le repère de manière à regarder dans la direction négative des z. Afin de passer des coordonnées C_{World} aux coordonnées C_{view} exprimées dans l'espace view, on effectue l'opération suivante :

$$C_{View} = V \times C_{World}$$

Mais comme nous l'avons dit plus haut, les coordonnées dans l'espace World sont rarement utilisées. On remplace dont C_{World} par son expression à partir des coordonnées locales C_{Model} et on a :

$$\begin{aligned} C_{View} &= V \times M \times C_{Model} \\ &= ModelView \times C_{Model} \end{aligned}$$

La matrice $V \times M$ est appelée matrice *ModelView* (remarquez qu'on l'appelle ModelView bien que la multiplication soit faite dans l'ordre View fois Model).

A noter également qu'**il existe une matrice *ModelView* par objet de la scène** : bien que la matrice V reste la même tant que la caméra ne bouge pas, la matrice M est différente pour chaque objet, donc la multiplication des deux est aussi différente.

En OpenGL 2 on utilisait la matrice `GL_MODELVIEW`, elle correspond exactement à cette matrice. La grosse différence est qu'a présent nous devons explicitement la déclarer dans un shader comme une variable uniforme et l'envoyer manuellement depuis l'application.

Espace *Clip*

Cet espace est un peu particulier : il représente les coordonnées projetés des points dans un espace 3D projectif. La transformation convertissant les coordonnées view aux coordonnées clip n'est pas de nature affine (basiquement une transformation affine ne fait que des translation, rotation, changement d'échelle et symétries ; des trucs assez intuitif en fait). Cette transformation effectue potentiellement une distorsion sur l'espace : elle transforme le *frustum* (pyramide tronquée) vue par la caméra en un cube existant dans l'espace projectif. A nouveau en pratique on s'en fiche totalement, les fonctions s'occupe de construire la matrice pour nous. La matrice permettant de passer dans l'espace clip est appelée matrice de projection et est notée P. On a la relation suivante :

$$\begin{aligned} C_{Clip} &= P \times C_{View} \\ &= P \times V \times M \times C_{Model} \end{aligned}$$

La matrice $P \times V \times M$ est appelée matrice *ModelViewProjection*. Elle effectue la projection complète d'un point donné initialement à OpenGL. Les deux types de projection les plus utilisés sont la projection orthogonale (qui consiste juste à "oublier" la coordonnée z) et la projection perspective (on voit alors comme un humain, les points alignées sur une même ligne passant par l'origine sont projetés sur le même pixel et la taille des objets diminue lorsqu'ils sont loin).

Espace des *Normalized Device Coordinates* (NDC)

Cette transformation est effectué par la carte graphique directement. Vous n'avez donc pas à la faire vous même, mais la comprendre ne fait pas de mal !

Pour que la matrice P fonctionne, il est nécessaire de travailler avec des coordonnées homogène (x, y, z, w) (comme pour la translation). L'énorme différence avec une matrice de translation est la suivante : après application de la matrice P, la dernière coordonnée w ne vaut plus forcément 0 ou 1 (souvenez vous, on avait posé comme convention que $w = 0$ représente un vecteur et $w = 1$ représente un point). C'est ce changement de la coordonnée w qui indique qu'une projection a été faite. On n'est alors plus dans un espace à 3 dimensions mais dans un espace projectif comportant 3 dimensions spatiales et une dimension homogène. Deux ensembles de coordonnées proportionnels représente alors le même point 3D, on a l'équivalence suivante :

$$\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \sim \begin{pmatrix} \alpha x \\ \alpha y \\ \alpha z \\ \alpha w \end{pmatrix}, \forall \alpha \neq 0$$

