WebGL R5.A.06

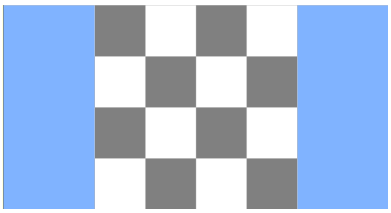
TP Extra N°1 : Texture et traitement d'image

Conseils

- Comme vous en avez désormais l'habitude, pour pouvoir utiliser les modules ES, il convient de lancer le serveur de développement par la commande :
  `$ npx vite`
  puis d'accéder par `firefox`, à l'url locale affichée dans le terminal. **Attention**, c'est le fichier `index.html` qui est lu.

Objectif général. L'objectif de ce TP est de comprendre d'une part l'application de texture en WebGL, et d'autre part comment le rendu d'une scène 3D peut être sauvegardé dans une texture pour subir ultérieurement un traitement d'image défini par un *fragment shader*. Ce sera appliqué pour donner un effet *Fish Eye* au rendu de scène 3D réalisé lors du TP N°2.

Exercice 1 : Appliquer une texture définie dans le code-même



Objectif

- Savoir comment appliquer une texture définie dans le code-même.
- Comprendre les paramètres de sur ou sous-échantillonnage d'une texture.

(1) Un carré sur lequel la texture sera appliquée.

Il s'agit ici de compléter le fichier `fisheye.js` pour afficher un carré recouvrant la surface de projection $[-1;1] \times [-1;1]$ transmise par le *vertex shader* au *fragment shader*. Plus précisément :

- Compléter le *vertex shader* pour qu'il :
  - reçoive un attribut de type `vec2` contenant la position du *vertex*.
  - transmette la valeur de cet attribut au *fragment shader* via un *varying*.
  - complète cet attribut pour définir la valeur de `gl_Position` (de type `vec4`).

  Compléter, dans le code Javascript, la récupération de la localisation de cet attribut.
- Compléter le *fragment shader* pour qu'il :
  - reçoive un *varying* de la part du *vertex shader* contenant la position du fragment dans $[-1;1]\times[-1;1]$. Nous n'en ferons rien dans un premier temps.
  - renvoie la couleur blanche par la variable `out vec4 outColor`.
- Compléter le code Javascript pour définir le buffer contenant les valeurs de l'attribut de position (à destination du *vertex shader*) :
  - Il faut transmettre les coordonnées 2D des sommets de 2 triangles composant le carré $[-1;1]\times[-1;1]$.
  - Vous pourrez vous inspirer de ce que vous avez réalisé lors des TPs précédents. Notez que nous avons déjà fourni le code de création du VAO où sera enregistré le buffer et la description de son parcours.
- Nous avons déjà fourni le code de dessin de l'image: vous devriez voir un carré blanc étendu au canvas, lui-même étendu par propriété CSS à la taille de la fenêtre.

(2) Appliquer la texture.

Dans le fichier `fisheye.js`, repérer les lignes concernant la création de texture qui se découpe ici en trois temps :

- La création de l'objet texture, identifié comme l'unité de texture de niveau 0 (nous ne verrons malheureusement pas la gestion de plusieurs unités de texture dans ce TP).
- La création des pixels d'une image (ici blanche et grise), utilisée pour la texture de niveau 0.
- La configuration de cette texture pour gérer son application (problèmes d'échantillonnage ou de coordonnées hors domaine $[0;1] \times [0;1]$).

Il s'agit à présent de plaquer cette texture sur le carré blanc construit ci-dessus. Pour cela :

- Compléter le *fragment shader* pour qu'il :
  - déclare un *uniform* `u_textureSampler` de type `sampler2D` qui récupèrera le niveau de texture à utiliser;
  - définisse des coordonnées de texture, appartenant à $[0;1] \times [0;1]$, à partir de la position transmise par le *varying* défini dans l'étape précédente. Une équation (où le *varying* est nommé `v_position`) qui peut convenir est :

    ```
    vec2 texelCoordinates = v_position.xy / 2.0 + vec2(0.5, 0.5);
    ```

  - définisse la couleur du fragment par l'appel à la fonction `texture(u_textureSampler, texelCoordinates)`.
- Compléter le code Javascript pour :
  - récupérer la localisation de l'*uniform* `u_textureSampler`;
  - lui donner la valeur `0` dans la partie du code concernant le dessin de l'image, après le rappel du programme utilisé.

