TP OpenGL 2
Transformations Géométriques

Licence Informatique 3ème année

Année 2020-2021

Ce TP sera à rendre sous forme d'une archive contenant les sources et le fichier de configuration `cmake`, soit sur discord, soit à l'adresse email `christophe.renaud@univ-littoral.fr`. Le nom de l'archive devra avoir la forme suivante : `tp2XXX.zip`, avec `XXX` remplacé par votre nom de famille. Le TP est à réaliser seul et toute copie de code sera sanctionnée.

1 Introduction

Dans ce TP nous allons étudier la façon d'effectuer les transformations géométriques classiques sous OpenGL.

2 Affichage 3D en OpenGL

OpenGL utilise 3 transformations géométriques successives pour afficher sur l'écran le contenu d'une scène tridimensionnelle : le déplacement des objets à la position que leur attribue le programme ; la projection des objets sur l'écran et enfin leur affichage dans la zone écran qui est attribuée à l'image (transformation en pixels). Dans ce qui suit nous allons rappeler le principe de chacune de ces opérations et détailler la façon de l'exécuter en OpenGL.

2.1 Gérer la position des objets

2.1.1 Utilité

Pouvoir gérer la position des objets au sein de la scène à visualiser est évidemment très important. D'une part du fait que l'on construit en général les objets en précisant leur géométrie par rapport à l'origine du repère (ce qui est plus simple). Lorsque l'objet est construit, on doit ensuite pouvoir le déplacer jusqu'à l'emplacement que l'on souhaite, en changeant éventuellement son orientation et sa taille. D'autre part, lorsqu'on souhaite animer des objets, il faut là aussi pouvoir modifier leur emplacement et leur orientation en fonction du mouvement que l'on souhaite leur affecter.

2.1.2 Principe sous OpenGL

Par défaut, OpenGL considère que l'observateur est positionné à l'origine du repère global, de coordonnées $(0, 0, 0)$. De manière habituelle, on définit les objets par rapport à cette même origine. Il est donc nécessaire de déplacer soit les objets, soit l'observateur, lorsque l'on souhaite voir la scène!!!

En OpenGL on considère que l'observateur est immobile et que c'est la scène et tous les objets qui la constituent qui bougent. L'observateur est toujours positionné à l'origine et regarde selon l'axe Oz dans la direction des z négatifs (voir figure 1).

Lorsque l'on placera ou déplacera des objets dans la scène, il faudra donc tenir compte de ce paramètre.

2.1.3 Fonctions OpenGL

Pour placer un objet, OpenGL utilise une matrice de transformation géométrique fournie par l'utilisateur. Cette matrice devra contenir toutes les transformations à appliquer sur un objet afin de connaître sa position finale. Comme chaque objet peut disposer de ses propres transformations géométriques, OpenGL définit une « matrice courante de transformation » qui doit être initialisée pour chaque objet à déplacer. Notons, sans entrer dans les détails pour le moment, qu'OpenGL gère en fait plusieurs matrices dites « courantes » . Lorsque l'on souhaite utiliser la matrice dédiée aux transformations géométriques, il est

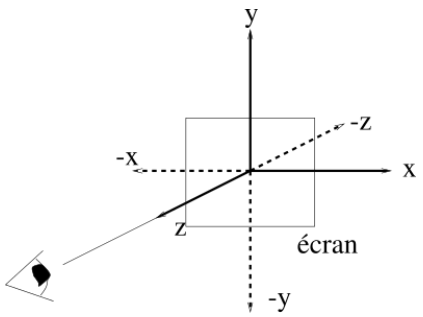


FIGURE 1 – Vue du repère observateur sous OpenGL

nécessaire de prévenir OpenGL que c'est cette matrice là qui doit être utilisée. Cette matrice est définie par l'intermédiaire de la constante `GL_MODELVIEW` et elle est rendue « active » par la fonction suivante :

```
void glMatrixMode(GLenum mode);
```

qui permet de spécifier que la matrice « `mode` » devient la matrice courante. En utilisant pour `mode` la valeur `GL_MODELVIEW`, on précise à OpenGL que l'on souhaite utiliser la matrice courante de transformation.