Un ensemble de coordonnées qui soit tel que $w = 1$ correspond effectivement aux coordonnées du point dans le "véritable espace 3D" (distordu, mais 3D quand même). Ainsi en divisant les 4 coordonnées d'un point dans l'espace clip par sa composante w_{Clip}, on se ramène à $w_{Clip} = 1$ et on obtient $(x_{Clip}/w_{Clip}, y_{Clip}/w_{Clip}, z_{Clip}/w_{Clip}, w_{Clip}/w_{Clip} = 1)$ dont les 3 première coordonnées sont les coordonnées 3D du point, appelées *Normalized Device Coordinates* :

$$\begin{pmatrix} x_{NDC} \\ y_{NDC} \\ z_{NDC} \end{pmatrix} = \begin{pmatrix} x/w_{Clip} \\ y/w_{Clip} \\ z/w_{Clip} \end{pmatrix}$$

A l'issue de cette transformation (non matricielle !), on a une propriété bien utile : l'ensemble des points visibles depuis la caméra ont des coordonnées entre -1 et 1 sur chacun des trois axes. Cela permet à OpenGL de jeter tous les points non visibles très facilement. De plus deux points se projetant sur le même pixel ont le même (x, y), il suffit alors de garder celui qui est le plus proche de la caméra pour avoir le point visible.

Espace écran

Cette transformation est la dernière : elle transforme les NDC des points en coordonnées de pixels. Pour cela on utilise les propriétés du viewport (souvenez vous de la fonction `glViewport` permettant de spécifier à OpenGL sur quelle partie de la fenêtre il doit dessiner, par défaut c'est toute la fenêtre). Puisque les NDC sont entre -1 et 1, cela revient juste à multiplier par la demi taille du viewport et à translater à sa position. A nouveau OpenGL s'occupe entièrement de cette transformation, la seule chose à faire est de spécifier le viewport avec la fonction mentionnée plus haut (ou pas si vous voulez dessiner dans toute la fenêtre).

▲ Cours : Utilisation de la bibliothèque `glm`

La lib `glm` est extrêmement utile lorsqu'il s'agit de construire des matrices pour OpenGL. D'une part elle utilise exactement les même noms de type / fonction que le langage `GLSL`. Ainsi vous apprenez un langage et l'utilisation d'une lib en même temps. D'autre part elle respecte les convention OpenGL (matrices représentés en mémoire colonne par colonne ou encore vue de la caméra du coté négatif des z). En plus de cela c'est une bibliothèque dite "full header". Cela signifie que tout le code de la lib est dans des `.hpp` et qu'il n'y a donc pas à la compiler, il suffit juste d'inclure les headers !

Tout le code de cette lib est dans le namespace `glm`. Trois headers doivent etre inclut pour la suite des TDs : `<glm/glm.hpp>` qui contient les types et fonctions de base qu'on peut trouver dans `GLSL`, `<glm/gtc/matrix_transform.hpp>` qui est une extension contenant des fonctions pour construire des matrices utiles et `<glm/gtc/type_ptr.hpp>` qui nous permettra d'envoyer nos données aux shaders.

Voici un exemple simple de création et d'addition de vecteurs avec `glm` :

```
glm::vec3 v1(1, 2, 3);
glm::vec3 v2(4, 5, 6);
glm::vec3 v3 = v1 + v2; // v3 vaut (5, 7, 9)
```

Comme vous pouvez le constater, on peut additionner des vecteurs avec l'opérateur `+`, bien que les variables soit d'un type non primitif ! C'est une des fonctionnalités du `C++` très utilisée pour les lib de maths : la redéfinition des opérateur. Cela fonctionne avec tous les types de glm : vec2, vec3, vec4, mat3, mat4, etc.

Voici comment créer une matrice de translation :

```
glm::mat4 translation = glm::translate(glm::mat4(1), glm::vec3(1, 0, 0));
```

Le premier argument de la fonction est une matrice qui sera multiplié par la translation avant d'etre renvoyé. Ici on passe la matrice identité, on obtiendra donc une matrice de translation "pure". Le deuxième argument est le vecteur de translation. Pour bien comprendre le rôle du premier argument, voici comment créer une rotation suivie d'une translation :

```
// rotation autour de (0, 1, 0) d'angle 45 degré:
glm::mat4 R = glm::rotate(glm::mat4(1), 45.f, glm::vec3(0, 1, 0));
// rotation suivie de translation
glm::mat4 RT = glm::translate(R, glm::vec3(1, 0, 0));
```

Ici on construit d'abord une rotation "pure" de 45 degré autour de l'axe $(0, 1, 0)$, puis on appelle la fonction translate sur la matrice de rotation crée, on obtient en sortie la multiplication de la matrice de rotation par la matrice de translation, soit $R \times T$. (notez que `glm` prend des angles exprimés en degrés, il se charge de faire la conversion en radians).

