# Introduction à OpenGL : Système solaire

Thibaud Lambert, Brett Ridel

La bibliothèque OpenGL est une interface de programmation regroupant un ensemble de fonctions permettant de générer des images 2D et 3D. L'objectif de ce TP est d'introduire et de pratiquer les notions principales du pipeline OpenGL à travers la mise en place d'un système solaire. Un résumé de l'ensemble des fonctions OpenGL est disponible ici : `https://www.khronos.org/files/opengl-quick-reference-card.pdf`.

Afin de pouvoir utiliser OpenGL, il faut tout d'abord créer un **context** et une fenêtre capable d'afficher un rendu. Ces deux tâches sont effectuées ici par la bibliothèque GLFW, dont la documentation est disponible à l'adresse suivante : `http://www.glfw.org/docs/latest/`. Pour les matrices et les vecteurs, la bibliothèaue Eigen est utilisée : `http://eigen.tuxfamily.org/dox-devel/group__QuickRefPage.html`.

Le fichier **main.cpp** initialise le contexte OpenGL, crée la fenêtre principale et exécute l'application interactive. La fonction **main()** appelle la méthode **Viewer::init** après la création de la fenêtre et du contexte OpenGL, **Viewer::reshape** lorsque la fenêtre est redimensionnée et **Viewer::update** lorsque l'image doit être redessinée. Il n'est pas nécessaire de modifier le fichier **main.cpp** pour effectuer le TP.

La méthode **Viewer::init** est appelée une seule fois après la création de la fenêtre et du contexte OpenGL. Cette méthode regroupe le code d'initialisation de la scène : chargement des maillages et des shaders et la définition de différents paramètres du pipeline OpenGL (couleur de l'arrière plan, dimension du viewport).

La méthode **Viewer::update** se charge de mettre à jour la scène puis effectue le rendu.

## 1 Texture procédurale

Pour l'instant, l'application charge et affiche une sphère blanche sans aucune transformation des coordonnées. Voir les fichiers des vertex et fragment shaders :

- **data/shaders/simple.vert**
- **data/shaders/simple.frag**

Afin de mieux distinguer la forme du maillage, nous allons lui rajouter une texture. Cette étape permet de se familiariser avec la syntaxe **GLSL**. Pour cela nous allons faire varier la couleur de la surface de l'objet en fonction de la position du point de la surface au sein d'un damier 3D (figure 1). Nous pouvons facilement identifier les cellules "noires" et les cellules "blanches" du damier à partir des indices $i, j, k$ de la cellule contenant le point courrant: nous lui affecteront une couleur $C_1$ ou $C_2$ en fonction de la parité de la somme des indices.

### 1.1 Par sommets

Dans un premier temps nous appliqueront le motif de damier par sommet. Cela nécessite deux modifications:

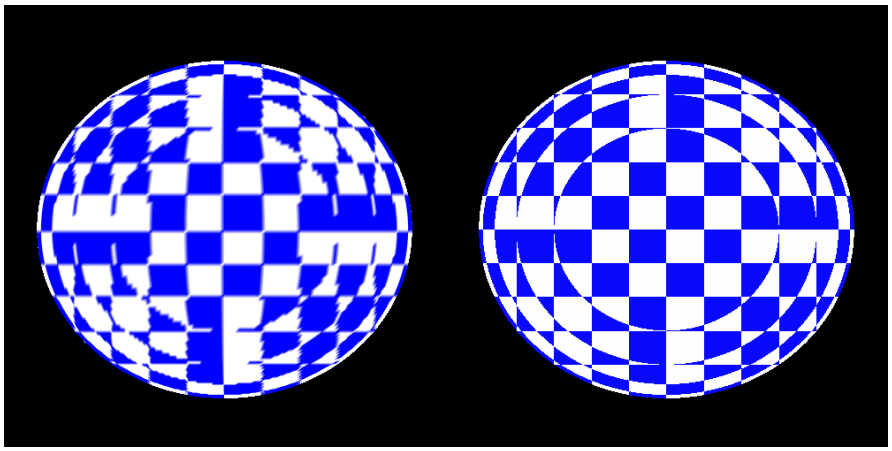

Figure 1 – La sphère texturée avec un damier 3D. Gauche : par sommets, Droite : par pixels

- Calculer la couleur dans le vertex shader.
- Transférer les informations de couleurs des sommets aux fragments et la copier en sortie du fragment shader. Pour cela, il passer par une variable en sortie du vertex shader (**out**) et une autre en entrée du fragment shader (**in**) avec le même nom (voir exemple slide 76-77 du cours).

Que remarquez vous d'un point de vue de la qualité visuelle ?

