Introduction à OpenGL – TD

Malek.Bengougam@gmail.com

1. **Primitives**

1.1 Attributs des primitives.

Il est possible de spécifier des attributs par primitive ou bien par sommet. En OpenGL classique (1.0 à 1.5) ces attributs sont fixes et limités à ce que l'API -et le matériel- sont/étaient capable de gérer.
On dispose par exemple de couleurs, que l'on peut spécifier à l'aide de glColor***() et ses variantes, de coordonnées de textures -glTexCoords**(), de normales etc…

Lorsque l'on souhaite spécifier des couleurs, nous avons le choix entre des valeurs dont les intensités sont définies sur une plage non-signée en octet par exemple (de 0 à 255) ou en réel (de 0.f à 1.f) le plus souvent.
Les fonctions s'adaptent ici au type de donnée et au nombre de composantes. Ainsi pour 3 composantes R, V, B et un type de donnée en octet (unsigned byte) on utilise la variante **glColor3ub**(). Pour des floats on utilise plutôt **glColor3f**().

Exercice 1.1. Spécifiez des couleurs distinctes pour chaque sommet (par exemple, rouge pour le premier, vert pour le second, et bleu pour le troisième).

Remarque : Cet exercice permet de révéler un aspect important du matériel : la « rasterisation » ou conversion en ligne de la primitive. On remarque la présence de gradient de couleur ce qui indique une interpolation linéaire des valeurs des attributs de sommet.

1.2 Orientation des primitives.

Par défaut, OpenGL considère qu'une primitive est visible (ou 'avant') si ses sommets (vertices) sont spécifiés dans l'ordre inverse des aiguilles d'une montre (*counter-clockwise*). Cependant, ceci n'a vraiment d'impact que lorsque la fonctionnalité de suppression des faces cachées (ou 'arrières') est activée.
Pour cela on utilise la fonction **glEnable**() avec comme paramètre **GL_CULL_FACE**.

Exercice 1.2. Activez le 'face culling' et essayez d'inverser les sommets 2 et 3 de notre triangle.

2. Fenêtrage

La fonction **glViewport**() permet de créer une correspondance entre le repère en pixel de la fenêtre et le repère orthonormé d'OpenGL (NDC). Cette fonction prend 4 paramètres : les deux premiers sont la position du viewport, et les deux derniers correspondent aux dimensions du viewport.
Attention, l'origine du repère de la fenêtre est située en bas à gauche de la fenêtre.

Les fonctions de dessin opèrent donc dans les limites du viewport, mais parfois il peut être utile de limiter encore plus le rendu. La fonction **glScissor**() permet de spécifier une zone rectangulaire (*mêmes paramètres que glViewport()*) limitant l'écriture des pixels à cette portion.
Cette fonction correspond à une fonctionnalité matérielle qui doit être activée par glEnable() : il s'agit du test de découpage et le paramètre est **GL_SCISSOR_TEST**.

Exercice 2.1. Créez 4 viewports de sorte à séparer la fenêtre en 4 parties distinctes. Chacun des viewports est effacé avec une couleur de fond différente. Affichez des primitives différentes dans chaque viewport (par ex, un triangle dans le viewport en haut à gauche, un 'triangle-strip' dans un autre, une ligne dans le troisième etc..

Note : **l'ordre des appels de fonctions est important**. Scissor et Viewport doivent être avant le dessin (clear...).

3. Transformations

3.1. Matrices et transformation

OpenGL utilise des matrices 4 lignes x 4 colonnes afin de représenter les transformations affines et linéaires. On parle de coordonnées homogènes (en 4 dimensions donc ici) car on utilise la même représentation pour tout type de transformation.

Les fonctions **glTranslate***(), **glScale***(), **glRotate***() permettent respectivement d'ajouter une transformation de type translation (déplacement), homothétie (facteur d'échelle) et rotation (angle en degrés, et axe de rotation).
On remplace généralement * ici par d (pour *double*) ou f (pour *float*) selon le type de donnée souhaité.

Il est possible de chaîner les appels à ces fonctions afin de produire une transformation du repère local de l'objet vers le repère de la scène (ou de la caméra). OpenGL dispose d'un mode dédié à ce type de matrice que l'on active avec la fonction **glMatrixMode**() et en spécifiant ici **GL_MODELVIEW**. On dispose alors d'une matrice transformant les sommets des primitives de leur repère local (model) vers la camera (view).

