TP2 : Transformations et Indexation.

Introduction

L'objectif du TP est de construire un outil de visualisation de modèle 3D simple permettant :

- La navigation autour du modèle à l'aide d'une caméra de type Arcball.
- Le chargement d'un modèle 3D dans la scène OpenGL.

Pour ce TP, le travail du dernier TP est encapsuler dans la classe `GLwidget`. Le projet contient une nouvelle classe `CameraArcball` qui sera complétée pendant le TP.

Dans le dossier `source`, utilise CMake dans un dossier temporaire pour générer un Makefile.

Pipeline de Transformation

Comment déplacer, tourner, changer l'échelle d'un modèle 3D ? Comment créer une caméra pour modifier la position du point de vue ? 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**.

1. Du côté d'OpenGL, les transformations peuvent être appliquer **très efficacement**.
2. Du côté utilisation, un simple produit matriciel permet de combiner différents type de 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.

Exemple de translation

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

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 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

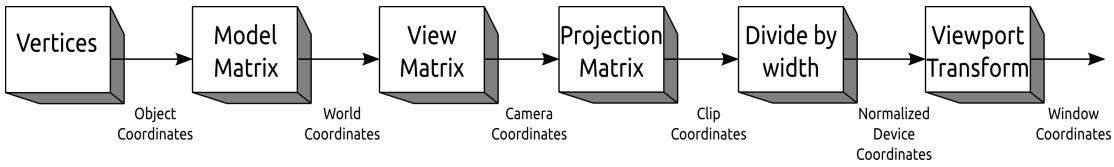

Figure 1: Pipeline de transformation d'OpenGL

Le pipeline de **transformation** intervient avant le pipelinde de **rasterization**.

Matrice Modèle

La matrice modèle place l'objet dans le repère du monde. Un modèle 3D est défini par un système de coordonnées locales dont l'origine $(0, 0, 0)$ correspond au centre de l'objet. Les coordonnées locales sont transformées en coordonnées de l'espace monde.

Exemple : Dans un jeu, le déplacement d'un personnage (rotation translation) est définie par la matrice modèle qui est composée à partir des intéractions du joueur.

Matrice de Vue

La matrice de vue permet de choisir le point de vue de la scène. C'est l'équivalent d'une caméra au détail près qu'en informatique graphique **la caméra ne bouge pas, c'est le monde qui bouge**. La matrice de vue définie une autre transformation sur le monde pour le mettre là où on veut le regarder.

Exemple : Initiallement la caméra est à l'origine de l'espace monde. Si l'on veut déplacer la caméra de 3 unités sur la droite, cela revient à déplacer le monde de 3 unités sur la gauche.

Matrice de Projection

Deux transformations ont déjà été appliquées. On a positionné notre modèle dans le monde et ensuite on a placé le monde où on voulait le voir. Maintenant il faut rendre compte de la perspective, de la distance par rapport à la caméra et c'est la qu'intervient la matrice de projection. La matrice de projection va transformer ce que voit la caméra dans un cube unité.

Note : Il existe deux types de projection, perspective et orthographique. Dans ce tp on ne s'intéresse qu'à la première.

Conclusion

En appliquant à un point x le produit matriciel $P * V * M * x$ on place un point dans le monde, sous le point de vue souhaité et avec la perspective voulue. Cette décomposition en trois matrices permet de changer indépendamment chacune des composantes essentielles à la construction d'une scène virtuelle 3D.

Travail demandé

La première partie du TP consiste en deux étapes :

1. Créer les matrices de modèle, vue et projection pour placer le triangle dans la scène virtuelle.
2. Manipuler la matrice de vue pour se déplacer autour du triangle.

Construire les matrices de modèle, vue et projection

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.

```
/*
File : glwidget.cpp
Function : init(const char* filename)
Location : between glUseProgram(programID) et glUseProgram(0)
*/

glm::mat4 model(1.0f) //identity

//Get an id for our "uniform" matrix
GLuint modelMatrixID = glGetUniformLocation( programID, "model" );

//Send our matrix to the currently used shader program
glUniformMatrix4fv(      modelMatrixID, 1,
                         GL_FALSE, glm::value_ptr(model) );
```

***glGetUniformLocation* :** On peut envoyer différents types de données à openGL, des attributs et des uniformes. Les attributs sont des données qui varient par sommet (position, couleur, normale). Les uniformes sont des données invariantes par sommet (matrix modèle, vue, projection). Les positions de nos sommets sont envoyées aux **vertex shader** comme des attributs et *glGetAttribLocation* est utilisée pour le déclarer au shader. La matrice modèle est envoyé comme un uniforme et la fonction associée est *glGetUniformLocation*.