Quatre autres fonctions communes, dans la cadre des matrices de « déplacement » des objets, sont décrites ci-après :

- `void glLoadIdentity(void)` : initialise la matrice courante avec la valeur de la matrice identité ;
- `void glTranslatef(float tx, float ty, float tz)` : multiplie la matrice courante par une matrice qui représente une translation de vecteur $t = (t_x, t_y, t_z)$;
- `void glRotatef(float angle, float x, float y float z)` : multiplie la matrice courante par une matrice qui représente une rotation d'angle `angle` autour de l'axe (x, y, z) ;
- `void glScalef(float kx, float ky, float kz)` : multiplie la matrice courante par la matrice qui représente le changement d'échelle qui multiplie respectivement par kx, ky, kz les coordonnées x, y, z de tous les points de l'espace.

2.1.4 Exemple

Les instructions ci-après permettent de sélectionner la matrice correspondant au repère du modèle, de l'initialiser avec la matrice identité et enfin de lui appliquer une translation de -2.0 selon l'axe des Z.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef(0.0, 0.0, -2.0);
```

À l'issue de leur exécution, l'observateur et le repère associé au modèle sont dans les positions schématisées sur la figure 2. On rappelle que l'observateur reste en $(0, 0, 0)$ et que c'est le point O' de la figure 2 qui se trouve en $(0, 0, -2)$.

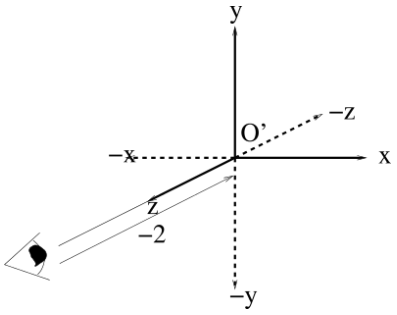


FIGURE 2 – Effet d'une translation d'axe Oz de -2 unités

2.2 Projection sur l'écran

Lorsque les objets ont été disposés dans la scène, il faut passer à la seconde étape du processus de visualisation : la projection sur l'écran pour transformer les objets 3D en objets 2D. OpenGL implante les

2 projections classiques que sont la projection perspective et la projection orthogonale. Avant de définir la projection à utiliser et donc sa matrice, il est nécessaire de préciser à OpenGL de rendre active la matrice courante utilisée pour gérer ces projections. Cela se fait par l'appel suivant :

```
glMatrixMode(GL_PROJECTION);
```

Notez que comme la valeur de cette matrice peut être quelconque au moment de sa sélection, il est conseillé de la réinitialiser avec la matrice identité.

2.2.1 Projection perspective

La fonction suivante multiplie la matrice courante par une matrice de projection perspective :

```
void gluPerspective(float angle, float aspect,
                    float distance_ecran,
                    float distance_max);
```

avec

- `angle` : angle d'ouverture de la pyramide de vision ;
- `aspect` : aspect de la section de la pyramide ($= \frac{largeur}{hauteur}$). Une valeur de 1 ne provoque aucune distortion ;
- `distance_ecran` : distance entre l'observateur et l'écran ; tout objet situé devant le plan écran ne sera pas visualisé ;
- `distance_max` : distance maximale au delà de laquelle les objets sont considérés comme invisibles.

Notez que les objets de la scène sont « clippés » par rapport à la pyramide de vision, afin d'éviter de traiter les objets (ou parties d'objets) invisibles.

2.2.2 Projection orthogonale

La fonction suivante multiplie la matrice courante par une matrice de projection orthogonale :

```
void glOrtho(GLdouble gauche, GLdouble droite,
             GLdouble bas, GLdouble haut,
             GLdouble distance_ecran,
             GLdouble distance_max);
```

avec

- `gauche` : tous les points dont la coordonnée x sera plus petite que cette valeur ne seront pas affichés ;
- `droite` : tous les points dont la coordonnée x sera plus grande que cette valeur ne seront pas affichés ;
- `bas` : tous les points dont la coordonnée y sera plus petite que cette valeur ne seront pas affichés ;
- `haut` : tous les points dont la coordonnée y sera plus grande que cette valeur ne seront pas affichés ;
- `distance_ecran` :distance entre l'observateur et l'écran ; tout objet situé devant le plan écran ne sera pas visualisé ;
- `distance_max` :distance maximale au delà de laquelle les objets sont considérés comme invisibles.

Notez que les objets de la scène sont « clippés » par rapport au parallélépipède de vision, afin d'éviter de traiter les objets (ou parties d'objets) invisibles.

2.3 Définir une zone d'affichage

Il reste enfin à preciser la zone de la fenêtre dans laquelle l'image doit être affichée. Par défaut, toute la fenêtre est utilisée. Il est cependant possible de restreindre l'affichage à une « sous-zone » de la fenêtre, en fonction de ce que l'on souhaite faire (affichages multiples, aspect de l'image, etc ...). La fonction OpenGL permettant de préciser la zone d'affichage est la suivante :

