TD Transformations

Malek.Bengougam@gmail.com

L'objectif de ce TD est de mettre en pratique les transformations matricielles, et de passer progressivement d'un rendu 2D à un rendu 3D.

Partie 1 : transformation avec matrices 2D

1.1 mat2

En 2D, il est la plupart du temps suffisant d'utiliser des matrices 2x2. Pour rappel, une matrice stocke les facteurs d'un système d'équation linéaire, où les paramètres sont les coordonnées d'un repère :

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$$

On peut donc voir [a c] comme les facteurs de l'axe X, et [b d] les facteurs de l'axe Y.
Lorsqu'il s'agit de multiplier (concaténer) des matrices on applique la formule suivante :

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix} \times \begin{vmatrix} x & y \\ z & t \end{vmatrix} = \begin{vmatrix} ax + bz & ay + bt \\ cx + dz & cy + dt \end{vmatrix}$$

En GLSL le type **mat2** correspond à une matrice colonne 2x2.

Côté C++, on peut affecter une valeur à une matrice -qui est une variable uniforme- à l'aide de :

```
glUniformMatrix2fv(GLint location, GLint num, bool transpose, void* pointeur)
```

Notez que la fonction se termine par le suffixe 'v' ce qui
Les paramètres sont les suivants :

- location : l'identifiant d'une variable uniform dans le shader, récupéré avec **glGetUniformLocation**(program, « nom »)
- num : utile dans le cas des tableaux de matrices, généralement à 1
- transpose : permet de transposer la matrice avant de l'envoyer au shader, GL_FALSE sinon
- pointeur : l'adresse de notre matrice (ou un tableau de matrices si num > 1)

Bien évidemment, pour les matrices 3x3 et 4x4 il existe une variante de cette fonction, respectivement **glUniformMatrix3fv**() et **glUniformMatrix4fv**().

1.3 Rotation 2D

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$

1.4 Translation

Il n'est pas possible de représenter une translation (qui est par essence une addition) avec une simple matrice 2x2 qui effectue essentiellement des multiplications. Cependant, comme le résultat est un produit scalaire des lignes à gauche et des colonnes à droite, et que le produit scalaire inclus des additions, on peut l'exploiter -via une sorte de hack mathématique- à notre avantage.

2. Matrices homogènes 3D

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + c \\ dx + ey + f \\ 1 \end{bmatrix}$$

Afin de combiner ces matrices de nature différente, il devient nécessaire d'ajouter une dimension supplémentaire à nos matrices. Ainsi, la dernière colonne représentera toujours la dernière addition. En s'assurant que la dernière composante des colonnes à droite vaut 1, l'addition est alors possible.

On parle alors de coordonnées homogènes et de matrices homogènes -une matrice pouvant combiner des transformations linéaires et des transformations affines (comme la translation, projection…), et ceci afin de les distinguer des transformations linéaires d'une matrice 3x3.

Par exemple, la matrice de « scale » suivante est une transformation linéaire purement 3D. Elle se distingue notamment par le fait que le vecteur à droite est une colonne dont la dernière composante n'est pas égale à 1, et que la dernière colonne à gauche contient un facteur d'échelle pour l'axe z :

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} S_x.x \\ S_y.y \\ S_z.z \end{bmatrix}$$

Attention ! Si la matrice est 3D en coordonnées homogènes, les transformations restent elles en 2D !

Une matrice de translation homogène ressemble alors à ceci :

$$\begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ 1 \end{bmatrix}$$

Une matrice de rotation 2D (autour de l'axe Z ou forward) ressemble à cela :

$$\begin{bmatrix} cos\emptyset & -sin\emptyset & 0 \\ sin\emptyset & cos\emptyset & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x.cos\emptyset - y.sin\emptyset \\ x.sin\emptyset + y.cos\emptyset \\ 1 \end{bmatrix}$$

C'est donc essentiellement la même matrice que la rotation 2D, avec une 3eme colonne à l'identité.

En GLSL, on utilisera le type mat3 pour représenter toute matrice 3x3.

