Modélisation et Synthèse d'Image
TP2 : Transformations et Indexation

Introduction

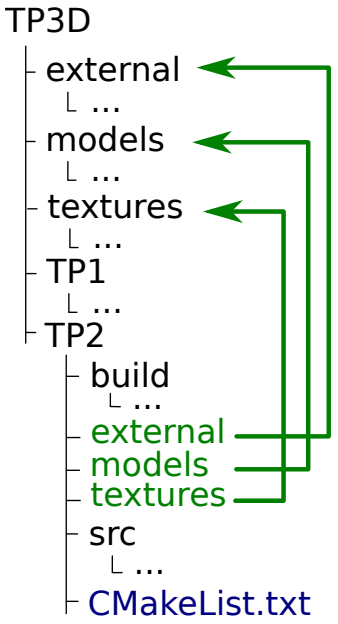
L'objectif du TP est de construire un outil de visualisation de modèle 3D simple permettant la navigation dans une scène 3D. La base de code qui vous est fournie reprend les éléments de la dernière séance : création de fenêtre GLFW et initialisation de contexte OpenGL, affichage de géométrie. Pour la récupérer, allez à l'adresse suivate : http://romain.vergne.free.fr/blog/?page_id=228.

Pour l'utiliser, faites comme la dernière fois :

- accédez au dossier TP3D (`cd ~/TP3D/`)
- récupérez/dézippez les fichiers externes et le TP2 :

```
wget http://maverick.inria.fr/~Romain.Vergne/SI-RICM4/TP2.zip
unzip TP2.zip && rm -f TP2.zip
```

- placez vous dans le répertoire TP2 (`cd TP2`)
- créez des liens symboliques vers les fichiers externes :
  - `ln -rs ../external/`
  - `ln -rs ../models/`
  - `ln -rs ../textures/`
- créez un dossier pour la compilation (`mkdir build`)
- accedez à ce dossier (`cd build`)
- lancez cmake (`cmake ..`)
- lancez la compilation (`make`)
- exécutez (`./polytech_ricm4_tp2`)



Voici la structure que devrait avoir votre arborescence de fichiers.

Note : La version d'*OpenGL* est maintenant la 3.0, et *GLSL* 150.

Pipeline de Transformation

Comment déplacer, tourner, changer l'échelle d'un modèle 3D ? Comment créer une modifier le point de vue dans la scene ? Ces éléments sont à la base de la création d'une scène virtuelle et même si leur utilisation est cachée par une interface utilisateur, il reste essentiel d'en connaître le fonctionnement.

Les coordonnées homogènes

C'est quoi ? Un point 3D est décrit par ses coordonnées (x, y, z). En informatique graphique, quand il s'agit d'utiliser OpenGL pour manipuler des points on utilise une quatrième coordonnées w. Les coordonnées (x, y, z, w) sont dites *homogènes*.

Pourquoi ? Les coordonnées homogènes permettent d'implémenter rotations, translations, changement d'échelle et projection *sous forme de matrices*. Elles ont 2 gros avantages :

1. Rapidité d'évaluation sur le GPU (les processeurs sont optimisés pour cela).
2. Simplicité d'utilisation (multiplier plusieurs matrices revient à combiner plusieurs transformations)

Attention : Le produit matriciel n'est pas *commutatif* : $RT \neq TR$. Pour appliquer une translation T *puis* une rotation R on utilise le produit RT.

Utile :

1. Si $w = 1$ alors $(x, y, z, 1)$ représente une position dans l'espace.
2. Si $w = 0$ alors $(x, y, z, 0)$ représente une direction (un vecteur).
3. Si $w \neq 0$ on peut projeter un point $p = (x, y, z, w)$ vers l'espace usuel en normalisant par w : $\tilde{p} = (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1)$

Exemple de translation

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix}$$

Sans Coordonnées Homogènes Avec Coordonnées Homogènes

Si $w = 1$ on a bien une translation représenté sous forme matricielle et pouvant être facilement combinée avec d'autres transformations via le produit matriciel.

Les différentes matrices

$$\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Changement d'échelle (homothétie) Translation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Rotation autour de l'axe x Rotation autour de l'axe y Rotation autour de l'axe z

D'un point 3D à l'écran

Le pipeline de *transformation* s'applique sur chacun des sommets d'un maillage (en général dans le vertex shader) et intervient avant la *rasterization*.