```
void glViewport(GLint xbas, GLint ybas,
                GLsizei largeur, GLsizei hauteur);
```

Les différents paramètres, dont la valeur est exprimée en pixels, représentent les coordonnées du coin inférieur gauche du rectangle d'affichage, suivies de sa largeur et de sa hauteur.

2.4 Le tampon de profondeur ou Z-buffer

Pour indiquer à OpenGL qu'il doit effectuer la suppression des parties cachées on utilise les instructions suivantes dans la fonction `main` :

```
glutInitDisplayMode(GLUT_DEPTH);
glEnable(GL_DEPTH_TEST);
```

La fonction `glutInitDisplayMode` est appelée avant la création de la fenêtre (appel de la fonction `glutCreateWindow`).

La fonction `glEnable` est elle appelée avant l'appel de la fonction `glutMainLoop`.

De plus, dans la fonction `dessiner`, au moment d'effacer l'écran, on utilise la fonction `glClear` avec le paramètre `GL_DEPTH_BUFFER_BIT` combiné aux autres paramètres éventuels par un `ou` logique. On écrira par exemple

```
glCear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

pour effacer l'écran et donner à chaque pixel la plus grande profondeur dans le tampon de profondeur (OpenGL élimine les parties cachées en utilisant la technique du Z-buffer).

2.5 Application

2.5.1 Préliminaires

Récupérez le fichier `Tp2.zip` et recopiez-le dans le répertoire dans lequel vous effectuez vos TP d'informatique graphique. Désarchivez ce fichier ; à l'issue de cette opération, un répertoire nommé `Tp2` a été créé, contenant un fichier de configuration pour l'utilitaire `cmake` et un sous-dossier `Src` contenant les fichiers suivants :

- `tp3d.c` : le fichier principal, contenant les instructions de création et d'initialisation de l'environnement graphique ;
- `graphique.c` : ce fichier contient la fonction `dessiner` qui sera appelée pour chaque affichage, la définition de la fonction `cube(float dim)` qui permettra de tracer un cube de côté `dim`, centré à l'origine et enfin une fonction `repere(float lg)` qui permet de tracer les axes du repère sur une longueur de `lg` dans chacune des 6 directions de celui-ci ;
- `graphique.h` : qui contient simplement le prototype de la fonction `dessiner`.

2.5.2 Exercice 1

1. Complétez la fonction `init_screen` du fichier `tp3d.c` de manière à ce qu'elle :
   - définisse une projection perspective d'ouverture 60, de rapport d'aspect unitaire et pour laquelle les seuls objets visibles seront ceux situés à une distance appartenant à $[1,\ 50]$ (pensez bien à sélectionner la matrice de projection dans le contexte graphique ...) ;
   - définisse une zone d'affichage occupant toute la fenêtre.
2. Modifiez la fonction `dessiner` pour que le repère du modèle soit déplacé de 5 unités selon la direction $-Z$ (pensez bien à sélectionner la matrice de transformation (changement de repère) dans le contexte graphique ...) ;
3. Compilez votre application et vérifiez que, lors du test, vous obtenez bien une croix blanche centrée au milieu de la fenêtre (figure 3.a).
4. Complétez la fonction `cube` de telle sorte qu'elle dessine un cube rouge, de côté `dim`, centré à l'origine. Vous modifierez ensuite la fonction dessiner, de telle sorte qu'elle trace un cube de côté 2 après le dessin du repère. Lors de l'exécution de l'application, vous devez voir apparaître un carré rouge centré dans la fenêtre (figure 3.b).

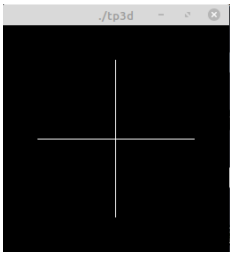
2.5.3 Exercice 2

Modifiez la fonction `dessiner` de manière à ce qu'elle :

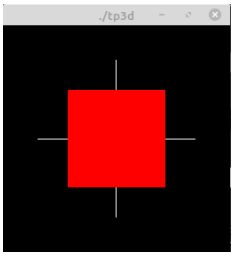
- conserve les mêmes transformations que précédemment ;
- y ajoute une rotation de 20^o autour de l'axe Oy (axe vertical).

