WebGL R5.A.06

TP N°2 : Définition d'une caméra pour visualiser une scène 3D

Objectif général. L'objectif de ce TP est de comprendre les différentes opérations matricielles mises en œuvre dans le rendu d'une scène 3D par une caméra de type *perspective*, tout en construisant pas à pas une version du monument mégalitihique *Stonehenge* où chaque élément de la structure circulaire est remplacé par une grande lettre "F" en 3D (d'où le nom du fichier `f-henge.js`), et au-dessus duquel la caméra vole dans un mouvement circulaire.

Exercice 1 : Positionner un premier F 3D et le visualiser avec une caméra sensible à la perspective

Objectif

- réviser l'écriture d'un fichier WebGL avec *shaders* vue lors du TP1.
- comprendre la projection orthographique réalisée entre les deux *shaders*.
- comprendre la notion de *frustum* et savoir calculer ses paramètres pour qu'un objet soit visible sur l'image après la projection, sans déformation.
- introduire dans le code les versions les plus simples des deux matrices utilisées pour simuler une caméra projective : la *projectionMatrix* et la *modelViewMatrix*.

(1) Inclure un premier F 3D

La géométrie d'un F 3D est une suite de primitives triangulaires, chaque primitive étant définie par ses trois sommets. Nous rappelons que l'ordre des trois sommets définit quelle face de la primitive est à l'intérieur ou à l'extérieur de l'objet. Les primitives triangulaires ont été définies par triangulation de tuiles rectangulaires. Cette géométrie est donnée sous la forme d'un `array` contenant la suite des trois coordonnées spatiales de chacun des trois sommets qui définissent une primitive triangulaire, primitive après primitive. Cet `array` est renvoyé par la fonction `getFGeometry()` définie dans le fichier `FGeometry.js`, lui-même inclus dans le fichier `index.html`.

En vous inspirant de ce que vous avez réalisé pour le TP1, compléter le fichier `f-henge.js` pour définir :

- un *vertex shader* qui se contente de renvoyer en coordonnées homogènes la position 3D spatiale du *vertex*;
- un *fragment shader* qui se contente de renvoyer la couleur `(0.8,0.2,0.2)` sans transparence pour le pixel lié au fragment;
- un programme construit à partir de la compilation de ces deux *shaders*;
- un *buffer* construit à partir de l'`array` définissant la géométrie d'un F 3D, accompagné de la description de son parcours utilisé pour alimenter la valeur de l'attribut du *vertex shader* qui contient la position 3D spatiale de chaque *vertex*. Attention, au préalable il faut : récupérer la *localisation* de cet attribut fournie par le programme (cette *localisation* est un argument nécessaire pour construire la description du parcours du buffer).

- les instructions de dessin de l'image, constituée de :
  - la spécification du programme de *shaders* utilisé (ici, il n'y en a qu'un : celui construit plus haut);
  - le rafraîchissement du `viewport` avec un fond de couleur `(0.5,0.7,1.0)`;
  - le dessin des 32 primitives triangulaires du F 3D (attention il faut indiquer le nombre de *vertices*).

Remarque : Le *fragment shader* dessine les pixels du `viewport` correspondant aux fragments du carré $[-1;1] \times [-1;1]$ où les objets 3D de la scène sont projetés. Ce `viewport` définit lui-même (normalement) un sous-ensemble (et non un sur-ensemble) du `canvas`. Comme pour le TP1, nous souhaitons dessiner sur la totalité du `canvas`. En revanche, contrairement au TP1, le CSS vient ensuite déformer le `canvas` pour qu'il prenne les dimensions de la fenêtre englobante (voir l'en-tête de `index.html`). Mais cela ne change rien : ce sont bien les dimensions *internes* (attributs HTML) du `canvas` et non celles de cette fenêtre englobante qu'il faut donner au `viewport` : `canvas.width x canvas.height`.

À ce stade, le F 3D n'est pas encore visible. Pour cela, il faut appliquer quelques opérations matricielles décrites ci-après.

(2) Opérations matricielles pour rendre visible ce premier F 3D : étude théorique

Entre le *vertex shader* et le *fragment shader*, seuls les éléments de la scène 3D inclus dans le cube $\mathcal{C} = [-1;1] \times [-1;1] \times [-1;1]$ sont considérés, projetés selon une *projection orthographique* sur le plan d'équation $z = -1$, puis échantillonnés en autant de fragments que de pixels définissant le `viewport`. Enfin, le `viewport` est transformé par la propriété CSS du `canvas`, qui le dilate aux dimensions de la fenêtre englobante.