La transformation de l'objet représente la position de l'objet dans le monde : elle transforme un point de l'espace objet vers l'espace monde. Elle est représentée par la matrice **M** (comme modèle). Il faut une matrice objet pour chaque objet de la scène. Un modèle 3D est défini dans un système de coordonnées locales (l'espace objet) dont l'origine $(0, 0, 0)$ correspond au centre de l'objet. Lorsqu'il est déformé, les données le décrivant ne changent pas : c'est la transformation de l'objet qui rend compte de cette déformation (qui a lieu dans l'espace monde). Par exemple, dans un jeu le déplacement d'un personnage ou d'un objet est défini par cette transformation.

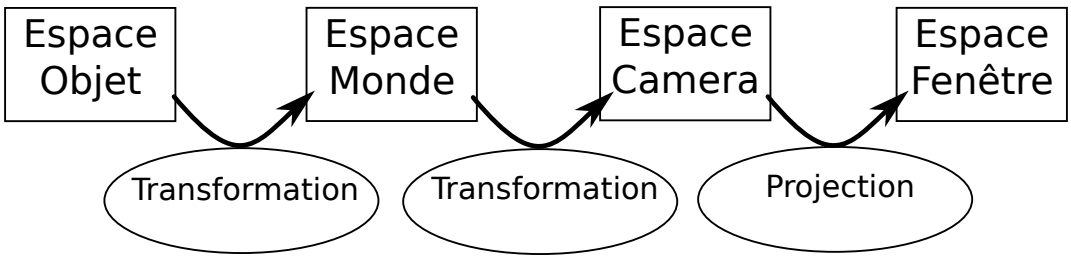


Figure 1: Pipeline de transformation (simplifié) d'OpenGL

La transformation de la caméra représente la position de la caméra dans le monde : elle transforme un point de l'espace monde vers l'espace caméra (c'est à dire l'espace objet de l'objet caméra). Elle est représentée par la matrice **V** (comme vue). Il n'y a qu'une seule matrice caméra dans la scène. C'est cette transformation qui rend compte du changement de point de vue dans la scène 3D. Ainsi, lorsqu'on se déplace dans une scène 3D, les coordonnées des objets dans l'espace monde ne changent pas.

La projection représente la prise de vue de la caméra : elle transforme un point de l'espace caméra vers l'espace fenêtre. Elle est représentée par la matrice **P** (comme projection). Il n'y a qu'une seule matrice de projection dans la scène. Cette transformation rend compte de la perspective induite par la caméra. Elle peut être vue comme une transformation changeant le frustrum (piramide tronquée) représentant le champ de vision de la caméra en cube unité.

En conclusion, quand on multiplie le point x par le produit matriciel $P \times V \times M$, on fait passer celui-ci depuis le repère objet (dans lequel il est décrit) au repère de la fenêtre (vers lequel il va être rendu). Cette décomposition en trois matrice permet de changer indépendemment et facilement tout les aspects de l'image à rendre : la position des objets, celle de la caméra, et les paramètres de prise de vue. À noter que conceptuellement, c'est toujous le monde qui est déformé pour rentrer dans le champ de vision de la caméra, qui elle reste fixe. Cette transformation est effectuée dans le vertex shader, c'est ce que nous allons voir maintenant.

Note : Le pipeline est un peu plus complexe que ce qui est décrit ici. Pour plus de précisions, voir : http://www.songho.ca/opengl/gl_transform.html.

Travail demandé

La première partie du TP consiste en deux étapes :

1. Créer les matrices de modèle, vue et projection pour placer l'objet dans la scène virtuelle.
2. Manipuler la matrice de vue pour se déplacer autour de l'objet.

Coté *GLSL*

Nous allons commencer par écrire le vertex shader nous permettant de procéder à la transformation *MVP*.

Ouvrez le vertex shader nommé "vertex.glsl" dans le dossier "shader/". Vous remarquerez que celui-ci crée une couleur pour le sommet, qui est ensuite transmise au fragment shader.

Pour récupérer les matrices dans le shader, ajoutez-y le code suivant au début :

```
// Parametres (uniform)
uniform mat4 ModelMatrix;       // matrice du modele
uniform mat4 ViewMatrix;        // matrice de vue
uniform mat4 ProjectionMatrix;  // matrice de projection
```

Pour appliquer la transformation, il faut faire :

```
// Transformation MVC
gl_Position = ProjectionMatrix * ViewMatrix * ModelMatrix * vec4(in_position, 1.0);
```

Il n'est pas nécessaire de modifier le fragment shader à ce stade là.