Lors de l'exécution de votre programme, le carré rouge doit avoir tourné légèrement vers la droite (voir figure 4.a). Pour mieux percevoir les différentes faces, modifiez le code de la fonction `cube` de telle sorte que chacune des faces ait la couleur demandée ci-dessous :

- face avant ($z > 0$) : rouge ;
- face arrière($z < 0$) : blanc ;



(a)

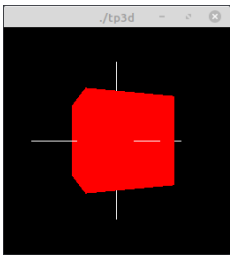


(b)

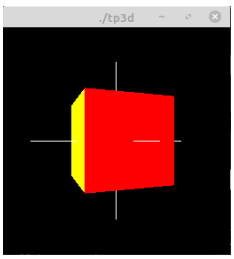
FIGURE 3 – Aperçu du repère vu selon la direction Oz (a) et de la face avant du cube rouge vu selon la même direction (b)

— face droite ($x > 0$) : vert ;
— face gauche($x < 0$) : jaune ;
— face supérieure ($y > 0$) : cyan ;
— face inférieure($z < 0$) : magenta.

Lors de l'exécution de l'application, la face gauche du cube, de couleur jaune, doit apparaître sur la gauche de la fenêtre (voir figure 4.b).



(a)



(b)

FIGURE 4 – Aperçu du cube après rotation (a) et changement de couleur des faces (b)

2.5.4 Exercice 3

Lors de l'exécution de votre programme, modifiez (**avec la souris**) la taille de la fenêtre d'affichage, de telle sorte qu'elle ait une forme rectangulaire. Vous notez alors que votre cube s'affiche sous forme d'un parallélépipède (voir figure 5.a). Cela est dû au fait que, par défaut, la zone d'écran utilisée pour l'affichage correspond à toute la zone gérée par la fenêtre. Ce qui implique que si cette fenêtre est déformée (par rapport au carré idéal), les objets qui s'y trouvent affichés le sont aussi. Il est possible de modifier cela en définissant une fonction qui sera systématiquement appelée lorsque la taille de la fenêtre sera modifiée **dynamiquement**.

Exemple :

1. écrire dans le fichier `graphique.c` la fonction suivante :

```
void retailler(GLsizei largeur, GLsizei hauteur)
{
  glViewport(0, 0, 50, 50);

  glutPostRedisplay();
}
```

   L'exécution de cette fonction aura pour effet de spécifier que la zone d'affichage se trouvera désormais dans la « sous-fenêtre » carrée de taille 50 (largeur et hauteur) et dont le coin inférieur gauche sera situé $(0, 0)$, dans la fenêtre de taille *largeur* × *hauteur* pixels. Les dimensions de cette « sous-fenêtre » étant les mêmes en hauteur et en largeur, les objets affichés ne seront pas déformés ; évidemment, pour cet exemple, ils seront plutôt ... petits !!! Notez que l'appel à la fonction `glutPostRedisplay` est ici obligatoire pour regénérer l'affichage.

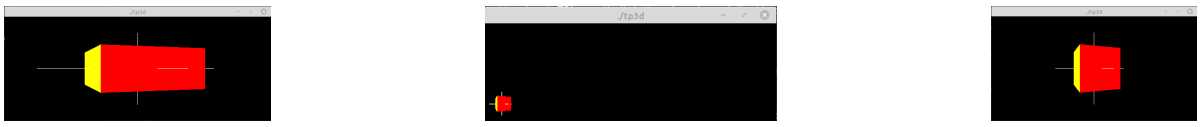
2. modifiez le fichier `graphique.h` en y rajoutant la ligne :

```
void retailler(GLsizei largeur, GLsizei hauteur);
```

3. ajoutez dans la fonction `main` du fichier `tp3d.c` l'appel de fonction suivante :

   ```
   glutDisplayFunc(dessiner); /* deja present !!! */
   glutReshapeFunc(retailler);
   ```

   La fonction `glutReshapeFunc` a le même rôle pour la déformation de la fenêtre que la fonction `glutDisplayFunc` pour une modification du contenu de celle-ci : elle définit la fonction qui doit être appelée lorsque la taille de la fenêtre est modifiée en cours d'exécution du programme. Ici la fonction appelée sera la fonction `retailler`, le système lui fournissant automatiquement les nouvelles dimensions de la fenêtre dans ses paramètres `largeur` et `hauteur` ;
4. compilez et testez ces modifications. Vous devez obtenir une fenêtre se comportant comme celle de la figure 5.b.