## 1.2 Contrôle de la taille des cellules

Nous souhaitons maintenant faire varier la taille des cellules de la grille dynamiquement. Pour cela, il est possible d'utiliser une variable **uniform**. Un uniform est une variable globale dont la valeur est définie par l'appliquation. Elle est déclarée avec le mot clé **uniform** (voir exemple slide 76-77 du cours). Rajoutez un uniform pour la taille d'une case (dans le vertex shader).

Pour définir sa valeur, il faut tout d'abord récupérer l'identifiant de la variable, puis définir sa valeur (dans la fonction **Viewer::update**) :

```
int uniform_name_id = _prog.getUniformLocation("uniform_name");
glUniform1f(uniform_name_id,value);
```

Contrôler la taille au clavier en configurant deux touches dans la méthode **Viewer::keyPressed**. (inspirez vous de ce qui est fait pour le rechargement des shaders)

## 1.3 Par pixels

Le calcul des couleurs par sommet ne donne pas une grille nette car les couleurs sont interpolées linéairement entre les sommets. Afin d'obtenir une grille nette, il est nécessaire de calculer la couleur pour chaque pixel. Déplacez le code du calcul de la couleur dans le fragment shader.

# 2 Caméra et transformation

## 2.1 Caméra

Il est pratique de contrôler la caméra en lui indiquant une position d'observation **position**, une cible à viser **target** et une orientation verticale **up**. Complétez la méthode **Camera::lookAt** qui permet de positionner spatialement la caméra dans l'environnement 3D. Pour cela, la méthode modifie la matrice **mViewMatrix**. (voir slide 15-17 du cours) Vous devez construire une base orthonormale 3D dont l'axe z est l'opposé du vecteur de visée. L'axe x est orthogonal au vecteur up et z. Le dernier axe y du repère peut finalement être obtenu par produit vectoriel entre les deux vecteurs précédent. Assemblez alors la matrice 4x4 de changement de base à partir de ces vecteurs en n'oubliant pas que le repère doit être centré sur la position d'observation.

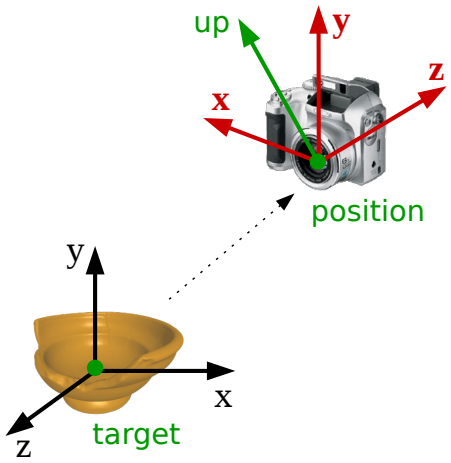


Figure 2 – Positionnement de la caméra

Notez que la classe **Viewer** possède déjà un attribut Camera (variable **_cam**). Initialisez ses paramètres intrinsèques (méthode setPerspective) et extrinsèques (méthode lookAt) dans la fonction **Viewer::init**.

Pour que notre rendu soit vue de la camera, il est nécessaire d'appliquer la matrice de vue $M_v$ et la matrice de projection $M_p$ à chacun de nos sommets. Envoyer les deux matrices au vertex shader à l'aide de la fonction **glUniformMatrix4fv**. Appliquer à chaque sommet les matrices.

```
int mat_name_loc = _prog.getUniformLocation("mat_name");
glUniformMatrix4fv(mat_name_loc,1,GL_FALSE,mat.data());
```

## 2.2 Matrice objet

Afin de pouvoir repositioner les objets dans l'espace, rajoutez en plus des deux autres matrices une matrice de transformation. Envoyer la au vertex shader et appliquer la aux sommets.

# 3 Systéme solaire

Pour appliquer les concepts précédents à un cas concret, vous allez mettre en œuvre un mini système solaire animé. Il inclura au moins la terre, qui devra tourner sur son axe et être en révolution autour du soleil. Vous ajouterez ensuite notre lune en révolution autour de la terre.

Commencez par un système solaire minimaliste avec une sphère au centre de la scène représentant le soleil et une seconde sphère, plus petite, tournant autour du soleil à une distance fixe.

Dans la méthode **Viewer::update**, le temps écoulé depuis le début de l'application est passé en paramètre. Faire varier les matrices de transformations enf ocntion du temps pour animer votre systéme solaire.

Conseil : Vous pouvez tracer plusieurs fois votre sphère en lui passant une matrice de transformation différente.

Faites en sorte que la terre tourne sur elle-même autour d'un second axe. Modifiez l'axe de la terre afin que son axe de rotation soit incliné (d'environ 23 degrés) comme c'est le cas en réalité. Vous pouvez utiliser la classe Eigen::AngleAxis pour effectuer une rotation autour d'un axe arbitraire. Sur le même principe, ajoutez une lune tournant autour de la terre

Vous pouvez continuer en ajoutant d'autres planètes et leurs lunes pour former le système solaire complet.