Note : Contrairement au TP précédent, ce shader est en *GLSL*150. Le mot clef `attribute` (pour les variables) a été remplacé par les mots clefs `in` (pour les données d'entrée) et `out` (pour les données de sortie). Le mot clef `uniform` (pour les paramètres) reste le même. Les données d'entrée sont différentes pour chaque instance du shader (ie. pour chaque sommet : position, couleur, ...), tandis que les paramètres sont uniques. Les données de sortie sont transmises au shader suivant (depuis le vertex shader vers le fragment shader).

Coté *C++*

Nous devons maintenant créer les matrices dans "main.cpp" et les transmettre à notre shader.

- **Matrice modèle :** On ne va pas déplacer notre modèle dans ce TP. L'espace du modèle et l'espace du monde sont confondus, la matrice modèle est donc la matrice identité.
- **Matrice de vue :** On va placer notre camera en $(0, 0, 2)$ (position) et la pointer sur $(0, 0, 0)$ (orientation).
- **Matrice de projection :** On va utiliser une projection en perspective avec un angle de 45°.

Création des matrices. Ce code permet de créer les matrices, il est à mettre dans l'initialisation du programme :

```
// Creation des matrices
mat4 model_matrix      = mat4(1.0);
mat4 view_matrix       = lookAt(vec3(0.0, 0.0, 2.0),     // Position
                                vec3(0.0),               // Orientation
                                vec3(0.0, 1.0, 0.0));    // Direction verticale
mat4 projection_matrix = perspective(45.0f,              // Angle de vue
                                     WIDTH / HEIGHT,     // Ratio de la fenetre
                                     0.1f,               // Limite de proximite
                                     100.0f);            // Limite d'eloignement

// Recuperation des ID des matrices dans le shader program
GLuint MmatrixID = glGetUniformLocation(programID, "ModelMatrix");
GLuint VmatrixID = glGetUniformLocation(programID, "ViewMatrix");
GLuint PmatrixID = glGetUniformLocation(programID, "ProjectionMatrix");
```

Transmission des matrices. Ce code permet de transmettre les matrices au shader, il est à mettre dans la boucle de rendu, après la mise en place du shader :

```
glUniformMatrix4fv(MmatrixID, 1, GL_FALSE, value_ptr(model_matrix));
glUniformMatrix4fv(VmatrixID, 1, GL_FALSE, value_ptr(view_matrix));
glUniformMatrix4fv(PmatrixID, 1, GL_FALSE, value_ptr(projection_matrix));
```

Vous devriez maintenant voir le carré initial légèrement rétreci sous l'effet de la perspective.
Pourquoi est-il nécessaire de spécifier une direction verticale ?

Changement de point de vue. Pour déplacer la camera dans la scène, il suffit de modifier `view_matrix` dans la boucle de rendu. Pour ce faire, *glm* propose des fonctions de manipulation matricielles. Étant donné un espace caméra défini par une matrice `m`, on a :

- `rotate(mat4 m, float angle, vec3 axis)` : tourne `m` de `angle` (en degrés) autour de `axis`.
- `scale(mat4 m, vec3 value)` : change l'échelle de `m` selon x, y, z en fonction des composantes de `value`.
- `translate(mat4 m, vec3 displacement)` : déplace `m` de `displacement`.

Ces fonctions retournent le résultat de la composition de la tranformation définie par `m` avec une transformation spécifique. Autrement dit, elles implémentent une multiplication matricielle à gauche.
Utilisez ces fonctions pour modifier l'initialisation de la matrice de vue.

Animation de la caméra. En effectuant un changement de point de vue infime à chaque passage dans la boucle de rendu, on peut créer une animation.

Animez votre caméra en la faisant tourner autour de l'axe y.

Navigation dans la scene. Nous allons maintenant coder une fonction nous permettant de controler le point de vue sur a scene grâce au clavier. *GLFW* nous permet de connaitre l'état des touches du clavier (appuyées ou non). Ainsi, à chaque passage dans la boucle de rendu, nous pouvons interroger l'état des touches et modifier la matrice de vue en conséquence.

```
if (glfwGetKey( GLFW_KEY_UP ) == GLFW_PRESS)
{
  // ici : modifier view_matrix

}
```

Vous disposer d'un identifiant pour chaque touche du clavier. En voici quelques uns :

- `GLFW_KEY_UP` : touche "haut"
- `GLFW_KEY_DOWN` : touche "bas"
- `GLFW_KEY_RIGHT` : touche "droite"
- `GLFW_KEY_LEFT` : touche "gauche"
- `GLFW_KEY_Z` : touche "Z"
- `GLFW_KEY_LSHIFT` : touche "shift" de gauche

Dans la fonction `view_control()`, implémentez de quoi tourner la caméra autour de l'origine.

