OpenGL – TD 04

Let's go 3D !

Lors de cette séance, nous aborderons la création d'une scène virtuelle 3D simple mais complète ainsi que son animation, permettant d'aborder le concept de modélisation de scène « hiérarchique ». Pour ce faire, nous allons reproduire une balance (de pesée) ainsi que son animation qui restera simple. Nous pourrons naviguer tout autour de cet objet 3D.

Exercice 01 – Mise en place de l'application 3D

Pour passer à la 3D, nous allons devoir faire certaines modifications afin d'avoir à la fois un code plus clair et surtout qui fonctionne bien en 3D. En effet, il faudra légèrement changer notre moteur de rendu, et également gérer le déplacement de notre caméra dans l'espace 3D.

A faire (il n'y aura aucune conséquence sur le rendu avant le 04) :

01. Tout d'abord, comme évoqué à la fin du TD précédent, nous allons un peu mieux organiser notre code. Pour ce faire, créez déjà, comme d'habitude, un répertoire nommé `TD04`. Importez y un fichier exercice du TD précédent et nommez le `exo1.cpp`. Veillez à retirer tout le travail du TD précédent mais également la création de la variable `myEngine` au début du fichier. Puis ajoutez dans le répertoire `TD04` les fichiers `draw_scene.[ch]pp` que vous aurez récupéré sur votre espace d'elearning ou via votre chargé de TD. Enfin, incluez (avec un `#include`) le fichier `draw_scene.hpp` dans le fichier `exo1.cpp`.

Note : Lorsque vous modifierez uniquement les fichiers `draw_scene`, il ne sera plus nécessaire de recopier les fichiers `exXX.cpp` entre chaque exercice. Si vous souhaitez conserver le code réalisé pour un exercice spécifique, faites des backup des fichiers `draw_scene`.

02. La première chose à faire, sera de basculer notre moteur de rendu (`myEngine` situé maintenant dans le fichier `draw_scene.hpp`). Tout d'abord, avant l'initialisation du moteur `myEngine`, nous allons lui demander de basculer en 3D. Donc dans le fichier, avant l'initialisation du moteur (fonction `initGL` appelée sur notre moteur), intégrez la ligne

```
myEngine.mode2D = false; // Set engine to 3D mode
```

Il faut également fixer la projection, qui n'est plus une projection en 2D, mais une projection en 3D. Dans la fonction `onWindowResized` remplacez le code de calcul de la projection 2D par la ligne suivante :

```
myEngine.set3DProjection(90.0,aspectRatio,Z_NEAR,Z_FAR);
```

Pour information, le premier argument représente l'angle de vue de la caméra (ici 90°), le suivant est le rapport largeur / hauteur de la fenêtre, le troisième indique la distance entre le point focale de la caméra et le plan image où va être dessiné la scène (appelé plan z proche ou *z near plan*). Le dernier argument est la distance maximale de vue de la caméra qui forme un plan au loin. Il est donc appelé le plan z lointain ou *far z plane*). Rien de ce qui est derrière ce plan ne sera dessiné.

Enfin, nous devrons, dans la boucle de rendu, explicitement activer la gestion des occlusions et rendre notre scène en intégrant les lignes de code suivantes :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);