Or une projection orthographique ne permet pas de rendre un effet de perspective. Pour cela, la scène 3D est plutôt définie dans un *frustum* $\mathcal{F}$. Un *frustum* est un volume de l'espace 3D ayant la géométrie d'une pyramide tronquée. Plus précisément, en considérant la caméra dans sa position par défaut :

- La pointe virtuelle de la pyramide tronquée correspond à la position de la caméra (l'origine du repère 3D).
- Les plans tronquant la pyramide sont orthogonaux à l'axe de visée de la caméra (l'axe des Z), à une distance n (*near*) de la pointe virtuelle pour le sommet de la pyramide, et f (*far*) pour sa base. L'axe de visée traverse en leur centre les deux rectangles qui constituent le sommet et la base de la pyramide tronquée.
- La pente des côtés haut et bas de la pyramide (qu'une droite parallèle à l'axe vertical des Y pourrait traverser) est définie par l'angle défini par ces deux côtés à la pointe virtuelle de la pyramide et nommé *Field of View* (*fov*). La distance n et l'angle *fov* définissent la hauteur h du rectangle qui constitue le sommet de la pyramide tronquée.
- La largeur w du rectangle qui constitue le sommet de la pyramide est quant à elle définie par le produit de la hauteur h par un *ratio d'aspect* a. La pente des côtés gauche et droite de la pyramide (qu'une droite parallèle à l'axe horizontal des X pourrait traverser) s'en déduit directement.

Le *frustum* $\mathcal{F}$ est donc défini par les paramètres suivants : n, f, *fov* et a.

Ensuite, il suffit d'appliquer la transformation linéaire transformant $\mathcal{F}$ en $\mathcal{C}$ qui, suivie de la projection orthographique finale réalisée implicitement entre les deux *shaders*, réalise alors la projection de la scène 3D incluse dans $\mathcal{F}$ sur le rectangle qui en constitue le sommet, créant ainsi un effet de perspective.

Pour que le F 3D soit visible, il faut donc définir un *frustum* $\mathcal{F}$ qui le contienne :

- La transformation de $\mathcal{F}$ en $\mathcal{C}$ déforme par homothétie le rectangle $w \times h$ en le carré $[-1;1] \times [-1;1]$. Aussi au moment du rendu final, ce carré $[-1;1] \times [-1;1]$ est lui-même déformé en l'image non carrée définie par la fenêtre englobant le `canvas`. Pour éviter toute déformation de l'image il convient donc que les deux déformations se compensent et donc de choisir $a =$ `canvas.clientWidth/canvas.clientHeight`.
- Comme le repère 3D est défini avec l'axe des Z pointant vers nous, la valeur n est la distance (donc positive) séparant la caméra du plan de projection $z = -n$. Pour définir une pyramide tronquée, il est nécessaire de choisir $n > 0$. Nous suggérons $n = 1$.

- La valeur f est la distance séparant la caméra du plan de fond $z = -f$. La *bounding box* du F 3D est $[-15; 85] \times [0; 150] \times [-15; 15]$. Il faut donc le translater de $-z_t$ pour que $[-z_t - 15; -z_t + 15]$ puisse être contenu dans $[-f; -n]$. **Choisir $-z_t$ et f en conséquence.**

- Enfin, pour que $\mathcal{F}$ inclue le F 3D il faut que l'angle d'ouverture verticale fov définisse une hauteur supérieure à 150 dans le plan $z = -z_t + 15$ qui contient la face-avant du F 3D. **Quelle relation trigonométrique lie la distance $z_t - 15$, la hauteur minimale 150 et l'angle fov recherché?**

(3) Opérations matricielles pour rendre visible ce premier F 3D : application pratique

Modifier `f-henge.js` pour que :

- Le `vertex shader` renvoie le vecteur de type `vec4` construit à partir de la position 3D spatiale du *vertex*, transformé par son produit matriciel avec `u_projectionMatrix * u_modelViewMatrix` où :
  - `u_projectionMatrix` est la matrice de type `mat4` transformant l'espace, avant projection orthographique : ici elle sera définie dans le code javascript à partir d'une variable `projectionMatrix` (voir ci-dessous).
  - `u_modelViewMatrix` est la matrice de type `mat4` transformant un objet défini par rapport au repère initial, en cet objet positionné dans la scène et du point de vue de la caméra : ici, elle sera définie dans le code javascript à partir d'une variable `modelViewMatrix` (voir ci-dessous).