Note : Lorqu'on compose des rotations, on effectue successivement des multiplications à gauche de matrices. Il faut donc prendre en compte la rotation des axes induites par les rotations précédentes. La fonction `inverse(mat4 m)` permet d'obtenir l'inverse de `m`. Cette fonction intervient dans la définition de l'axe de rotation. Typiquement, pour tourner autour de l'axe `x`, il ne suffira pas d'utiliser `vec3(1.0,0.0,0.0)` pour la rotation, mais plutôt `vec3(inverse(view_matrix)*vec4(1.0,0.0,0.0,0.0))`.

Bonus

Si vous avez fini la partie précédente, vous pouvez continuer par celle-ci. Elle vous propose des extensions des concepts que nous venons de voir.

Contrôle de la vitesse de déplacement.

Jusqu'à maintenant, si vous restez appuyé sur une touche, celle-ci sera détectée à chaque passage dans la boucle de rendu. Un changement de point de vue sera donc effectué à chaque itération. Or la fréquence à laquelle la boucle va s'executer dépend de critères arbitraires : puissance de l'ordinateur, charge *CPU*, *et caetera*. Pour contrer ce phénomène, vous pouvez calculer le temps dt écoulé entre chaque itération. Ainsi, vous pourrez moduler l'amplitude du déplacement du point de vue par dt :

$$dx = v \times dt$$

Dans cette équation, v est une constante représentant la vitesse à laquelle sera effectuée la déformation d'amplitude dx. Pour mesuer dt, vous aurez besoin de la fonction `glfwGetTime()`.

Calcul de *Frame Rate*.

La fréquence à laquelle la boucle de rendu est effectuée est appellée *Frame Rate* et s'exprime en *FPS* (pour *Frame Per Second*). C'est un bon indicateur de la puissance requise par le processus d'affichage en ragard de la puissance de l'ordinateur. Dans la plus part des applications dites "temps réel" (comme les jeux par exemple), on cherche à maintenir un *Frame Rate* supérieur à 30 *FPS*. D'après la section d'avant, on peut calculer le *Frame Rate* ainsi :

$$FR = \frac{1}{dt}$$

Vous pouvez calculer le *Frame Rate* à chaque passage dans la boucle de rendu, et l'afficher dans le nom de la fenêtre par exemple.

Déplacement des objets de la scene.

De la même manière que nous avons précédemment modifié la matrice de vue pour changer le point de vue, nous pouvons changer la matrice modèle pour déplacer un objet dans le monde. Construisez un nouvelle fonction permettant d'associer l'appuis sur les touches du clavier à la modification de la matrice modèle.

Mode de manipulation de la scene.

Pour pouvoir continuer à contrôler la caméra, il faut créer deux modes : l'un permettant de contrôler la caméra (et génerant l'appel à `view_control`), et l'autre permettant de contrôler l'objet (et génerant l'appel à votre nouvelle fonction). Le passage dans un mode peut être associé à l'appui sur une touche. Par exemple :

- `F1` : mode camera
- `F2` : mode objet

Changement des paramètres de prise de vue.

Comme précédemment, nous pouvons contrôler la prise de vue en modifiant la matrice de projection lors de l'appuis sur les touches. Ajoutez un mode de contrôle de la prise de vue.

Contrôle de la vue à la souris.

Afin d'offrir une interface plus fluide, il est recommandé de pouvoir manipuler la vue 3D grâce à la souris. Utilisez pour cela les fonctions `glfwSetMouseButtonCallback()`, `glfwSetMousePosCallback()`, `glfwSetMouseWheelCallback()`, et `glfwGetMouseButton()`.

Organisation

Rendre une archive *nom1_ nom2.zip* contenant :

- Le code commenté prêt à être compilé/executé et générant les sorties demandées.
- Un rapport contenant vos résultats (réponses, images, commentaires, ...).

À envoyer par mail à `romain.vergne@inria.fr` avec "[RICM4-TP2] Nom1 Nom2" en objet.