Vous devez observer que le carré n'est pas carré. C'est une déformation que vous avez déjà rencontrée lors du TP 2 et que l'on peut corriger comme suit :

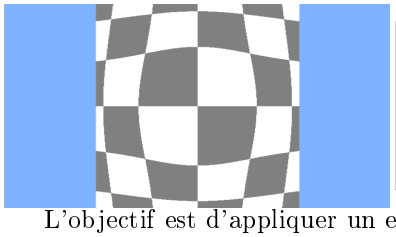
- Compléter le *vertex shader* pour qu'il :
  - déclare un *uniform* `u_aspect` de type flottant qui récupèrera le ratio hauteur/largeur de la fenêtre d'affichage;
  - multiplie avec ce ratio, la composante `x` de l'attribut de position reçu depuis le code Javascript.
- Compléter le code Javascript pour :
  - récupérer la localisation de l'*uniform* `u_aspect`;
  - lui donner la valeur `gl.canvas.clientHeight/gl.canvas.clientWidth` dans la partie du code concernant le dessin de l'image, après le rappel du programme utilisé.

Il nous reste à expérimenter les différents filtres utilisables lors de l'application de la texture :

- Ici, la texture de 4 pixels par 4 pixels est sur-échantillonnée pour définir les pixels du carré affiché. Le filtre utilisé est identifié par `gl.TEXTURE_MAG_FILTER` et le filtre proposé est `gl.NEAREST` qui choisit la couleur du texel le plus proche. On pourrait utiliser également une interpolation linéaire par `gl.LINEAR`. Observer l'effet d'un tel choix et constater qu'il ne convient pas à cette image de texture.
- Les filtres de définition de texel lorsque les coordonnées n'appartiennent pas au domaine $[0;1] \times [0;1]$ ne peuvent pas être testés dans cette première version.

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

Exercice 2 : Utiliser le *fragment shader* pour appliquer un traitement de type *Fish Eye*



Objectif

- Définir un *fragment shader* pour appliquer un algorithme de traitement d'image.
- Comprendre les filtres d'extension d'une texture.

L'objectif est d'appliquer un effet de *Fish Eye* sur l'image, consistant à appliquer un zoom dont l'intensité dépend de la distance du point au centre de l'image. Plus précisément, en un point (x, y) dont la distance au centre l'image est d, nous appliquons la couleur du texel dont la position par rapport au centre de la texture est donnée par $(\theta * x, \theta * y)$ avec $\theta = \arctan(c * d)$ où c est un paramètre de la transformation.

En particulier, en considérant des repères dont l'origine est le centre de l'image (ou de la texture), la couleur des points $(x, 0)$ avec $x \in [0; 1]$ est donnée par le texel de coordonnées $(x_t, 0)$ avec $x_t = f(x) = (x * \arctan(c * x), 0)$. Pour $c = 2.0$, afficher sur `https://www.geogebra.org/` la courbe d'équation $y = f(x) = x \tan^{-1}(2x)$ et la droite $y = x$ afin de comprendre que :

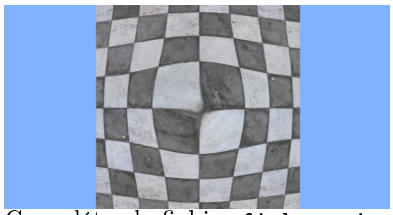
- La courbe reste croissante pour qu'un même texel de coordonnée $x_t = f(x)$ ne soit jamais affiché en deux positions x différentes (la texture serait alors comme localement répétée avec effet mirroir).
- Lorsque la courbe est sous la droite $y = x$, la couleur affichée sur une position x est celle d'un texel $f(x)$ se situant plus proche du centre de la texture : la texture semble donc dilatée, créant un effet de zoom d'autant plus important que les deux courbes sont éloignées.
- Lorsque la courbe passe au-dessus de $y = 1$, le texel $f(x)$ est en dehors du domaine de définition de la texture et ce sont alors les filtres d'extention de la texture qui définissent la couleur.