Les fonction `translate`, `rotate` et `scale` sont surtout utilisées pour construire la matrice Model. Elles prennent toutes en premier argument une matrice On peut également les utiliser pour construire la matrice View mais il est plus simple d'utiliser la fonction `lookAt`. Voici un exemple :

```
glm::mat4 View = glm::lookAt(glm::vec3(0, 2, 1), glm::vec3(8, 1, 2), glm::vec3(0, 1, 0));
```

Cette appel crée une matrice View correspondant à une caméra placé en $(0, 2, 1)$ (premier argument, position de l'œuil), observant le point $(8, 1, 2)$ (deuxième argument) et dont le vecteur vertical suit le vecteur $(0, 1, 0)$ (troisième argument). Attention, cela ne signifie pas que le vecteur y de la caméra sera $(0, 1, 0)$, mais qu'il sera situé dans le plan constitué des vecteurs $(0, 1, 0)$ et $(8, 1, 2) - (0, 2, 1)$ (vous verrez en pratique !)

Enfin les matrices de projection les plus communes peuvent être crée avec les fonctions `ortho` et `perspective` :

```
glm::mat4 Ortho = glm::ortho(-5.f, 5.f, 0.f, 2.f, 0.1f, 100.f);
```

On crée ici une matrice de projection orthogonale observant un rectangle allant de -5 à 5 sur son axe x et 0 à 2 sur son axe y. En plus de cela elle n'est capable de voir que les points dont la coordonnée z est entre -0.1 et -100.

```
glm::mat4 Perspective = glm::persective(70.f, WIN_WIDTH / (float) WIN_HEIGHT, 0.1f, 100.f);
```

Cette fois ci on crée une matrice de projection perspective (vue de type humaine). Celle ci représente une caméra ayant un angle de vision de 70 degré sur l'axe des y et de ratio `WIN_WIDTH / (float) WIN_HEIGHT` correspondant au ratio *largeur/hauteur* de la fenêtre. Comme la matrice ortho elle peut voir entre -0.1 et -100 sur l'axe des z.

Une fois que l'on a créé des matrices, il faut les envoyer au vertex shader sous la forme de variables uniformes. Pour cela il faut utiliser la fonction `glUniformMatrix4fv` comme dans le TP précédent (mis à part qu'on envoyait des matrices 3x3). Le dernier argument de la fonction est un pointeur sur les données de la matrice, or `glm` ne nous fournit pas directement ce pointeur, nous n'avons que des `mat4` ! Pour cela nous devons utiliser la fonction `value_ptr` sur une matrice de `glm` pour obtenir le pointeur correspondant :

```
glm::mat4 Ortho = glm::ortho(-5.f, 5.f, 0.f, 2.f, 0.1f, 1000.f);

GLint projectionMatrixLocation = glGetUniformLocation(program, "uProjectionMatrix");
glUniformMatrix4fv(projectionMatrixLocation, 1, GL_FALSE, glm::value_ptr(Ortho));
```

Voici un bout de programme permettant d'avoir une vue perspective sur une sphere et un cube respectivement translaté et rotaté (on suppose que sphereVAO et cubeVAO sont les VAOs permettant de dessiner ces deux objets) :