drawScene();
```

03. Enfin, il nous reste à gérer la caméra. Dans ce TD, nous utiliserons une caméra dite « Trackball Camera ». C'est un type de caméra qui tourne autour du centre de la scène et qui, généralement, est utilisée pour regarder des objets ou des scènes. Commencez par intégrer le code ci-dessous juste au dessus de l'appel à `drawScene`.

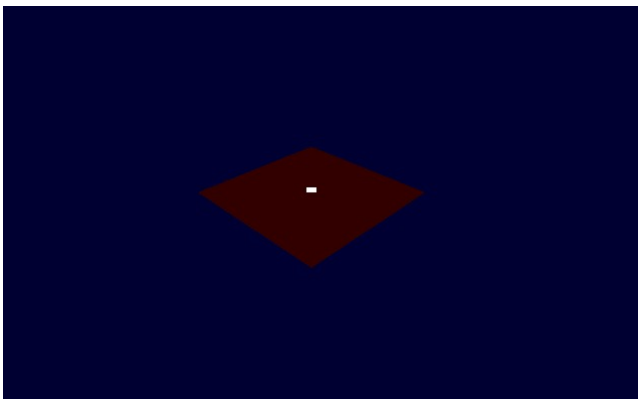
```
/* Fix camera position */
myEngine.mvMatrixStack.loadIdentity();
Vector3D pos_camera =
Vector3D(dist_zoom*cos(deg2rad(angle_theta))*cos(deg2rad(angle_phy)),
       dist_zoom*sin(deg2rad(angle_theta))*cos(deg2rad(angle_phy)),
       dist_zoom*sin(deg2rad(angle_phy)));
Vector3D viewed_point = Vector3D(0.0,0.0,0.0);
Vector3D up_vector = Vector3D(0.0,0.0,1.0);
Matrix4D viewMatrix = Matrix4D::lookAt(pos_camera,viewed_point,up_vector);
myEngine.setViewMatrix(viewMatrix);
myEngine.updateMvMatrix();
```

Ces lignes de code définissent d'une part la position de la caméra dans l'espace 3D (variable `pos_camera`) mais également le point visé (`viewed_point`). Nous créons ensuite une matrice 4D (`view_Matrix`) qui définit les transformations nécessaires pour envoyer les coordonnées des objets dans le repère de la caméra. Cette matrice est ajoutée à notre pile de matrice via le `myEngine` (`setViewMatrix`) puis envoyée à OpenGL (`updateMvMatrix`).

Il serait bon également de pouvoir agir sur notre caméra. Pour ce faire ajouter à la fonction de gestion du clavier les lignes suivantes :

```
    case GLFW_KEY_UP :
        angle_phy += 1.0;
        break;
    case GLFW_KEY_DOWN :
        angle_phy -= 1.0;
        break;
    case GLFW_KEY_LEFT :
        angle_theta += 1.0;
        break;
    case GLFW_KEY_RIGHT :
        angle_theta -= 1.0;
    break;
```

A ce stade, vous pouvez compiler et exécuter votre programme. Vous devriez voir un point blanc au centre de la scène 3D juste au dessus d'un carré sombre, soit le rendu suivant (les couleurs peuvent varier éventuellement) :



Vous pouvez visualiser la scène en fil de fer avec la touche 'l' et revenir à la normale via la touche 'p' (si vous avez gardé cette gestion du TD précédent). Vous pouvez également tourner autour de la scène avec les flèches de votre clavier.

Note importante : Regardez, au début du fichier `draw_scene.cpp`, les constructeurs des structures `GLBI_Set_Of_Points` et `GLBI_Convex_2D_Shape` qui prennent en argument l'entier 3 : il s'agit d'indiquer que nous souhaitons faire de ces objets, des objets 3D et non 2D. Et vous constaterez qu'à leur construction, ces objets prennent des coordonnées 3D.

04. Maintenant nous allons affichez un repère. En utilisant une nouvelle structure de set de points, créez un objet `frame` qui définit un repère en 3D (et non en 2D comme au TD précédent), avec un axe x rouge, l'axe y en vert et l'axe z en bleu, tous de taille 10.0. Dans la fonction `drawFrame`, dessinez ce repère. Dans la fonction `drawScene`, remplacer le dessin du point par le dessin du repère.

Exercice 02 – Dessin et animation d'une sphère

Cet exercice va maintenant vous apprendre à dessiner des objets en 3D. Pour cet exercice, et par la suite, nous nous contenterons d'exploiter des objets 3D dit « canoniques ». Ceux-ci représentent des formes de base que vous placerez et dessinerez dans la scène. Ces objets sont créés à l'initialisation et permettent, lorsque vous les dessinez, de les rendre dans leur repère propre. Autrement dit au niveau du repère défini actuellement. C'est tout à fait similaire à ce que vous avez vu au TD précédent mais en 3D…

A faire :

01. Dessiner une sphère : Dans un premier temps, nous allons apprendre à créer (et rendre) des objets canoniques telle la sphère. Ces formes de base sont déjà pré-définies dans le fichier `basic_mesh.hpp` du répertoire `tools`. Ainsi pour créer une sphère, définissez d'abord un « objet » de type `IndexedMesh` comme ceci :