***glUniformMatrix4fv* :** Cette fonction permet d'envoyer directement au shader une matrice 4×4 et d'associer à un identifiant les données envoyées.

1. Le paramètre 1 spécifie juste que l'on envoie une seule matrice.
2. Le paramètre *GL_FALSE* spécifie si il faut transposer la matrice ou pas.

Maintenant on va écrire le code **GLSL** pour récupérer notre matrice modèle et l'utiliser.

```
#File : vertex.glsl
#Note : * = multiplication en GLSL

uniform mat4 model;

#TODO : Dans le main placer le sommet dans l'espace monde
```

Compiler et exécuter votre code. Normalement vous devez toujours voir votre triangle sous le même point de vue.

Vue et Projection : Les matrices de vue et de projection sont encapsulées dans la classe *CameraArcball*.

```
/*
File : glwidget.cpp
Function : init( const char* filename )
Location : After model matrix initialization
Note : Use camera attributes ( camera.view, camera.viewMatrixID, ... )
*/

//Initialize view matrix and projection matrix
//Do not pay attention to the two last parameters for now
glm::vec3 center(0.0,0.0,0.0);
float radius = 1.0f;
camera.initialize(width, height, center, radius);

//TODO

//1- Get an id for camera.viewMatrixID
//Load the matrix into the shader
//2- Get an id for camera.projectionMatrixID
//Load the matrix into the shader
```

On va initialiser les deux matrices maintenant. La librairie *glm* simplifie énormément l'initialisation de la scène.

```
/*
File : camera.cpp
Function : initialize(int _width, int _height)
*/

//TODO

//1- Initialize view matrix : move the world in camera space
view = glm::mat4(1.0); //start from identity
view = glm::translate(view, translation); //translation
//X-rotation
view = glm::rotate(view, rotation[0], glm::vec3(1.0,0.0,0.0);
//Y-rotation
view = glm::rotate(view, rotation[1], glm::vec3(0.0,1.0,0.0);

//2- Initialize projection matrix using
//      glm::perspective( fov, aspect, nearPlane, farPlane )
//      aspect = width / height (warning : they are integer)
```

Dans le vertex shader , récupérer la matrice de vue et la matrice de projection. Utiliser les pour placer les sommets comme expliqué précédemment.

Enfin il faut recharger la matrice de vue à chaque fois que l'on dessine. Ainsi lorsque l'on utilisera une caméra les données sur la carte graphique seront à jour.

```
/*
File : glwidget.cpp
Function : renderShader()
Location : beginning
*/
//We re-send our view matrix to the graphic card
glUniformMatrix4fv(      camera.viewMatrixID, 1,
                         GL_FALSE, glm::value_ptr(camera.view) )
```

Compiler et exécuter. Le triangle devrait avoir changé un peu sous l'effet de la perspective.

Manipuler une caméra *Arcball*

Dans cette exercice on construit une caméra Arcball qui permet de tourner autour du modèle. Pour placer un modèle dans le repère de la caméra :

1. Translater le monde de la distance entre la caméra et le modèle.
2. Faire tourner le modèle autour de l'origine (là où est la caméra).

Pour faciliter son implémentation, on stocke un vecteur `rotation` et un vecteur `translation` qui permet de reconstruire complètement la matrice de vue. Ses vecteurs sont modifiés en fonction des intéractions avec le clavier.

```
/*
File : camera.cpp
Function : update()
*/
//TODO :Reconstruire la matrice de vue en partant de l'identite.
```

Enfin il faut effectuer la mise à jour de la caméra à chaque interaction avec le clavier.

```
/*
File : glwidget.cpp
Function : displayLoop()
Location : after eventHandler()
*/
camera.update();
```

En utilisant les touches du clavier décrites dans la fonction `eventHandler` dans la classe `GLwidget` on peut désormais naviguer autour du modèle.

Maillage

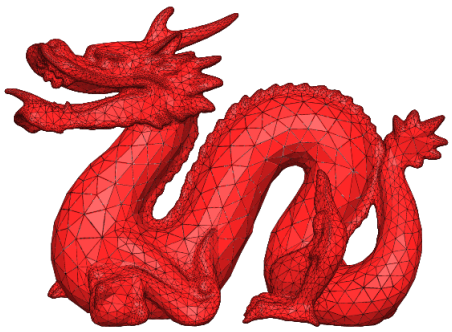

Figure 2: Exemple de maillage

C'est quoi ?