Programme C++ :

```
[...]

// MVPMatrix est la matrice ModelViewProjection dans le vertex shader
GLint MVPLocation = glGetUniformLocation(program, "uMVPMatrix");
```

```
// Construction matrice de projection, elle ne changera pas
glm::mat4 P = glm::persective(70.f, WIN_WIDTH / (float) WIN_HEIGHT, 0.1f, 1000.f);

// Construction de la matrice view, elle ne changera pas non plus tant qu'on ne peut pas
// controller la caméra avec la souris (prochain TD)
// D'après les conventions OpenGL, cela revient à créer la matrice identité dans ce cas
// car par défaut la caméra est placé en (0, 0, 0) et regarde vers l'axe négatif des
// z
glm::mat4 V = glm::lookAt(vec3(0, 0, 0), vec3(0, 0, -1), vec3(0, 1, 0));

// Projection fois View (ViewProjection Matrix) :
glm::mat4 VP = P * V;

// Il faut activer le test de profondeur des qu'on fait de la 3D:
glEnable(GL_DEPTH_TEST);

// boucle de rendu:
bool done = false;
while(!done) {
    // il faut "clear" deux buffers à présent: couleur et profondeur
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Construction de la translation pour la sphere, prémultiplié par VP
    // On obtient donc la matrice ModelViewProjection pour la sphère
    glm::mat4 MVP = glm::translate(VP, glm::vec3(-1, 0, 0));
    // Envoi de la matrice au vertex shader
    glUniformMatrix4fv(MVPLocation, 1, GL_FALSE, glm::value_ptr(MVP));

    glBindVertexArray(sphereVAO);
       // dessin de la sphere, la variable NumVertexSphere contient
       // le nombre de sommets associé
       glDrawArrays(GL_TRIANGLES, 0, NumVertexSphere);
    glBindVertexArray(0);

    // Construction de la rotation pour le cube, prémultiplié par VP
    // On obtient donc la matrice ModelViewProjection pour la cube
    // Puisqu'on prémultiplie par VP, on a "oublié" la translation
    // précédente, le cube ne subira donc qu'une rotation
    MVP = glm::rotate(VP, 45.f, glm::vec3());
    // Envoi de la matrice au vertex shader
    glUniformMatrix4fv(MVPLocation, 1, GL_FALSE, glm::value_ptr(MVP));

    glBindVertexArray(cubeVAO);
       // dessin du cube, la variable NumVertexCube contient
       // le nombre de sommets associé
```

```
        glDrawArrays(GL_TRIANGLES, 0, NumVertexCube);
    glBindVertexArray(0);

    SDL_GL_SwapBuffers();

    [...]
```

Vertex Shader :

```
#version 330

layout(location = 0) in vec3 aVertexPosition;

// Matrice ModelViewProjection reçu depuis l'application
uniform mat4 uMVPMatrix;

void main() {
    gl_Position = uMVPMatrix * vec4(aVertexPosition, 1.f);
}
```

Le shader n'effectue ici que la transformation. On pourrait parfaitement l'améliorer pour également prendre une couleur en entrée et la passer dans le fragment shader (l'exemple est minimal), etc.

▲ Cours : Utilisation des classes de forme

Les classes de forme fournies avec le template sont Sphere, Cylinder et Cone. Elles s'utilisent toutes de la même manière et permettent de construire 3 attributs pour les objets cités : position, normale et coordonnées de texture. Les fragments shaders qui vous sont fourni utilisent les deux derniers attributs pour l'affichage. Vous n'aurez pour l'instant à coder que le vertex shader, mais il faudra faire en sorte qu'il fournisse en sortie ("out") ces deux attributs (une simple copie).