```
IndexedMesh* sphere;
```

Puis dans la fonction `initScene`, créez explicitement la sphère :

```
sphere = basicSphere();
sphere->createVAO();
```

Cela crée une sphère de rayon 1.0 (il est possible de modifier le rayon mais c'est inutile). Vous pouvez maintenant rendre la sphère (en jaune par exemple) en appelant sur l'objet la fonction `draw`. La seconde fonction (`createVAO`) enregistre cet objet dans OpenGL et est nécessaire.

Note importante : L'étoile située derrière `IndexedMesh` lors de la déclaration de la sphère est un pointeur. C'est une manière différente d'accéder à la structure. Comme vous avez pu le voir, pour accéder aux membres (variables et fonctions) de la structure, il vous faudra utiliser `->` à la place du point (`.`). Les pointeurs sont très souvent utilisés pour leur efficacité et leur flexibilité mais leur manipulation est dangereuse et peut provoquer de mauvais accès mémoire.

02. Positionner et dimensionner la sphère : Faites en sorte maintenant que la sphère soit centrée sur le point (4,0,5) et de taille 3. Mais comment peux t'on faire ? Avec les matrices bien évidemment !

03. Animer la sphère : Faites tourner (à l'aide d'une variable intermédiaire) cette sphère sur le cercle de rayon 4, centré en (0,0,5) et situé sur le plan (Oxy). Pour ce faire, vous utiliserez deux méthode, une avec une seule translation (la plus simple à priori), l'autre avec une translation, une rotation et une nouvelle translation.

Exercice 03 – Zoomer

Faites en sorte que l'on puisse zoomer et dézoomer dans la scène. Pour ce faire, vous devez modifier la variable `dist_zoom` à l'aide de deux touches du clavier (de votre choix).

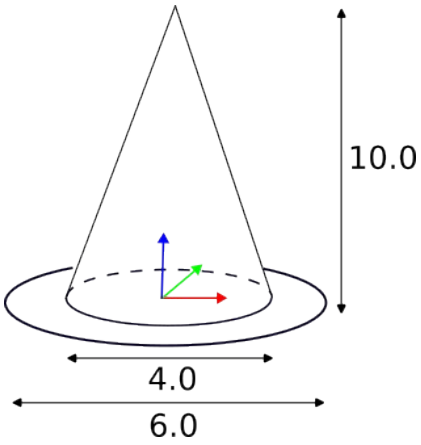
Note : Une bonne manière de modifier la valeur du zoom est de le multiplier par lui même par une valeur comme 0.9 (pour diminuer) ou 1.1 (pour augmenter).

Exercice 04 – Dessin de la balance

Nous allons passer à la balance. Celle-ci est constituée globalement de trois partie, la base, le bras balancier, et les plateaux.

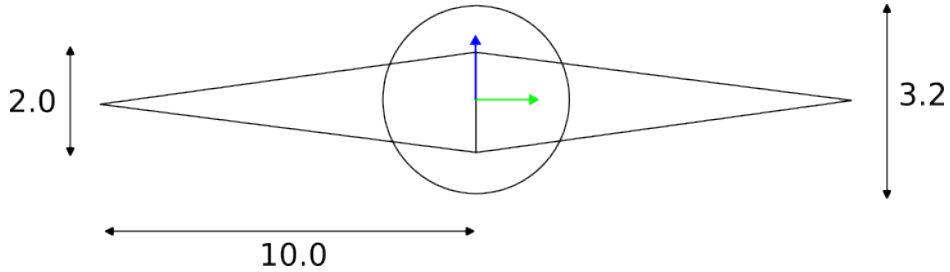
A faire :

01. La base : Complétez la fonction `drawBase` dans le fichier *draw_scene* afin de représenter la base de la balance définie dans son repère par un disque et un cône comme ci-dessous (se fier aux chiffres et pas à la forme globale du schéma) :