Note : Les matrices peuvent également stocker des valeurs liées à d'autres systèmes de coordonnées, comme un espace colorimétrique (RGB, HSB, YUV …). Ci-dessous une matrice permettant de convertir d'un espace YUV vers RGB :

```
mat3 yuv2rgb = mat3(1.0, 0.0, 1.13983,
                    1.0, -0.39465, -0.58060,
                    1.0, 2.03211, 0.0);
```

Partie 2 : transformation 3D homogènes

2.1 mat4

Pour les mêmes raisons que l'on a rencontré dans le cas 2D, il est nécessaire d'utiliser des coordonnées homogènes 4D (3D + 1 dimension) afin de pouvoir combiner des transformations linéaires et des transformations affines.

Ci-dessous, la forme standard d'une matrice homogène et d'un vecteur en coordonnées homogènes.

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax + by + cz + d \\ ex + fy + gz + h \\ ix + jy + kz + l \\ 1 \end{bmatrix}$$

Exercice 1 : implémentez et passez au Vertex Shader les matrices de scale, rotation et translation en coordonnées homogènes 4D.

En GLSL on utilise le type mat4, ainsi que la fonction glUniformMatrix4fv() côté C++.

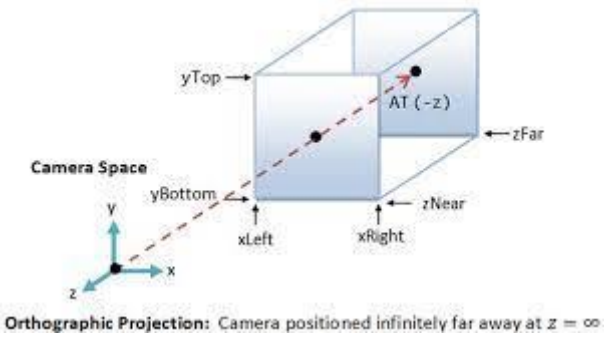
L'ordre traditionnel des transformations dites « rigides » (similaire à ce que ferait un composant Transform sous Unity 3D, l'Unreal Engine ou tout moteur de rendu 3D) est le suivant : d'abord scale, puis rotation, puis translation.

Remarquez que les matrices de translation ont toujours l'identité dans la diagonale, et les valeurs de déplacements dans la dernière colonne. On pourrait ainsi, dans le cas des matrices « rigid-body » insérer directement les translations dans la dernière colonne (sauf dans le cas des matrices inverses).

Attention, l'ordre des multiplications est important ! On décide de suivre la convention mathématique, l'ordre étant alors de la droite vers la gauche.

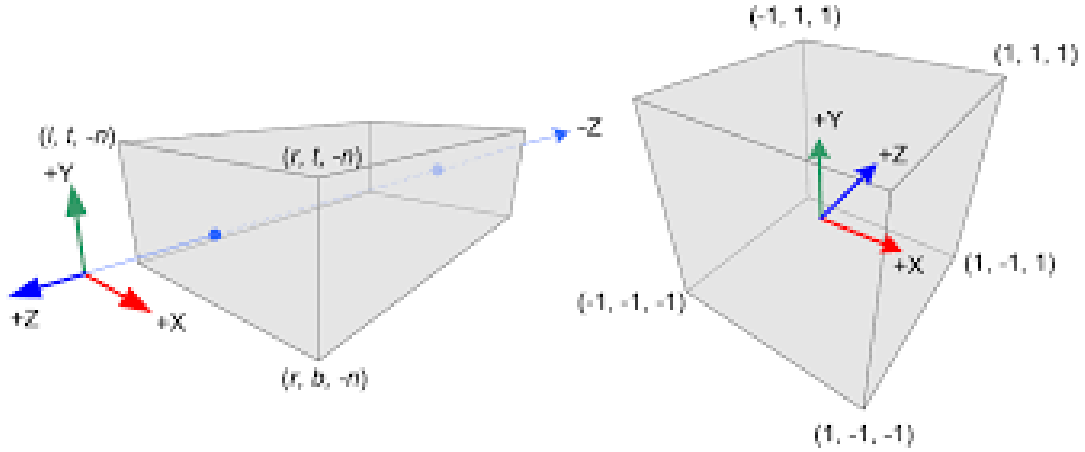
Exercice 2 (optionnel) : implémentez la multiplication matricielle 4x4.
Combinez scale, rotation et translation dans une seule matrice

2.2 Projection orthographique



Orthographic Projection: Camera positioned infinitely far away at $z = \infty$

Nous allons mettre en place une projection nous permettant de simuler un monde en pixels.
On va devoir faire correspondre les coordonnées normalisées NDC avec notre monde virtuel.
Il s'agit essentiellement d'un changement d'échelle pixel -> fragments [0..1].