Important : La fonction **glLoadIdentity**() permet de remplacer la matrice courante par la matrice identité.

Les transformations s'exécutent dans l'ordre inverse des appels de fonction. Ceci parce qu'OpenGL suit la convention mathématique où les vecteurs sont des colonnes et les multiplications s'effectuent de la droite vers la gauche. Exemple : v' = T . R . S . v. Ici le vecteur v est d'abord multiplié à la matrice S avant que le résultat soit ensuite multiplié à R puis à T. Il est également possible d'effectuer v' = (T.R.S) . v ce qui produit le même résultat. Rappelez-vous cependant que les multiplications ne sont pas commutatives ici.

Exercice 3.1.a. Utilisez la fonction `glRotatef(45.f, 0.f, 0.f, 1.f)` où 45.f est une valeur angulaire en degrés, et (0,0,1) correspond à l'axe de rotation. Que remarquez-vous ?

Exercice 3.1.b. Utilisez `glLoadIdentity()` de sorte à ce que le triangle soit tourné mais fixe.

Exercice 3.1.c. Utilisez les fonctions `glScalef()`, `glTranslatef()` et `glRotatef()` de sorte à ce que le triangle soit 1.5 fois plus grand, tourné vers la gauche -selon l'axe avant- et positionné sur la droite de l'écran.

3.2. Animation

Dans cet exercice nous allons utiliser le temps écoulé depuis le début du programme comme un facteur d'animation. La fonction **glutGet**() permet de récupérer cette information en spécifiant **GLUT_ELAPSED_TIME**. Le temps retourné par cette fonction est exprimé en millisecondes, il faut donc le diviser par 1000.0 (en float ou double) afin de convertir le temps écoulé en secondes.

Exercice 3.2.a Utilisez le temps écoulé comme un facteur de l'angle de rotation, comme un facteur d'homothétie (*scale*) ou comme un facteur de déplacement.

Exercice 3.2.b. Plutôt que d'utiliser

3.3. Pile matricielle

Il est possible de sauvegarder l'état de la concaténation des matrices à n'importe quel instant en utilisant la fonction **glPushMatrix**(). Pour restituer l'état sauvegardé (empilé) ou utilise alors **glPopMatrix**().

Exercice 3.3. Sans utiliser `glLoadIdentity()`, affichez deux triangles, l'un à sa position ortho-centrée (comme dans l'exercice 1.1) et l'autre orienté comme dans l'exercice 3.2.

3.4. Matrices custom

A la manière de `glLoadIdentity`(), il est possible de charger une matrice de manière explicite en utilisant la fonction **`glLoadMatrixf`**(), ou `glLoadMatrixd`() pour des réels double précision.
Le paramètre est un tableau de réels (*float* ou *double* selon la fonction) et ce tableau est composé de 16 valeurs, soit une matrices 4 colonnes x 4 lignes.

$$\begin{pmatrix} c[0] & c[4] & c[8] & c[12] \\ c[1] & c[5] & c[9] & c[13] \\ c[2] & c[6] & c[10] & c[14] \\ c[3] & c[7] & c[11] & c[15] \end{pmatrix} \times \begin{pmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{pmatrix}$$

Attention cependant, `glLoadMatrixf`() remplace la matrice courante !
Si vous souhaitez au contraire concaténer une nouvelle matrice alors dans ce cas on utilise plutôt **`glMultMatrixf`**() qui s'insère dans la pile des matrices.
Un appel à glMultMatrixf() insère donc une matrice au sommet de la pile. Si la matrice courante est la matrice C et que l'on ajoute la matrice M le calcul est alors le suivant : C.M.v, et donc M est appliquée avant C.

$$\begin{pmatrix} c[0] & c[4] & c[8] & c[12] \\ c[1] & c[5] & c[9] & c[13] \\ c[2] & c[6] & c[10] & c[14] \\ c[3] & c[7] & c[11] & c[15] \end{pmatrix} \times \begin{pmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{pmatrix} \times \begin{pmatrix} v[0] \\ v[1] \\ v[2] \\ v[3] \end{pmatrix}$$

Exercice 3.4. Remplacez les appels à `glRotatef`(), `glScalef`() et `glTranslatef`() par des matrices en utilisant `glLoadMatrixf`() et/ou `glMultMatrixf`().

