WebGL

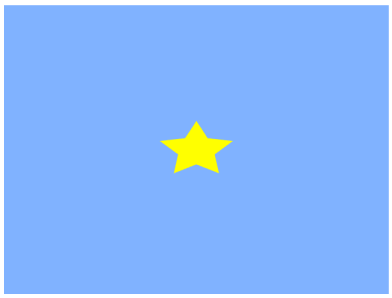
R5.A.06

TP N°1 : Apprivoiser la structure d'un code WebGL et les shaders

Conseils

- Editer vos fichiers de programme avec votre éditeur préféré.
- Visualiser le fichier `index.html` avec `firefox` en activant l'option `Outils Supplémentaires > Outils de développement` afin d'afficher la console qui contiendra d'éventuels messages d'erreur.
- Pour afficher la valeur d'une variable javascript dans la console, pour pouvez utiliser l'instruction `console.log(...)`.
- Il n'est pas possible d'afficher dans la console la valeur d'une variable des shaders.

Exercice 1 : Dessiner une étoile de couleur or sur fond bleu, à 5 branches, centrée en $(0, 0)$



Objectif

- apprivoiser la structure générale d'un programme WebGL.
- comprendre la gestion d'un **attribut** (buffer).
- comprendre la définition d'une forme par sommets (*vertices*) et faces triangulaires.

(1) Préparer le dessin de l'image

Dessiner cette étoile consiste à définir une forme d'une couleur unique (or) et composée de triangles interpolant des triplés de sommets. Ainsi :

- chaque sommet (*vertex*) possède une position qui lui est propre : le vecteur de type `vec4` (3D+1 en coordonnées homogènes) renvoyé par le *vertex shader* (qui est exécuté pour chaque sommet) est donc défini à partir d'un attribut de type `vec2` définissant la position du sommet dans le plan (x, y). Plus précisément, il suffit d'étendre les coordonnées du vertex en entrée en prenant $z = 0$ et $w = 1$.
- chaque pixel (*fragment*) inclus dans la projection de la forme sur le plan (x, y) est d'une couleur unique : le *fragment shader*, exécuté pour chacun de ces pixels, doit donc simplement renvoyer la même couleur or (définie par ses canaux R, G, B, alpha). Nous choissons de fixer cette couleur dans le *fragment shader* (et non de le récupérer via un *uniform*).

À partir de l'exemple du triangle vu en cours, compléter le code javascript et GLSL du fichier `stars.js` pour afficher l'étoile sur fond bleu (couleur du *viewport*), en faisant l'hypothèse que vous disposez d'un tableau (`Float32Array`) contenant les coordonnées (x, y) des 30 sommets constituant la forme d'une étoile. La construction de ce tableau est l'objet de la question suivante. Pour le moment, vous pouvez le déclarer et l'utiliser pour remplir un buffer de type `ARRAY_BUFFER`, dont les entrées de type `gl.FLOAT` doivent être lues deux par deux au fil des exécutions répétées du *vertex shader*. L'affichage des triangles avec la fonction `drawArrays(...)` sera à utiliser pour les primitives géométriques `TRIANGLES`.

Vous ferez attention d'écrire les lignes de code correspondant aux commentaires déjà présents : il vous sera donc nécessaire de bien comprendre chacune d'entre elles.

(2) Calculer les coordonnées des sommets de l'étoile

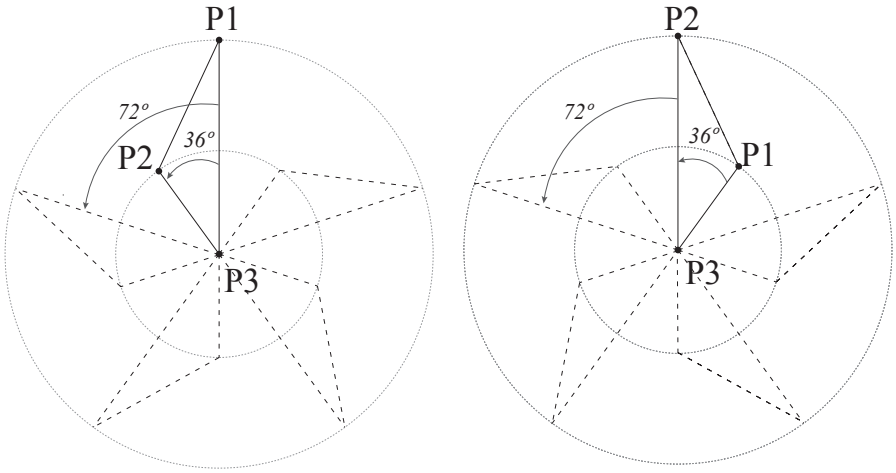


Figure 1 : Triangles à produire pour construire l'étoile.