(a) (b) (c)

FIGURE 5 – Déformation du cube sans gestion dynamique de la zone d'affichage (a) utilisation d'une zone 50×50 pour la zone d'affichage (b) et centrage dynamique de la zone d'affichage (c).

Question : Modifiez la fonction `retailler` de telle sorte que l'affichage se fasse toujours dans une zone **carrée** de la fenêtre existante, cette zone ayant les propriétés suivantes :
— elle est centrée dans la fenêtre ;
— elle occupe le maximum d'espace disponible.
Un aperçu du résultat à obtenir est fourni sur la figure 5.c, dans laquelle la fenêtre à été agrandie horizontalement.

3 Déplacement d'un objet 3D

Dans cette partie, vous allez voir comment appliquer **dynamiquement** différentes transformations géométriques pour déplacer un objet. Chacune de ces transformations sera appliquée à notre cube lors de l'appui de certaines touches qui leurs seront associées.

3.1 Translation du cube selon Oz

Vous allez développer ici le code nécessaire à la gestion des 2 opérations suivantes :
— une translation du cube selon l'axe Oz, dans la direction $-Z$, lorsque la touche `-` sera appuyée ;
— une translation du cube selon l'axe Oz, dans la direction $+Z$, lorsque la touche `+` sera appuyée ;
Pour cela vous allez suivre pas à pas les étapes détaillées ci-dessous :

1. créer un fichier `touches.c` et y définir une variable **globale** réelle nommée `trans_axeZ` ;
2. y ajouter le code de la fonction suivante :

   ```
   void gerer_clavier(unsigned char touche, int x, int y)
   ```

   Cette fonction aura pour effet d'incrémenter de 1.0 la valeur de `trans_axeZ` si le paramètre `touche` a pour valeur le caractère `'+'` et de décrémenter la valeur de cette variable de 1.0 si ce paramètre a pour valeur le caractère `'-'`. Elle appelle aussi la fonction `glutPostRedisplay` pour redessiner l'objet avec la nouvelle matrice ;
3. créer le fichier `touches.h` et y faire les déclarations externes de `gerer_clavier` et de `trans_axeZ` ;
4. incluez le fichier `touches.h` dans le fichier `graphique.c` pui modifiez la fonction `dessiner` pour qu'elle prenne en compte la translation définie par `transAxeZ` lors de l'appel à la fonction `glTranslatef` déjà présent ;
5. modifier le fichier `tp3d.c` de manière :
   — à y inclure `touches.h` ;
   — à y définir la fonction `gerer_clavier` comme fonction à appeler en cas d'événement clavier ;
   — à y initialiser `trans_axeZ` dans la fonction `init_screen` (utilisez la valeur de tanslation de -5 précédemment utilisée ;

Après avoir modifié le fichier `CMakeLists.txt` et regénéré la fichier `Makefile`, compilez et testez cette nouvelle fonctionalité.

3.2 Rotation du cube autour de Oy

Vous allez compléter la fonction `gerer_clavier` de telle manière qu'elle gére les 2 touches **supplémentaires** suivantes :

— `'6'` : rotation de $+10^o$ du cube autour de l'axe Oy ;
— `'4'` : rotation de -10^o du cube autour de l'axe Oy.

Pour ce faire vous définirez une nouvelle variable globale réelle dans `touches.c`, nommée `angle_rotY`. Cette variable devra être initialisée à 0.0 dans la fonction `init_screen`.

Vous modifierez la fonction `dessiner` de manière à ce qu'elle prenne en compte cette rotation autour de l'axe Oy.

3.3 Rotation du cube autour de Ox

Même question mais en gérant cette fois les rotations autour de l'axe des x à l'aide des touches `'8'` (-10^o) et `'2'` ($+10^o$).

3.4 Déformation du cube

Complétez votre application de telle manière qu'elle gére les six touches **supplémentaires** suivantes :

— `gauche` : divise par 2 les coordonnées x et `droite` : multiplie par 2 les coordonnées x ;
— `bas` : divise par 2 les coordonnées y et `haut` : multiplie par 2 les coordonnées y ;
— `page précédente` : divise par 2 les coordonnées z et `page suivante` : multiplie par 2 les coordonnées z ;

Vous définirez trois nouvelles variables globales réelles dans `touches.c`, nommées respectivement `kx`, `ky` et `kz`. Ces trois variables devront être initialisées à 1.0 dans la fonction `init_screen`.

Vous modifierez bien entendu la fonction `dessiner` pour que toutes ces transformations soient prises en compte.