  On introduit donc deux nouvelles variables définies depuis le code javascript : sont-ce des attributs ou des uniforms? Compléter le code en conséquence (localisation etc.)

- Lors du dessin de l'image, avant le rafraîchissement du viewport, définir une variable `projectionMatrix` initialisée avec la matrice représentant la transformation linéaire de $\mathcal{F}$ en $\mathcal{C}$. Pour cela, nous vous fournissons le fichier `matrices.js` qui propose en particulier la fonction `projection()` qui, avec les arguments calculés dans la question précédente, renvoie une telle matrice, du type `matrix` de `math.js`. Utiliser cette variable `projectionMatrix` pour définir la valeur de `u_projectionMatrix` à l'aide de :
  - la fonction `flatten()` de `math.js`;
  - la méthode `valueOf()`.

  Consulter par exemple `https://mathjs.org/examples/matrices.js.html`. **Attention!** Rappelez-vous, en GLSL les matrices sont décrites par colonnes et non par lignes.

- Le rafraîchissement du `viewport` doit à présent inclure la réactualisation des informations de profondeur pour que le calcul des masquages entre primitives lors de la projection orthographique soit correcte :

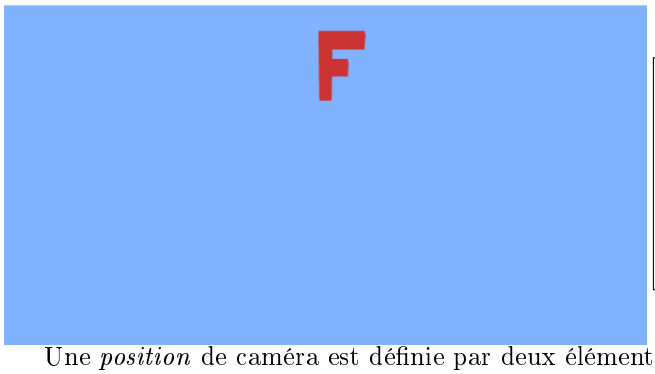
```
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
gl.enable(gl.DEPTH_TEST);
```

- Avant de dessiner les primitives triangulaires, définir une variable `objectMatrixWorld` qui positionne le F 3D dans la scène : elle est initialisée à l'aide de la fonction `translation()` fournie dans `matrices.js`. Définir ensuite une variable `modelViewMatrix` qui sera utilisée pour définir la valeur de `u_modelViewMatrix`. Ici, comme la caméra est à l'origine du repère initial, `modelViewMatrix=objectMatrixWorld`.

- Jouer sur les valeurs de f, $-z_t$, et fov pour bien comprendre leur rôle dans ce contexte, puis leur redonner leur valeurs théoriques pour sauvegarde de cette version. À cette occasion, si vous préférez définir l'angle fov en degré plutôt qu'en radians, vous pouvez utiliser la fonction `degToRad()` définie dans `angles.js` avant de l'utiliser dans `projection()`.

Copier le fichier `f-henge.js` en `f-henge-1.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 2 : Déplacer la caméra



Objectif

- manipuler les matrices de transformation (translation, rotation) en 3D.
- comprendre comment modifier la matrice *modelViewMatrix* pour traduire un changement de position de la caméra.

Une *position* de caméra est définie par deux éléments :

- Une *visée* : demi-droite partant d'un point de l'espace, définie par défaut comme le demi-axe des Z négatifs.
- Un vecteur UP : vecteur orthogonal à la demi-droite d'orientation qui indique où est le haut de l'image, défini par défaut comme $(0, 1, 0)$ (vecteur directeur du demi-axe vertical des Y positifs).

Si l'on change la position de la caméra, le rendu de la scène doit changer : il doit rendre compte du changement des positions relatives entre les objets de la scène et la caméra. Aussi, l'application linéaire (la matrice) qui change la position de la caméra décrit le changement de la position relative de celle-ci par rapport à sa position initiale : le repère du monde 3D. L'application linéaire qui décrit le changement de la position relative des objets, donc du monde 3D, par rapport à la caméra est alors l'application inverse.