Pour construire l'étoile, il s'agit de produire 2 séries de 5 triangles. Chacune des séries est illustrée par la Figure 1. Le triangle en trait plein est le triangle de base (composé des points P1, P2, et P3) qui sera répété par rotation de $2\pi/5$ ($= 72°$).

Pour construire la première série (Figure 1 à gauche), on utilisera les coordonnées suivantes :

- $P1 : (x, y) = (-0.2\ \sin(k2\pi/5),\ 0.2\ \cos(k2\pi/5))$ avec $k = 0...4$
- $P2 : (x, y) = (-0.1\ \sin(k2\pi/5 + 2\pi/10),\ 0.1\ \cos(k2\pi/5 + 2\pi/10))$ avec $k = 0...4$
- $P3 : (x, y) = (0, 0)$

Pour construire la seconde série (Figure 1 à droite), on utilisera les coordonnées suivantes :

- $P1 : (x, y) = (-0.1\ \sin(k2\pi/5 - 2\pi/10),\ 0.1\ \cos(k2\pi/5 - 2\pi/10))$ avec $k = 0...4$
- $P2 : (x, y) = (-0.2\ \sin(k2\pi/5),\ 0.2\ \cos(k2\pi/5))$ avec $k = 0...4$
- $P3 : (x, y) = (0, 0)$

Les deux coordonnées des trois sommets de chaque face seront ajoutées à un unique tableau, face après face. Ce tableau sera ensuite *casté* en `Float32Array` avant d'être utilisé pour remplir le buffer évoqué dans la question précédente, avec la fonction `bufferData(...)`.

(3) De l'importance de l'ordre des sommets définissant une primitive géométrique 3D

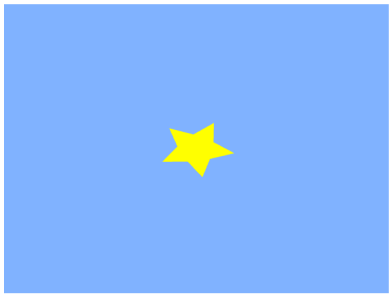
Chaque triangle possède deux faces dont, par défaut, la visibilité depuis le point de vue de la caméra est testée pour savoir si elle doit être dessinée. Ici, il est inutile de réaliser ce test pour une des deux faces. On peut configurer le *viewport* pour que seule la visibilité d'une des deux faces de chaque triangle soit testée, en exécutant l'instruction `gl.enable(gl.CULL_FACE);`.

Chaque triangle est alors considéré comme un morceau de la surface d'un objet 3D plein : seule la face du côté extérieur de l'objet peut être visible. Cette face extérieure est celle qui est visible lorsque l'ordre de définition des sommets suit le sens trigonométrique (anti-horaire). Si vous avez suivi l'ordre de création des sommets suggéré dans la question précédente, l'activation de la propriété `CULL_FACE` du *viewport* ne devrait donc rien changer à l'image dessinée.

En revanche, changer l'ordre des sommets pour quelques faces, et constater qu'elles disparaissent alors de l'image. Il convient donc à l'avenir de continuer à bien définir systématiquement les sommets d'une primitive géométrique dans l'ordre anti-horaire.

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

Exercice 2 : Tourner l'étoile de 45° dans le sens anti-horaire



Objectif
- comprendre la gestion d'un **uniform** pour le *vertex shader*.
- comprendre le stockage des matrices en GLSL.
- réviser les transformations géométriques par produit matriciel.

(1) Appliquer une matrice de rotation pour tourner l'étoile de 45°

Vous avez vu en mathématiques que, pour tourner un point du plan de 45° dans le sens anti-horaire autour de l'origine, il suffit de multiplier le vecteur de ses coordonnées par la matrice suivante :

$$\mathbf{M_{Rotation}} = \begin{bmatrix} \cos(\pi/4) & -\sin(\pi/4) \\ \sin(\pi/4) & \cos(\pi/4) \end{bmatrix} \tag{2.1}$$

Pour tourner l'étoile, il convient d'appliquer la même transformation à tous les sommets qui la définissent. Le *vertex shader* doit donc multiplier le vecteur reçu en entrée par cette unique matrice de rotation, déclarée comme `uniform` et de type `mat2`.