4. Projections

Le repère normalisé d'OpenGL n'est pas pratique pour représenter des scènes 2D ou 3D car les unités sont restrictives et difficiles à manipuler.
Il serait plus idéal de raisonner en pixels ou en unités spécifiques (mètres, pieds, ou autre unité arbitraire).

OpenGL offre un mode matriciel pour de telles matrices, il s'agit du paramètre **`GL_PROJECTION`** de **`glMatrixMode`**(). Là également nous pouvons utiliser des combinaisons de glTranslate/Rotate/Scale* afin d'obtenir une conversion des unités de notre choix vers le repère orthonormé.

OpenGL propose les fonctions **`glOrtho`**() et **`glFrustum`**() pour définir des volumes de projections mais leur usage n'est pas vraiment aisé.
Nous allons plutôt utiliser des fonctions prédéfinies dans <GL/GLU.h> (GL Utility) afin de nous faciliter la vie.
Les fonctions qui nous intéressent sont **`gluOrtho2D`**() et **`gluPerspective`**().

Note : il faut maintenant également linker avec `-lglu32` sous Windows (Mingw etc..) ou `-lglu` sous Linux.

Exercice 4.1.a. Utilisez **`gluOrtho2D`**() afin de définir un repère dont l'origine se trouve en bas à gauche de l'écran et dont les unités sont en pixels (1 unité = 1 pixel du viewport ou de la fenêtre).
Il sera sans doute nécessaire d'appliquer un facteur d'échelle à vos primitives afin qu'elles soient visibles.

Exercice 4.1.b. Utilisez **`gluPerspective`**() afin de créer une projection en point de fuite et donner l'illusion d'un monde en 3D. Les paramètres 'near' et 'far' correspondent aux distances des plans de coupe au proche de la caméra (near) et au loin (far) de la caméra. Les valeurs usuelles sont respectivement 0.1f et 1000.f.
Il sera sans doute nécessaire de déplacer vos primitives en profondeur (dans l'axe -Z) afin que celles-ci soient visibles.

5. Caméra

Du point de vue d'OpenGL, la notion de caméra n'existe pas. C'est un concept purement virtuel qui consiste en pratique à déplacer les objets de sorte à ce que l'origine (la base mathématique) soit le repère de l'objet caméra. C'est donc un changement de repère où le repère local est celui de la caméra.

Tout ceci peut se faire avec des successions de `glRotatef`() et `glTranslatef`() mais il est important de tenir compte du fait qu'il s'agit d'une transformation inverse : on passe en effet du repère de travail (WORLD) vers un repère local, ce qui est précisément l'inverse de ce que l'on fait habituellement.
La pile matricielle GL_MODELVIEW est destinée à transformer les objets dans le repère de la caméra (donc ne pas utiliser GL_PROJECTION). Il faut juste s'assurer qu'une fois nos objets positionnés et orientés dans la scène, les transformations de caméra soient bien appliquées dans l'ordre inverse.

Alternativement on peut calculer l'inverse de chaque matrice et les ajouter sur la pile de matrice.
A noter qu'une transformation de caméra est généralement composée seulement de translation et rotations, et qu'il est possible de simplifier les calculs en tenant compte de certains aspects (ou hack mathématiques).
Ici nous allons nous contenter d'utiliser une fonction de glu qui gère une forme de caméra classique en infographie, une look-at.

La fonction **`gluLookAt`**`(pos.x, pos.y, pos.z, target.x, target.y, target.z, up.x, up.y, up.z)` permet de spécifier :

- La position de la caméra (pos.xyz)
- La position du regard (target.xyz, attention c'est une position, pas une direction)
- La direction du vecteur 'up' (haut) donnant l'orientation de la scène.

Exercice 5.1. Utilisez cette fonction (en tout premier dans la pile) afin de simuler une caméra regardant votre scène depuis un point de vue haut.

6. Objets 3D

Maintenant qu'il vous est possible de représenter un point de vue en perspective, essayez d'afficher un objet simple en 3D, tel un cube par exemple, mais toujours en utilisant le mode de rendu GL_TRIANGLES.

Rappel : un cube est composé de 8 vertices et 6 faces rectangulaires, chacune scindée en deux triangles ici.