Ainsi il suffit, pour chaque objet de la scène, de définir la matrice `modelViewMatrix` comme le produit de :

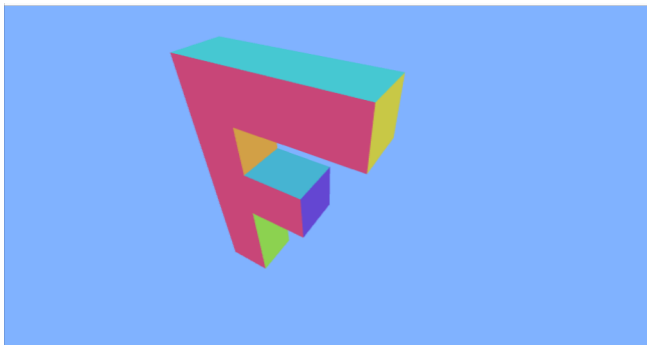
- l'inverse de la matrice `cameraMatrix` qui change la position de la caméra (par rapport à sa position initiale : le repère du monde 3D),
- et de la matrice `objectMatrixWorld` qui change éventuellement la position de l'objet (par rapport à sa position initiale : le repère du monde 3D également).

Mettre à jour le fichier `f-henge.js` pour intégrer le changement de position de la caméra :

- Avant de modifier la définition de `modelViewMatrix`, commencer par introduire une nouvelle variable `cameraMatrix` dont la valeur est une matrice de type `matrix` de `math.js` représentant le changement du vecteur UP de la caméra de 40° dans le sens inverse des aiguilles d'une montre. Cela correspond à une rotation autour de l'axe des Z définie par la fonction `zRotation()` du module `matrices.js`. Attention, cette fonction prend comme argument un angle donné en radians.
- Modifier la définition de `modelViewMatrix` comme décrit ci-dessus. Vous pourrez utiliser en particulier la fonction `inv()` de `maths.js` pour inverser une matrice.
- Jouer sur l'angle du vecteur UP et constater que tourner la caméra dans un sens revient à tourner la scène sur l'image dans l'autre sens.
- Nous allons à présent jouer sur la visée de la caméra. Mettre à jour la matrice `cameraMatrix` en la multipliant par des matrices de transformation (rotation, translation) fournies par le module `matrices.js` afin que la caméra vise le centre O du repère du monde 3D depuis un angle de 40° vers le bas à une distance de 500 de O, tout en restant dans le plan $x = 0$. Attention à l'ordre des multiplications de matrices!
- Pour avoir l'image finale désirée, annuler le changement du vecteur UP (en fixant l'angle de sa rotation à 0° par exemple).

Copier le fichier `f-henge.js` en `f-henge-2.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 3 : Utiliser des primitives triangulaires de différentes couleurs



Objectif

- réviser l'utilisation des *shaders* pour que la couleur d'un pixel dépende de la primitive dont il est un point projeté.
- mobiliser les nouvelles notions vues dans ce TP pour définir un point de vue souhaité sur la scène.

(1) Définir une couleur par primitive triangulaire du F 3D

Jusqu'à présent la couleur était définie par une valeur constante dans le *fragment shader*. Mettre à jour le fichier `f-henge.js` pour que :

- les 6 premiers triangles du F 3D (correspondant à la face avant) ait comme couleur `(ui8Colors[0], ui8Colors[1], ui8Colors[2], 1)` où `ui8Colors` est défini dans `couleurs.js`;
- les 6 triangles suivants (correspondant à la face arrière) ait une couleur définie par les trois entrées suivantes du tableau `ui8Colors` : `(ui8Colors[4], ui8Colors[5], ui8Colors[6], 1)`;
- les 10 faces suivantes du F 3D (ou tuiles) ait une couleur définie par un triplet d'entrées différent du tableau `ui8Colors`. Attention, chaque tuile est constituée de 2 triangles, listés consécutivement.

Indications :

- définir dans le code javascript les trois premières entrées d'une couleur et les compléter par la valeur 1 pour le canal alpha (transparence) dans le code des *shaders*.
- répondre avant tout à la question suivante : la couleur, dans les *shaders*, doit-elle être définie comme un uniform ou comme un attribut (ou/et un `varying`)?