Pour préparer l'exercice suivant, la matrice doit être produite en javascript en définissant un `Float32Array` contenant les entrées de de la matrice de rotation, qui est ensuite associé à la variable `uniform` correspondante dans le code du *vertex shader*.

(2) De l'importance de l'ordre des entrées définissant une matrice en GLSL

Attention ! En GLSL, les matrices de type `mat2`, `mat3`, `mat4` sont décrites **par colonnes** et non par lignes. Les entrées du tableau de type `Float32Array` doivent donc correspondre à celles de la matrice de rotation, lues colonne par colonne (de haut en bas de chaque colonne, puis de gauche à droite).

Au contraire, ll est probable que vous ayez ordonné les entrées de la matrice ligne par ligne. Vous avez alors défini la matrice transposée, qui correspond à la même rotation mais dans le sens horaire. Vérifier donc que l'étoile a bien été tournée dans le sens anti-horaire. Sinon corriger la matrice définie dans le code javascript.

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

Exercice 3 : Animer l'étoile par rotation itérée

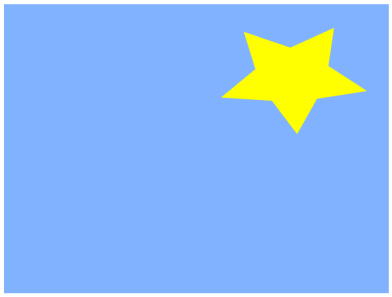
Objectif
- comprendre la réalisation d'une animation par l'instruction `requestAnimationFrame(...)`
- identifier les instructions à placer dans la fonction de dessin qui doit être itérée.

Ecrire une fonction javascript `draw()` chargée de l'animation, incluant certaines des instructions existantes. En particulier elle doit :

- Ré-initialiser la zone d'affichage WebGL avec la couleur bleu ciel.
- Définir à chaque appel une nouvelle valeur d'angle et donc une nouvelle matrice de rotation à donner en entrée du *vertex shader*. Bien choisir l'incrément de l'angle afin de réaliser une vitesse de rotation paisible.
- Afficher les triangles.
- Déclencher l'animation avec l'utilisation de la fonction javascript `requestAnimationFrame(<callback>)`.

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

Exercice 4 : Déplacer l'étoile par translation



Objectif
- comprendre l'importance de l'ordre des transformations géométriques.
- comprendre l'effet de la composante `w` des coordonnées homogènes.

(1) Translater l'étoile pour qu'elle soit centrée en $[0.5, 0.5]$

Nous choisissons de faire réaliser cette translation par le *vertex shader* (i.e. comme la rotation) par addition d'un vecteur de déplacement à la position initiale du *vertex* déjà transmise comme attribut en entrée du *shader*.

Pour préparer l'exercice suivant, ce vecteur de déplacement doit être défini dans le programme javascript, puis transmis en entrée du *vertex shader* comme une variable de type `vec2`. Il vous faut alors répondre à la question suivante : cette variable doit-elle être un `attribut` ou un `uniform` ?

Ensuite, dans le *vertex shader*, la translation doit-elle être appliquée avant ou après la rotation? Essayer les deux ordres et comprendre les deux effets ainsi produits. Choisir l'effet qui permet de centrer l'étoile en $[0.5, 0.5]$.

(2) De l'effet de la composante `w` des coordonnées homogènes

La dernière instruction de votre *vertex shader* consiste à définir la valeur de `gl_Position` comme le complément en `vec4` d'une position définie par un `vec2` (voir exercice 1). Lors de cette complétion, remplacer la valeur de 1 donnée à `w` par 2. Quel effet cela a-t-il?

Modifier la procédure principale du *vertex shader* pour que l'étoile soit deux fois plus grande grâce à la définition d'une valeur adéquate pour la composante `w`, mais toujours centrée en $[0.5, 0.5]$. *Indication : en GLSL il est possible de multiplier un scalaire de type `float` avec un `vec2`, chaque entrée du vecteur étant alors multipliée par le scalaire.*

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

Exercice 5 : Produire et afficher 12 étoiles sur une grille 4x3

Objectif
- comprendre comment dessiner plusieurs objets.