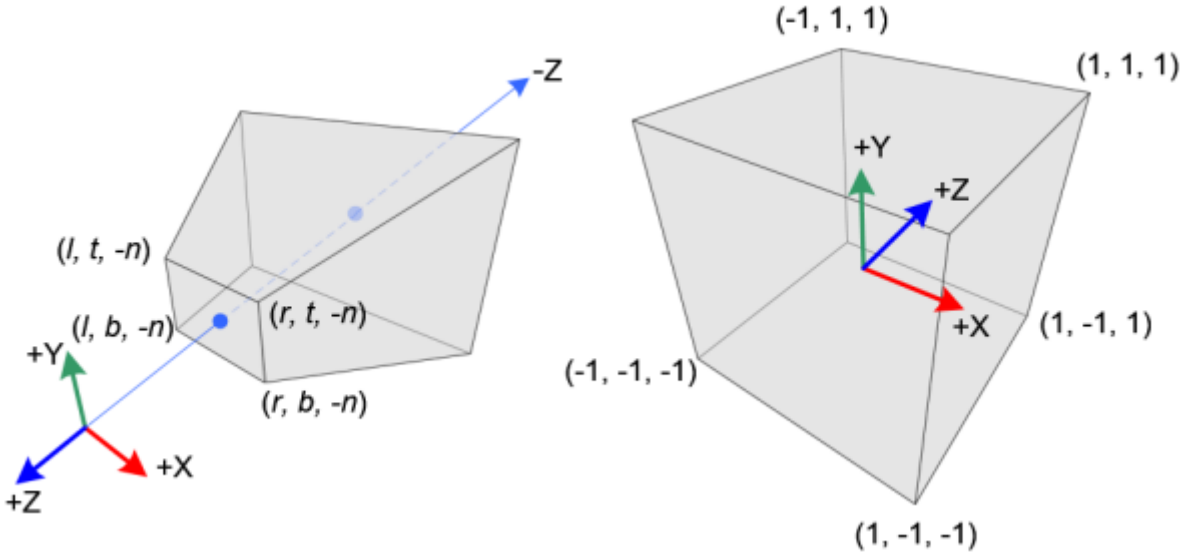
On souhaite aussi pouvoir déplacer l'origine du repère dans un coin, ou bien au centre. Cela correspond à une translation. Notez également l'inversion de l'axe +Z qui doit être en main-gauche en NDC.

On va ici devoir créer une matrice de projection 4x4 ayant la forme suivante en OpenGL :

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Right, left, top et bottom sont des valeurs exprimées dans les unités que vous désirez (en pixels par exemple). Far et near représente la distance des plans délimitant le champ de visibilité de la caméra (en unité arbitraire également). Si vous n'avez pas d'utilité pour cette profondeur, les valeurs par défaut sont respectivement +1.0 et -1.0f.

2.3 Projection perspective



La projection avec point de fuite prend la forme d'une pyramide tronquée par les deux plans near et far. Contrairement à la projection orthographique, la position de ces plans à une importance dans la visibilité et spécifie le début de la visibilité (near = 0.1f par exemple) et la fin de la visibilité (far=100.f par exemple). Ces valeurs sont en unités arbitraires, il vous revient donc de définir ces unités.

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{far+near}{near-far} & \frac{2\ near\ far}{near-far} \\ 0 & 0 & -1 & 0 \end{bmatrix} \text{ with } f = cotangent(\frac{fovy}{2})$$

Exercice 3 : Implémentez la matrice de projection perspective ci-dessus. Cette matrice doit être la dernière transformation avant d'écrire dans gl_Position.

Pour rappel, cotan= 1/tan, et aspect = width/height

A cause du plan near défini à 0.1 il est nécessaire de « reculer » la caméra, ou plutôt d'éloigner votre objet (en OpenGL rappelez-vous que le demi-espace forward est négatif) ce qui revient au même. En effet, si celui-ci est positionné à l'origine (0.f, 0.f, 0.f) alors son Z est derrière le plan Near, et de ce fait il n'est pas dans le champ de visibilité.