Pour ce faire, construisez (dans `initScene`) un disque (avec une shape 2D) et utilisez / créez l'objet canonique du cône (**avec une hauteur et un rayon de 1.0**). N'oubliez pas d'enregistrer le cône avec la fonction `createVAO`. Fixez la couleur de cette base à [235,207,52] (en range 0...255 bien sûr et donc à convertir). Vous appellerez ensuite cette fonction dans la fonction de dessin de la scène.

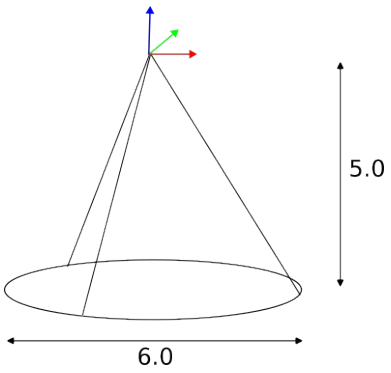
02. Le balancier : Complétez la fonction `drawArm` dans le fichier *draw_scene* afin de représenter le bras de balancier constitué d'une sphère et de deux cônes ainsi :



La couleur du bras est [245,164,66]. **Attention, vous devez reprendre la sphère et le cône canonique et non pas créer plusieurs sphères et plusieurs cônes**. Faites attention aussi aux axes (notamment l'axe y)

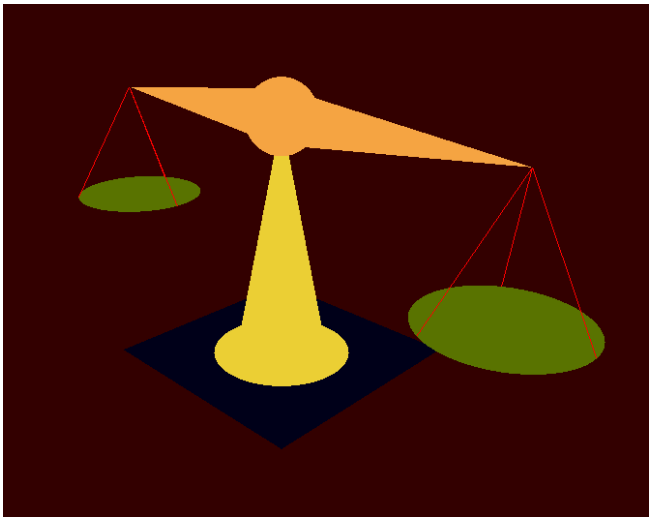
Ensuite, à l'aide de cette fonction, dessinez le bras au sommet du cône de la base précédente (dans la fonction `drawScene` du coup).

03. Les plateaux : Commencez tout d'abord par implémenter la fonction `drawPan` dans le fichier *draw_scene* afin de représenter les plateaux constitués simplement de 3 lignes et d'un disque comme ceci :



Les lignes sont rattachées au plateau tous les deux Pi sur trois, mais leur position de rattachement sur le cercle importe peu. La couleur des lignes est rouge [255,0,0] et celle du plateau [89,115,0].

Une fois réalisée cette fonction (et visuellement testée), il faut accrocher, donc dessiner, les deux plateaux au bout des deux bras du balancier. Vous devriez avoir quelque chose comme ceci :



Exercice 05 – Animation de la balance

Il nous reste à faire, d'une part pivoter la balance sur son axe. Et d'autre part, faire bouger les balancier de haut en bas.

A faire :

01. Rotation de la base : Modifiez votre dessin de scène globale afin de permettre à la balance de faire une rotation sur son axe, l'axe (Oz) en l'occurrence. Créez une variable `flag_anim_rot_scale` qui déclenchera ou arrêtera la rotation à l'appui de la touche 'r'.

02. Rotation du balancier : On souhaite maintenant faire monter et baisser le balancier d'un coté puis de l'autre, afin que les plateaux aient un mouvement de haut en bas, et cela grâce à une rotation du balancier de -20° à +20°. Bien évidemment, il faut que les plateaux restent bien horizontaux, eux.

Modifiez votre dessin de scène globale afin de permettre cette rotation dont le déclenchement dépendra d'une variable `flag_anim_rot_arm` déclenchée à l'appui de la touche 't'