Wikipedia nous dit que

1. "Un maillage est la **discrétisation** d'un milieu continu par des éléments bien définis."
2. "Le but d'un maillage est de représenter un modèle dans l'optique simulations ou **de représentation graphique**"

Un maillage nous permet de représenter des objets en utilisant des primitives simples (ligne, triangle, ...). Ce qui tombe bien c'est que openGL peut traiter ces données par millions via la carte graphique.

Les formats

Les maillages sont stockées sous forme de fichiers. Ces fichiers stockent les primitives mais peuvent également contenir d'autres informations comme les normales, les couleurs, les matériaux, les coordonnées de texture, ... Il existe différents formats de stockage : OBJ, PLY et OFF. Dans ce TP on utilisera le format OFF. Une classe `Mesh` effectue le chargement depuis un fichier et on peut ensuite accéder aux sommets, aux normales et aux indices des faces.

```
OFF

#vertex_count face_count  edge_count
3     1     0

#One line for each vertex
0.0   0.5   0.0
0.5   -0.5  0.0
-0.5  -0.5  0.0

#One line for each polygonal face
#vertex_number vertex_indices
3 0 1 2
```

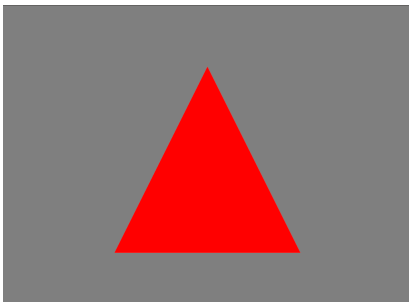

Figure 3: Exemple de fichier OFF pour un triangle

Dans la prochaine partie on va voir comment charger un maillage et la nouvelle notion que cela implique, **l'indexation**.

Indexation

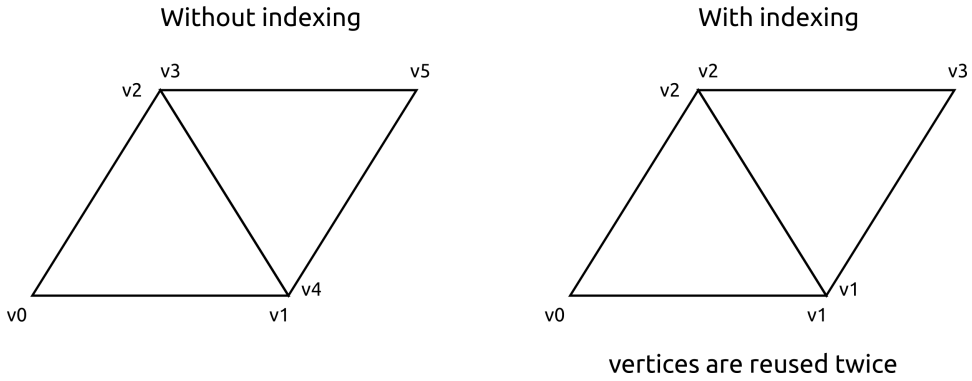

Figure 4: Exemple d'indexation de sommets

Lorsque l'on décrit un maillage, les faces sont exprimées en fonction des indices des sommets. Le même sommet donc le même indice est souvent utilisé plusieurs fois, c'est le principe de **l'indexation**. Cela permet une économie non négligeable sur le nombre de sommet que l'on va transmettre à la carte graphique.

Travail demandé

Désormais l'exécution devra se faire en prenant en argumant un chemin vers un maillage OFF.

```
./polytech_ricm4 ./../model/dragon.off
```

Chargement de maillage

Désormais on transmettra à la carte graphique, la liste des sommets mais également la liste des indices composants les faces. OpenGL se chargera de faire le lien entre les deux. On va déclarer notre maillage dans *glwidget.h*.

```
//Supprimer vector<vec3> vertices et les donnees dures du cpp
Mesh maillage;
```

On charge notre maillage.

```
/*
File : glwidget.cpp
Function : init( const char* filename )
Location : beginning
*/
maillage = Mesh(filename);
```

`Mesh` contient la liste des sommets, des normales, des faces du maillage mais aussi le centre du maillage et un rayon englobant le modèle. Le centre et le rayon vont nous servir à initialiser la caméra de manière à voir le maillage. A vous de remplacer dans le code :