Compléter le fichier `fisheye.js` pour :

- Mettre à jour le *fragment shader* afin d'appliquer cet algorithme de déformation d'image. Vous pourrez consulter la documentation GLSL pour retrouver les fonctions qui sont à votre disposition pour calculer la distance au centre d'un point ou encore le nom de la fonction correspondant à arctan().
  `https://registry.khronos.org/OpenGL-Refpages/es3.0/`
- Jouer sur le paramètre c de la transformation pour en comprendre l'effet.
- Afin de se convaincre que certains points de l'image sont définis avec des texels en dehors du domaine de la texture, donner la couleur noire au fragment si `texelCoordinates` est en dehors de $[0.0; 1.0] \times [0.0; 1.0]$.
- Commenter ces dernières lignes, et modifier le code Javascript pour tester les différents filtres d'extension de la texture `gl.TEXTURE_WRAP_S` et `gl.TEXTURE_WRAP_T` pouvant valoir (entre autres) `gl.REPEAT` ou `gl.CLAMP_TO_EDGE`. Bien comprendre l'effet de ces filtres sur la zone qui avait été noircie dans le point précédent.

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

Exercice 3 : Utiliser une texture chargée depuis un fichier



Objectif

- Comprendre la mise en attente nécessaire pour attendre le chargement d'une texture depuis un fichier.
- Comprendre l'intérêt des filtres linéaires pour l'échantillonnage d'une texture.

Compléter le fichier `fisheye.js` pour :

- Avant de créer la texture, récupérer l'image contenue dans le fichier `FloorsCheckerboard_S_Diffuse.jpg`, et attendre que l'image soit chargée pour exécuter la suite du programme. Une manière de réaliser cela consiste à exécuter :

  ```
  const image = new Image();
  image.src = "FloorsCheckerboard_S_Diffuse.jpg";
  image.onload = function() {
    render(image);
  };
  ```

  puis à déplacer l'ensemble du code restant dans le corps de la fonction :

  ```
  function render(pixels) {
  ...
  };
  ```

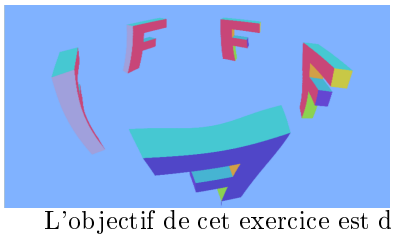
- Dans le code de création de la texture, remplacer la création du tableau `pixels` et son utilisation pour définir la texture de niveau 0 par :

  ```
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA,gl.UNSIGNED_BYTE, pixels);
  ```

- La nouvelle image texturée doit apparaître. Pour vous convaincre de la nécessité de l'attente de la fin du chargement de l'image, vous pouvez momentanément sortir la suite du code, du corps de la fonction `render()` et constater que l'image de texture n'apparaît pas.
- Enfin, au centre de l'image, vous devriez constater des blocs de niveaux de gris constants, dus au filtre `gl.NEAREST` choisi pour l'échantillonnage de la texture. Pour une image suffisamment grande et complexe comme celle-ci, il est souvent préférable d'avoir un effet de flou pour sur-échantillonner. Remplacer `gl.NEAREST` par `gl.LINEAR` et observez la différence de rendu au centre de l'image.

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

Exercice 4 : Définir une texture à partir du rendu d'une scène 3D



Objectif

- Comprendre la notion de framebuffer et son utilisation pour dessiner le rendu d'une scène 3D dans une texture.

