Programmation graphique 3D
M1 Informatique I3D - S1 Automne 2020–2021
Travaux Pratiques

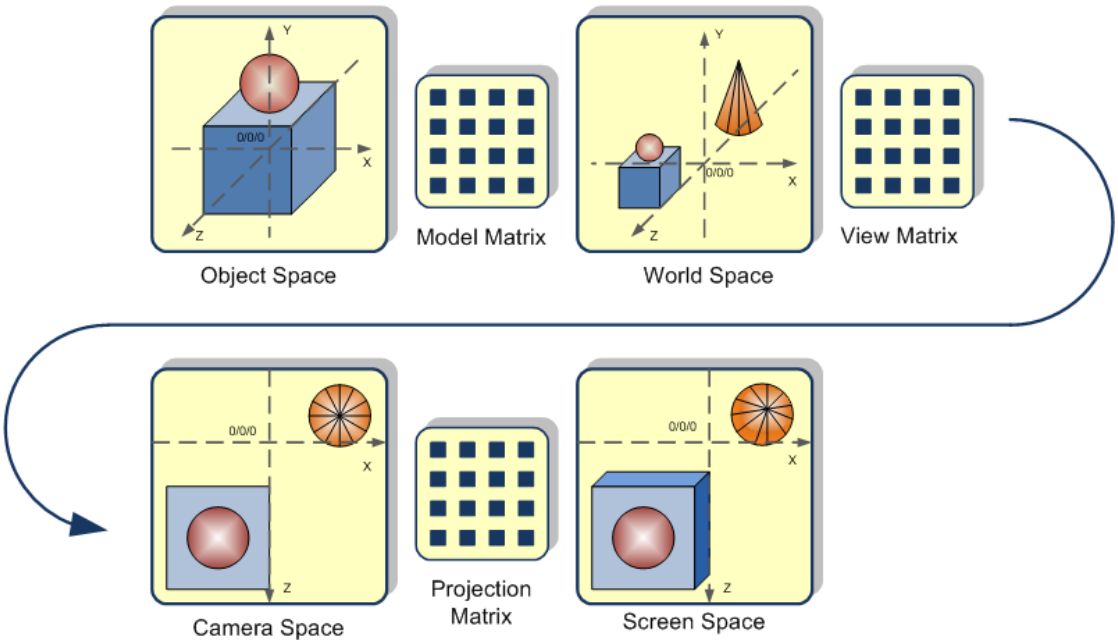
TP 2 : Transformations et rendu indexé

Exercice 1 : Matrices de transformations

NOTE : Partez du fichier corrigé *tp1_correction_03_drawQuad_useUniform_withGUI_withAnimation.js* du TP précédent.

Dans la première partie de ce TP, nous allons voir comment appliquer des transformations dans le pipeline graphique. Concrètement, il s'agit de définir 3 matrices :

- une matrice *Model* qui définit les transformations appliquées au model 3D en cours;
- une matrice *View* qui définit les paramètres de la caméra utilisée (position de l'oeil, point visé, orientation verticale);
- une matrice *Projection* qui définit le type de perspective utilisée (orthogonal ou perspective) avec ses paramètres (ouverture, ratio width/height 4/5 ou 16/9, positions des plans de caméra *near* et *far*)



1. On commencera par définir ces matrices manuellement.
   - Pour la matrice de projection, utilisez la fonction `perspective()` de `Matrix`
   - Pour la matrice vue, utilisez la fonction `look_at()` de `Matrix`
   - Pour la matrice model, utilisez une fonction de rotation de `Matrix` pour faire une rotation sur l'axe des Z avec un angle allant de 0 à 45 degrés en fonction du temps système (pensez à la fonction sin()).

   Comme ces données sont constantes pendant le rendu, il faudra utiliser des constantes sur GPU, c'est-à-dire des variables de type `uniform` et passer les valeurs depuis le CPU vers le GPU comme dans le TP précédent. Le langage de shader GLSL dispose du type `mat4`

qui définit une matrice 4x4.

RAPPEL : transformer des points dans l'espace d'un repère vers un autre consiste à multiplier un point par une matrice. Cette matrice contient la transformation à appliquer (ex: rotation, scale, translation, projection, etc...). Lorsque l'on applique plusieurs transformations, l'ordre des opérations se fait **de la droite vers la gauche**.

2. On va maintenant utiliser les matrices View et Projection de la caméra pré-existante dans la scène (`ewgl.scene_camera`). Pour cela, on commencera par remplacer la méthode ewgl.launch_2d() du précédent TP par sa version 3D ewgl.launch_3d() qui définit une caméra et les interactions avec la souris pour manipuler la vue 3D. On peut ensuite récupérer ces deux matrices avec les fonction `get_projection_matrix()` et `get_view_matrix()`.

Exercice 2 : Rendu indexé

Nous allons maintenant nous intéresser à un autre type de rendu : le rendu indexé. Auparavant, nous avons utilisé la méthode `glDrawArrays()` qui définissait la géométrie d'un maillage par ses points et le type de primitive (POINTS, LINES, TRIANGLES...) indiquait comment ils étaient regroupés au rendu (1 par 1, 2 par 2, 3 par 3, etc...). Ici on utilisera `glDrawElements()` qui se sert d'un buffer supplémentaire (**EBO: element array buffer**) composé des indices des points à utiliser pour définir la topologie du maillage en cours. Ce buffer d'indices de vertices sera ajouté au VAO (vertex array) avec un VBO de positions définissant uniquement la liste des points du maillage.

Nous l'appliquerons à la synthèse d'un terrain. On veut pouvoir afficher une grille régulière 2D de triangles. On commencera par définir tous les sommets de cette grille dans un VBO. On pourra ensuite indiquer dans un EBO, tous les indices des sommets à utiliser 3 par 3 pour former tous les triangles. Si on rajoute ensuite une fonction de bruit sur la position en z des sommets dans le vertex shader, on peut obtenir un terrain :