1. L'initialisation de la caméra : maillage.center, maillage.radius
2. L'initialisation des shaders : maillage.vertices

Indexation

A ce moment si vous compilez et exécutez l'affichage sera incohérent car nous n'avons pas encore utilisé l'indexation.

```
/*
File : glwidget.cpp
Function : init(const char* filename)
Location : After vertex position buffer creation
*/

glGenBuffers(1, &indexBufferObject);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferObject);
glBufferData( GL_ELEMENT_ARRAY_BUFFER,
        maillage.faces.size()*sizeof(unsigned int),
        maillage.faces.data(), GL_STATIC_DRAW );
```

La création du buffer des indices est quasi identique celle du vertex buffer object. La seule différence est la spécification de `GL_ELEMENT_ARRAY_BUFFER`. Maintenant à la place d'utiliser `glDrawArrays` qui ne prenait pas en compte les indices on va utiliser `glDrawElements`.

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferObject);
glDrawElements( GL_TRIANGLES, maillage.nb_faces*3,
        GL_UNSIGNED_INT, (void*)0);
```

Avant de dessiner on *bind* le buffer d'indice pour préciser à openGL quel buffer d'indice utiliser. **Note :** N'oubliez pas le `glDeleteBuffers` après la boucle de dessin. Compilez et exécuter. Désormais vous pouvez charger un modèle OFF et naviguer autour du modèle.

Chargement des normales.

Vu que tout les triangles ont la même couleur on ne perçoit aucun des détails de notre modèle. C'est là qu'intervient la lumière qui sera l'objet de notre prochain TP. Mais en attendant on peut avoir quelque chose de mieux.

```
/*
FILE : glwidget.cpp
FUNCTION : init(const char* filename)
LOCATION : after vertex position buffer object
*/

vertexNormalID = glGetAttribLocation(programID, "vertex_normal");
glGenBuffers(1, &normalBufferObject);
glBindBuffer(GL_ARRAY_BUFFER, normalBufferObject);
glBufferData( GL_ARRAY_BUFFER,
```

```
        maillage.nb_vertices*sizeof(glm::vec3),
        maillage.normals.data(), GL_STATIC_DRAW) ;
/*
FILE : glwidget.cpp
FUNCTION : renderShader()
LOCATION : after vertexPositionID enabling
*/

glEnableVertexAttribArray(vertexNormalID);
    glBindBuffer(GL_ARRAY_BUFFER, normalBufferObject);
    glVertexAttribPointer(
        vertexNormalID,
        3,
        GL_FLOAT,
        GL_TRUE,
        0,
        (void*)0
        );
```

Note : N'oubliez pas le `glDeleteBuffers` et le `glDisableVertexAttribArray` !

```
/*
FILE : vertex.glsl
*/

attribute vec3 vertex_normal;
varying vec3 normal_cameraSpace;

//Dans le main

normal_cameraSpace = (view*model*vec4(vertex_normal,0)).xyz;
```

On place les normales dans l'espace de la caméra et on les transmet au fragment shader en utilisant une variable **varying**.

```
/*
FILE : fragment.glsl
*/

varying vec3 normal_cameraSpace;

//Dans le main

gl_FragColor = vec4(normal_cameraSpace, 1.0f);
```

On récupère la variable dans le fragment shader et on utilise ses coordonnées comme des couleurs. Et là le résultat devrait être assez surprenant. On a du mal à distinguer la profondeur ce qui donne une impression de transparence.

```
/*
FILE : glwidget.cpp
FUNCTION : init(const char* filename)
LOCATION : beginning
*/

//Active le buffer de profondeur d'OpenGL
glEnable(GL_DEPTH_TEST);
```

De plus la plupart des modèles sont fermés. Toutes les faces et sommets dessinées à l'intérieur du modèle sont donc inutiles. Empêcher leur dessin s'appelle le **Back-Face Culling**. Là encore OpenGL le fait pour nous ... si nous le demandons.

```
/*
FILE : glwidget.cpp
FUNCTION : init(const char* filename)
LOCATION : beginning
*/

glEnable (GL_CULL_FACE); // cull face
glCullFace (GL_BACK); // cull back face
glFrontFace (GL_CCW); // GL_CCW for counter clock-wise
```

Questions

1. Quelles sont les transformations à appliquer à la matrice de vue pour simuler une caméra à première personne similaire à un jeu du même nom ?
2. Comment calculer la normale d'un triangle, donc d'une face ?
3. Comment en déduire les normales aux sommets d'un maillage triangulaire ?

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, ...).