Commencer par rétablir la procédure principale du *vertex shader* pour afficher l'étoile dans sa taille initiale, centrée en $[0.5, 0.5]$.

Il s'agit à présent de répéter l'instruction de dessin des triangles `gl.drawArrays(...)` 12 fois pour dessiner 12 copies de l'étoile, centrées en 12 points d'une grille 4x3. Pour cela il suffira de modifier la valeur du vecteur de translation pour qu'il prenne les valeurs suivantes :

$$V_{Translation}=(-0.75+0.5\,j,\quad -0.75+0.75\,k) \text{ avec } j=0...3 \text{ et } k=0...2$$

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

Exercice 6 : Affecter une couleur différente à chacune des 12 étoiles

Objectif
- comprendre la gestion d'un **uniform** dans le *fragment shader*.
- comprendre l'effet de la composante `alpha` des couleurs.

(1) Changer la couleur de chaque étoile dans le fragment shader

Dans le fichier `couleurs.js` fourni, un tableau `couleurs` permet de définir 12 couleurs par leurs composantes `R,G,B`. Notez que ces composantes sont définies comme des entiers codables sur 8 bits avant d'être normalisées en flottants appartenant à $[0, 1]$.

Affecter chacune de ces couleurs à une des 12 étoiles. Pour cela, définir une variable `uniform` de type `vec4` passée en entrée du *fragment shader* depuis le programme javascript. Les trois premières entrées du vecteur correspondront à un des 12 triplets `R,G,B` définis dans le tableau `couleurs`. La quatrième entrée correspond au canal `alpha` qui code la transparence de la couleur. Choisir l'opacité totale dans un premier temps (`alpha = 1`).

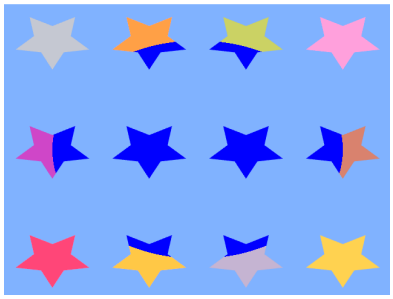
(2) Transparence, couleur du `viewport`, et fond du `canvas`

Fixer à présent la transparence à `alpha = 0`. Les étoiles deviennent rouges alors que `viewport` est bleu ciel. C'est que les éléments géométriques dessinés **remplacent** la couleur du `viewport`. En fixant la transparence à 0, c'est la couleur de fond du `canvas` définie dans le fichier `index.html` qui apparait.

Choisir une valeur de transparence qui donne aux étoiles une teinte rosée.

Copier le fichier `stars.js` en `stars-6.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 7 : Changer la couleur des pixels d'une étoile en fonction de leur position dans l'image



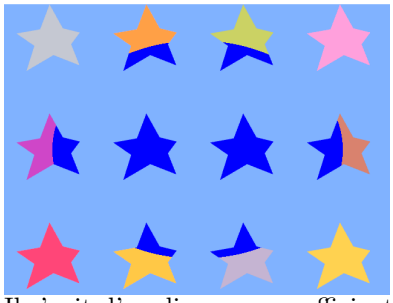
Objectif
- comprendre la gestion d'un **varying** passé du *vertex shader* au *fragment shader*.
- commencer à se familiariser avec quelques fonctions fournies par GLSL.

Dans le code du *fragment shader*, lorsque la distance ($\sqrt{x^2 + y^2}$) du pixel à l'origine (0,0) est inférieure ou égale à 0.75, émettre la couleur bleue. Pour calculer la distance, il faut :

- Récupérer les coordonnées du pixel. Pour cela, déclarer un *varying* de type `vec2` qui sera donné en sortie du *vertex shader* (`out vec2`) et en entrée du *fragment shader* (`in vec2`). Dans le *vertex shader*, affecter la position du *vertex* à cette variable. La valeur du *varying* de même nom dans le *fragment shader* sera alors calculée automatiquement par interpolation linéaire entre les valeurs défines pour les sommets de la face contenant le pixel : il correspond alors à la position du pixel dans l'image.
- Trouver dans la documentation du langage GLSL quelle fonction permet, à partir de la position d'un pixel, de calculer directement sa distance à l'origine.