L'objectif de cet exercice est d'appliquer l'effet *Fish Eye* sur le rendu de scène 3D réalisé lors du TP 2. Pour cela, ce rendu ne sera pas dessiné dans le *framebuffer* implicite du canvas, mais dans un *framebuffer* construit explicitement et associé à une texture qui elle, sera utilisée comme pour l'exercice 2 de ce TP.

Procéder donc comme suit :

- Copier `fisheye-2.js` en `fisheye.js` et ajouter en fin de fichier le dernier code produit lors du TP2 (`f-henge-6.js`). Retirer une des deux occurrences de la récupération du contexte WebGL depuis le canvas HTML, et corriger les éventuels noms de variables qui seraient utilisés simultanément dans les deux programmes d'origine.
- Déplacer les quelques instructions de dessin provenant de `fisheye-2.js` (rafraîchissement du `viewport`, activation du VAO, et instruction de dessin `gl.drawArrays`) en fin de la fonction récursive de dessin `draw` provenant de `f-henge-6.js`, juste avant l'instruction de récursion `requestAnimationFrame(draw);`. Avant ces quelques lignes, rappeler le programme à utiliser grâce à l'instruction `gl.useProgram(prgIP);`.
- Avant la définition de cette fonction récursive de dessin `draw`, créer un *framebuffer* explicite par les instructions suivantes :

  ```
  const fb = gl.createFramebuffer();
  gl.bindFramebuffer(gl.FRAMEBUFFER, fb);
  ```

  Par défaut, un tel *framebuffer* explicite n'a pas de *buffer* de profondeur. Il convient donc d'en ajouter un :

  ```
  const depthBuffer = gl.createRenderbuffer();
  gl.bindRenderbuffer(gl.RENDERBUFFER, depthBuffer);
  gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16, textureWidth, textureHeight);
  gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT, gl.RENDERBUFFER, depthBuffer);
  ```

  Enfin, on attache la texture `myTexture` créée dans le code provenant de `fisheye-2.js`, à ce *framebuffer* explicite :

  ```
  const attachmentPoint = gl.COLOR_ATTACHMENT0;
  gl.framebufferTexture2D(gl.FRAMEBUFFER, attachmentPoint, gl.TEXTURE_2D, myTexture, 0);
  ```

- En début de la fonction récursive de dessin `draw`, après avoir rappelé le programme utilisé pour le rendu de scène 3D, on déclare l'utilisation de ce *framebuffer* explicite à la place de celui attaché au canvas :

  ```
  gl.useProgram(prg);
  gl.bindFramebuffer(gl.FRAMEBUFFER, fb);
  ```

  De même, le `viewport` doit prendre les dimensions de ce *framebuffer* (c'est-à-dire celles de la texture).
- Dans cette même fonction, mais dans sa partie finale consacrée au traitement d'image, après avoir rappelé le programme utilisé, on déclare l'utilisation du *framebuffer* implicite attaché au canvas :

  ```
  gl.useProgram(prgIP);
  gl.bindFramebuffer(gl.FRAMEBUFFER, null);
  ```

Vous devriez être en mesure de visualiser un premier rendu qu'il reste à corriger :

- Ces carrés de couleurs sont dûs à la définition des dimensions de la texture toujours égales à 4 pixels x 4 pixels. Modifier la valeur des variables `textureWidth` et `textureHeight` pour utiliser plus de pixels pour stocker l'image issue du rendu de la scène 3D. Privilégier des puissances de 2 : 256, 512, 1024...
- Par ailleurs la mise en carré utilisée par le *vertex shader* provenant de `fisheye-2.js` n'est plus nécessaire. Retirer l'uniform contenant le ratio d'aspect, tant dans le *vertex shader* que dans le code Javascript.
- Enfin, vous pouvez expérimenter les différents filtres associés à la texture (échantillonnage ou extension hors du domaine $[0;1]\times[0;1]$), ainsi que la valeur du paramètre c de l'algorithme de *Fish Eye* pour arriver à un résultat qui vous semble satisfaisant.

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