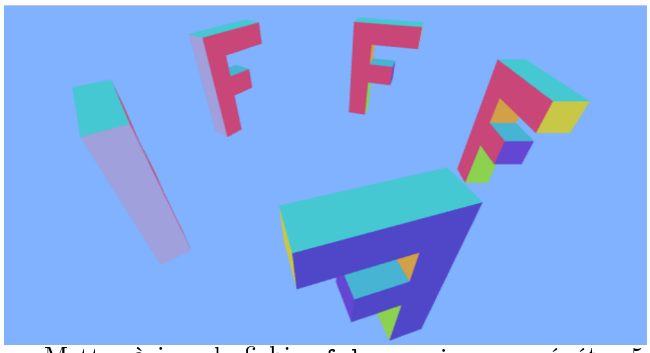
(2) Changer le point de vue sur la scène

Pour mieux voir les différentes couleurs ainsi appliquées, il convient de changer le point de vue de la caméra et/ou la scène (la position du F 3D). Mettre à jour le fichier `f-henge.js` pour obtenir l'image désirée. Vous pourrez jouer sur tout ou partie des paramètres suivants :

- axe de visée de la caméra (rotation);
- localisation de la caméra le long de cet axe de visée (translation);
- localisation du F 3D (translation);
- orientation du F 3D (rotation).

Copier le fichier `f-henge.js` en `f-henge-3.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 4 : Positionner plusieurs copies du F 3D en *F-henge*



Objectif

- comprendre quelles matrices il faut mettre à jour pour dessiner plusieurs copies d'un même objet.

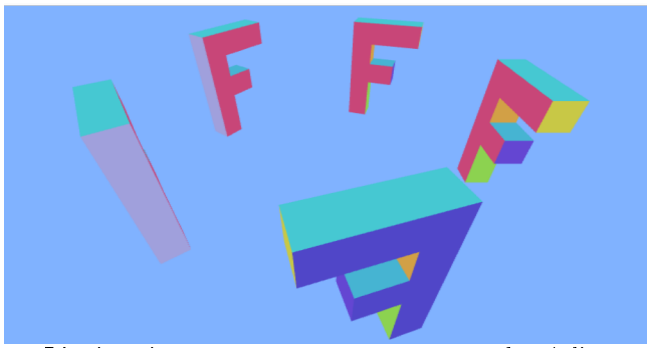
Mettre à jour le fichier `f-henge.js` pour répéter 5 fois l'instruction de dessin des primitives triangulaires d'un F 3D en ne modifiant à chaque itération que la matrice *modelViewMatrix*. Plus précisément, la caméra ne bougeant pas d'une copie à l'autre, seule la matrice `objectMatrixWorld`, qui décrit le changement de position de l'objet, doit être re-définie.

Suivre les instructions suivantes :

- Les 5 copies doivent être positionnées uniformément sur un cercle de rayon `radius = 200`.
- Chaque copie doit être orientée pour que sa face-avant soit parallèle au plan vertical tangent à ce cercle, du côté intérieur du cercle.
- La caméra doit être dans le plan $x = 0$, pointée vers le centre du cercle, selon un angle de 40° vers le bas, à une distance proportionnelle au rayon du cercle (disons égale à 2`radius`).

Copier le fichier `f-henge.js` en `f-henge-4.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 5 : Animer la scène



Objectif

- réviser la mise en place d'une animation en WebGL vue lors du TP1.
- comprendre quelles matrices il faut mettre à jour d'un frame à l'autre.

L'animation que nous vous proposons de réaliser est de faire tourner la caméra au-dessus du *F-henge*, de telle sorte qu'elle vise toujours le centre du cercle selon le même angle de 40° vers le bas, et à la même distance au centre : seul le plan vertical contenant sa demi-droite de visée tourne autour de l'axe Y au fil de l'animation.

Mettre à jour le fichier `f-henge.js` pour :

- Introduire une boucle d'animation : s'inspirer de ce qui a été fait lors du TP1.
- Ajouter une variable représentant l'angle du plan vertical décrit ci-dessus par rapport à sa position initiale, et l'incrémenter à chaque nouveau dessin de l'image.
- Identifier quelle matrice il faut mettre à jour avec cette variable (par une opération matricielle).

Copier le fichier `f-henge.js` en `f-henge-5.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.