Voici un exemple d'utilisation pour la classe Sphere :

```
#include "imac2gl3/shapes/Sphere.hpp"

[...]

// A l'initialisation dans le main:
// Création d'une sphere de rayon 1, avec comme facteur de discretisation 50 sur la latitude
// et 50 sur la longitude
imac2gl3::Sphere mySphere(1.f, 50, 50);

// Stockage dans un VBO entrelacé:
GLuint sphereVBO;
glGenBuffers(1, &sphereVBO);
glBindBuffer(GL_ARRAY_BUFFER, sphereVBO);
```

```
    glBufferData(GL_ARRAY_BUFFER, mySphere.getByteSize(), mySphere.getDataPointer(),
    GL_STATIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

// Construction du VAO associé:
GLuint sphereVAO;
glGenVertexArrays(1, &sphereVAO);
glBindVertexArray(sphereVAO);
    glEnableVertexAttribArray(POSITION_LOCATION);
    glEnableVertexAttribArray(NORMAL_LOCATION);
    glEnableVertexAttribArray(TEXCOORDS_LOCATION);
    glBindBuffer(GL_ARRAY_BUFFER, sphereVBO);
        glVertexAttribPointer(
            POSITION_LOCATION,
            mySphere.getPositionNumComponents(),
            mySphere.getDataType(),
            GL_FALSE,
            mySphere.getVertexByteSize(),
            mySphere.getPositionOffset());
        glVertexAttribPointer(
            NORMAL_LOCATION,
            mySphere.getNormalNumComponents(),
            mySphere.getDataType(),
            GL_FALSE,
            mySphere.getVertexByteSize(),
            mySphere.getNormalOffset());
        glVertexAttribPointer(
            TEXCOORDS_LOCATION,
            mySphere.getTexCoordsNumComponents(),
            mySphere.getDataType(),
            GL_FALSE,
            mySphere.getVertexByteSize(),
            mySphere.getTexCoordsOffset());
    glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);

[...]

// Dans la boucle principale:

glBindVertexArray(sphereVAO);
    glDrawArrays(GL_TRIANGLES, 0, mySphere.getVertexCount();
glBindVertexArray(0);

[...]
```

Ce qu'il faut retenir de cet exemple, c'est que chaque classe de forme comporte :

- Un constructeur adapté à la forme permettant de créer les positions des sommets en fonction des propriétés de la forme (rayon pour la sphère, taille d'un coté pour le cube, etc.) et des facteurs de discrétisation (la discrétisation consiste à transformer la forme continue en triangles, plus ces facteurs sont élevé, plus on place de triangles et donc plus la forme triangulé est proche de la forme continue).
- Une méthode `getByteSize()` renvoyant la taille totale des données en nombre d'octets (à passer à `glBufferData`)
- Une méthode `getDataPointer()` renvoyant un pointeur vers les données (à passer également à `glBufferData`)
- Une méthode `get<Attribute>NumComponents()` renvoyant le nombre de composante de l'attribut (à passer à la fonction `glVertexAttribPointer` lors de la spécification du format de l'attribut)
- Une méthode `getDataType()` renvoyant le type d'une composante d'attribut (à passer également à `glVertexAttribPointer` ; tous les attributs ont des composantes de même type)
- Une méthode `get<Attribute>Offset()` renvoyant l'offset de l'attribut par rapport au début du tableau des données (à passer également à `glVertexAttribPointer`)
- Une méthode `getVertexByteSize()` renvoyant la taille d'un sommet en octets (à passer également à `glVertexAttribPointer`)
- Une méthode `getVertexCount()` renvoyant le nombre de sommets contenu dans le tableau des données (à passer à `glDrawArrays`).
- Les données sont entrelacés et packés dans le tableau. Vous devez donc utiliser les méthodes précédentes pour construire correctement VBO, VAO et l'appel de dessin.

▸ **Exercice 1.** ***Préparer son code pour la 3D***

*Comme vous l'avez vu l'année dernière en cours (normalement !), l'algorithme de rasterisation nécessite un buffer de profondeur (*Depth buffer*) pour que les faces arrières ne viennent pas recouvrir les faces avant. Pour faire cela avec OpenGL, il faut activer la fonctionnalité en utilisant la fonction* `glEnable`*. Cette fonction prend une constante en paramètre indiquant la fonctionnalité à activer (il y en a plein !). La liste des constantes est disponible sur cette page : http ://www.opengl.org/sdk/docs/man3/xhtml/glEnable.xml. A vous de trouver la bonne ! Ajoutez ensuite l'appel dans l'initialisation de votre programme.*

Ensuite il faut à chaque tour de boucle "nettoyer" le buffer de profondeur (comme on nettoie le buffer de couleur pour que l'écran redevienne noir). On fait cela avec la fonction `glClear` *en ajoutant une constante dans l'appel en plus de* `GL_COLOR_BUFFER_BIT`*. A vous de trouver la bonne constante sur la page : http ://www.opengl.org/sdk/docs/man3/xhtml/glClear.xml. Il faut combiner cette constante à la première avec un ou binaire (opérateur* `|` *en C++), de la manière suivante :*

```
glClear(GL_COLOR_BUFFER_BIT | LA_CONSTANTE_A_TROUVER);
```

▸ **Exercice 2.** ***Écrire le vertex shader***

Vous devez à présent écrire le vertex shader. Celui ci doit être en accord avec les fragments shaders présent dans le dossier `shaders`*. Cela signifie que ses variables "out" doivent correspondre aux variables "in" des fragment shaders. Pour l'instant nous allons accorder le VS avec les FS* `texcoordscolor.fs.glsl` *et* `normalcolor.fs.glsl`*. Ces shaders affichent respectivement les coordonnées de texture et les normales comme des couleurs. Ouvrez ces shaders et identifiez les variables "in". Créez ensuite le fichier* `transform.vs.glsl`*. Vous devez y écrire le code pour un vertex shader faisant les choses suivantes :*

- *Il prend en entrée 3 attributs : position, normale et coordonnées de texture (de types vec3, vec3 et vec2).*
- *Il fournit en sortie les variables requises par les fragments shaders.*
- *Il contient une variable uniforme de type mat4 appelée* `uMVPMatrix` *destinée à contenir la matrice Model View Projection (elle sera envoyée par l'application au moment du dessin).*
- *Elle effectue la transformation de la position d'entrée par la matrice MVP et place le résultat dans la built in* `gl_Position` *(n'oubliez pas qu'il faut convertir la position d'entrée en vec4 en ajoutant un 1 à la fin !). Ainsi OpenGL obtiendra les position projetées des sommet et pourra faire la rasterisation dessus.*

▸ **Exercice 3.** ***Afficher une sphère***

Nous allons à présent essayer le shader. Pour cela créez une sphère de rayon 1 et de facteurs de discrétisation 60 et 30 à l'aide de la classe `Sphere` *présentée plus haut. Vous devez remplir convenablement un VBO et un VAO avec les données de la sphère pour pouvoir la dessiner avec OpenGL. Tout cela doit être fait dans la partie initialisation du programme. Toujours dans la partie initialisation, créez avec* `glm` *une matrice de projection perspective* `P` *d'angle vertical 70 degré, de ratio largeur fenêtre sur hauteur fenêtre et pouvant voir de* 0.1 *à* 1000 *sur son axe de profondeur (même appel que dans la partie cours présentant* `glm`*). Créez également une matrice view* `V` *positionnée en* $(0, 0, 0)$*, regardant le point* $(0, 0, -1)$ *et de vecteur vertical* $(0, 1, 0)$*. Enfin créez une matrice model* `M` *effectuant une translation de* $(0, 0, -5)$*. Chargez les shaders* `shaders/transform.vs.glsl` *et* `shaders/texcoordscolor.fs.glsl` *et utilisez le programme GLSL résultant.*

Dans la boucle principale, envoyez la matrice `P * V * M` *au vertex shader dans la variable uniforme* `uMVPMatrix` *puis dessinez l'objet. Observez le résultat et essayez ensuite de changer le vecteur translation pour voir comment se comporte l'objet.*

Essayez ensuite d'afficher un cone et un cylindre à la place de la sphère.

▸ **Exercice 4.** ***Un peu de mouvement***

Remettez en place le code dessinant la sphère et faites la bouger entre les points $(-2, 0, -5)$ *et* $(2, 0, -5)$ *(souvenez vous, pour que la sphère ne dépasse pas ces points il suffit d'appliquer judicieusement un cosinus sur la coordonnée* x *de la translation). Pour cela vous devrez modifier à chaque tour de boucle la matrice model avant d'envoyer la multiplication des 3 au shader.*

Faites ensuite en sorte que la sphère tourne autour du point $(0, 0, -5)$ à distance 2 sur le plan xy (pas de changement de profondeur donc, elle doit tourner en face de vous). Il faut construire la matrice modèle avec des combinaisons de rotation et translation. Attention à l'ordre ! n'oubliez pas que la dernière matrice multipliée à droite agit sur le nouveau repère local de l'objet. Il faut donc d'abord translater en $(0, 0, -5)$, qui devient alors le centre du repère local de l'objet. Ensuite on multiplie par une rotation selon l'angle voulu autour du vecteur $(0, 0, -1)$. Cela a pour effet de faire tourner les angles du repère local de l'objet. Enfin on effectue une translation de $(0, 2, 0)$ pour décaler la sphère sur son axe y (qui à changé du fait de la rotation effectué plus tôt).

▸ **Exercice 5. *La terre et la lune***

Exercice classique en 3D mais efficace ! Créez une deuxième sphère (avec le VBO et VAO qui vont avec) de rayon 0.1. Vous devez faire en sorte que celle ci tourne autour de la grosse sphère. Essayez d'abord en maintenant la grosse sphère statique, puis une fois que ça fonctionne, faites en sorte que la grosse tourne comme dans l'exercice précédent et que la petite la suive correctement.

▸ **Exercice 6. *Factoriser le code***

L'exercice suivant à du beaucoup augmenter la taille de votre code : l'ajout d'une sphère conduit à devoir recopier toute le code de création de VBO/VAO alors qu'il est quasiment identique ! Écrivez une classe `GLShapeInstance` *(dans les répertoires* `include/imac2gl3/shapes` *et* `src/imac2gl3/shapes`*) permettant de factoriser ce code. Elle doit contenir 3 variables membre : vbo, vao et vertexCount et son constructeur doit prendre en paramètre une Sphère (sous forme d'une référence constante). A partir de cette sphère il crée les vbo, vao membre et fixe la variable vertexCount au nombre de vertex de la sphère. Vous devez également écrire une méthode draw() qui dessiner l'objet à partir du VAO. Refactorisez ensuite votre code pour utiliser cette classe.*

Note : votre classe prend en paramètre un objet de type Sphere. A priori le code pour un Cylinder ou un Cone serait le même. A l'aide de templates C++ vous pouvez coder un seul et même constructeur pour toutes les formes possibles respectant la même interface.

▸ **Exercice 7. *Pile de matrices***

Nous avons vu dans les exercices précédents comment créer des matrices avec GLM et les envoyer à nos shaders avec la fonction `glUniformMatrix`*. Dans l'exercice concernant la terre et la lune nous pouvons remarquer que la matrice pour placer la lune est composé en partie de la matrice pour placer la terre. Cela est du au fait que la lune tourne autour de la terre, on peut ainsi considérer qu'elle est placée dans une sous-scène dans laquelle le centre du repère est situé au centre de la terre. Si nous voulions placer un autre satellite autour de la terre, il faudrait reconstruire la matrice ModelView de la terre et y ajouter la matrice correspondant au nouveau satellite avant de le dessiner.*

En OpenGL 2 on pouvait facilement faire cela en utilisant le mécanisme de pile de matrice : `glPushMatrix` *et* `glPopMatrix`*. Ces fonctions permettait de construire itérativement la*

matrice ModelView tout en sauvegardant l'état courant de la matrice pour facilement revenir en arrière en conservant la matrice. En pseudo-code on pouvait dessiner la terre ainsi que la lune et un satellite de la manière suivante :

```
glPushMatrix(); // Sauvegarde de la matrice courante
Construction matrice ModelView terre
Dessin terre
glPushMatrix(); // Sauvegarde la matrice de la terre
Construction matrice ModelView lune depuis la terre
Dessin lune
glPopMatrix(); // Restauration de la matrice de la terre: on oublie la lune
glPushMatrix(); // Sauvegarde la matrice de la terre
Construction matrice ModelView du satellite depuis la terre
Dessin satellite
glPopMatrix(); // Restauration de la matrice de la terre: on oublie le satellite
glPopMatrix(); // Restauration de la matrice initiale: on oublie la terre
```

Ce mécanisme permet de construire des scène structurées assez facilement. Nous allons le reproduire avec une classe C++.

- *Dans le module* `imac2gl3` *créez une nouvelle classe* `MatrixStack` *(le .hpp dans le répertoire des headers et le .cpp dans le répertoire des sources)*
- *Ajoutez une variable membre privée* `m_Stack` *de type* `std::stack<glm::mat4>` *(il faut inclure le header* `<stack>` *pour utiliser le type* `std::stack`*).*
- *Dans le constructeur de votre classe, ajoutez la matrice identitée dans la pile (*`m_Stack.push(glm::mat4(1.f))`*). De cette manière votre pile de matrice est initialisée à l'identité.*
- *Ajoutez la méthode publique* `void push()` *qui sauvegarde la matrice en tête de pile (pour cela il suffit d'ajouter à la pile la même matrice que celle actuellement en tête, c'est à dire :* `m_Stack.push(m_Stack.top())`*, la méthode* `top()` *renvoyant une référence sur la matrice du haut de la pile).*
- *Ajoutez la méthode publique* `void pop()` *qui retire la dernière matrice ajoutée à la pile (*`m_Stack.pop()`*)*
- *Ajoutez la méthode* `void mult(const glm::mat4& m)` *qui multiplie la matrice de tête de pile par la matrice* `m`*.*
- *Ajoutez la méthode* `const glm::mat4& top() const;` *permettant de récupérer la matrice en tête de pile.*
- *Ajoutez la méthode* `void set(const glm::mat4& m)` *permettant de remplacer la matrice en tête de pile par la matrice* `m`*.*
- *Ajoutez les méthodes* `void scale(const glm::vec3& s)`, `void translate(const glm::vec3& t)` *et* `void rotate(float degrees, const glm::vec3& r)` *qui appliquent les transformations classiques à la matrice en tête de pile.*

A noter que toutes ces méthodes s'écrivent en une ligne, il n'y a donc normalement pas à se prendre la tête.

Utilisez ensuite cette classe pour réécrire l'exercice sur la terre et la lune en l'utilisant.

▸ **Exercice 8.** ***Un cube***

Dans le template, les classe `Sphere`, `Cone` *et* `Cylinder` *vous sont fournies. Écrivez dans le même module (*`imac2gl3/shapes`*) la classe* `Cube` *(beaucoup plus simple à coder que les 3 autres). Pour cela vous devez implanter les même méthodes que pour les 3 autres classes en les adaptant pour le cube (la plupart des méthodes restent normalement identiques). Attention à ne pas trop copier : dans les 3 autres classes j'ai été obligé de faire de l'allocation dynamique car on ne connaît pas à l'avance le nombre de sommets. Or pour le cube le nombre de sommets est connu (8 sommets !), donc pas besoin d'allouer dynamiquement les données. N'oubliez pas les normales et les coordonnées de texture. La normale est le vecteur unitaire perpendiculaire à la surface en un point donné ; dans le cas du cube la normale est la même pour les 4 sommets d'une même face. Les coordonnées de texture doivent être comprises entre 0 et 1. Il suffit pour chaque face d'adopter la convention suivante : le sommet en bas à gauche de la face a* $(0, 0)$ *pour coordonnées de texture, celui en haut à gauche a* $(0, 1)$*, celui en bas à droite a* $(1, 0)$ *et celui en haut à droite a* $(1, 1)$*.*

▸ **Exercice 9.** ***La terre et la lune - le retour***

Dans l'exercice sur la terre et la lune vous avez normalement créé deux VBOs et deux VAOs, un de chaque pour chacun des deux objets. Or les formes sont les même : la terre et la lune sont toutes deux des sphères (de rayon différent certes, mais des sphère quand même). En faisant cela, on stocke inutilement deux modèles identiques sur la carte graphique. Reprenez le code de l'exercice en utilisant qu'un seul couple VBO, VAO (utilisez votre classe `GLShapeInstance`*) mais en dessinant deux fois l'objet avec une transformation adaptée pour gérer la différence de rayon entre la terre et la lune.*