Copier le fichier `stars.js` en `stars-7.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 8 : Animer la longueur des branches des étoiles



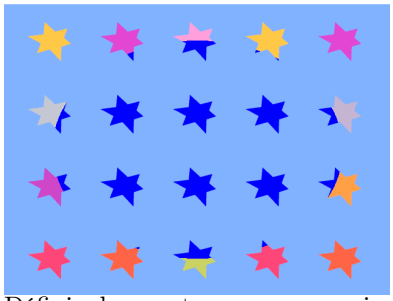
Objectif
- animer un **attribut**.
- utiliser de l'aléa.

Il s'agit d'appliquer un coefficient multiplicateur à la taille des branches de l'étoile modèle, tel que :

- les coefficients sont les mêmes d'une étoile à l'autre, mais sont différents entre les branches de chaque étoile : il faut donc introduire un nouvel **attribut** de type `float`;
- ils changent régulièrement mais à un rythme moins soutenu que celui du rafraîchissement de l'image : la mise à jour du `BUFFER` devra être réalisée dans la fonction `draw()` mais pas systématiquement (par exemple une fois tous les 50 appels de la fonction);
- lors des mises à jour, la valeur des coefficients sera tirée de façon aléatoire dans $[0.5, 1.5]$: vous pourrez utiliser la fonction `Math.random()` pour tirer un nombre aléatoire dans $[0, 1]$. Attention! La valeur du coefficient doit en revanche être la même pour les deux sommets positionnés en bout de chaque branche!

Copier le fichier `stars.js` en `stars-8.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 9 : BONUS : Paramétrer le programme



Objectif
- rendre l'affichage facilement modulable.

Définir dans votre programme javascript les 3 variables suivantes :

- le nombre de branches des étoiles;
- le nombre d'étoiles à positionner verticalement;
- le nombre d'étoiles à positionner horizontalement.

Le rayon intial L (avant animation) du grand cercle contenant les extrémités des branches d'une étoile est alors défini de telle sorte que :

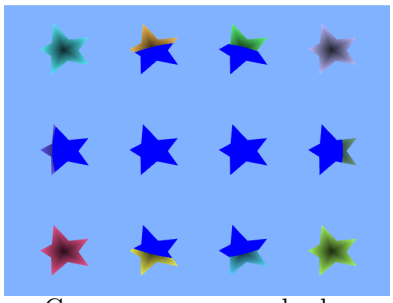
- la marge entre la grille des centres des étoiles et le bord du carré $[-1, 1] \times [-1, 1]$ est égale à $2L$ et les points de la grille sont alors régulièrement répartis à l'interieur de ces marges.
- si la grille contient plus d'étoiles en largeur qu'en hauteur, alors $3L$ est égal à la distance horizontale entre deux centres d'étoiles, sinon il s'agit de l'espacement vertical.

Le rayon du petit cercle portant les sommets de base des branches d'une étoile est quant à lui égal à $L/2$.

Vérifier votre programme en faisant varier ses 3 paramètres (par exemple 6 branches et une grille de 5 étoiles en largeur et 4 en hauteur).

Copier le fichier `stars.js` en `stars-9.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 10 : BONUS : Faire varier la couleur des pixels selon leur position locale à chaque étoile



Objectif
- le même que pour l'exercice 7.

Commencer par rendre les couleurs d'étoile opaques afin de réduire l'accumulation d'effets de couleurs.

Ensuite, si la distance du pixel à l'origine globale de l'image est plus grande que 0.75, alors il s'agit de multipler les composantes `R,G,B` de sa couleur par une valeur flottante qui dépend de la distance du pixel au centre de l'étoile :

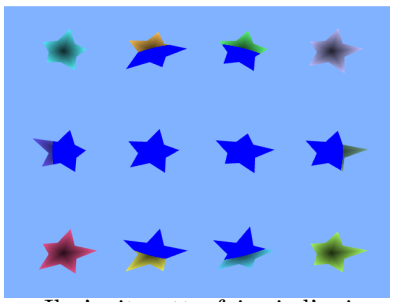
- 1.2 si le pixel est à l'extrémité d'une branche (grand cercle);
- 0.8 si le pixel est à la position d'un sommet de base d'une branche (petit cercle);
- 0.2 si le pixel est au centre de l'étoile;
- une interpolation de ces valeurs sinon.

Indications :

- dans le programme javascript, construire un *buffer* contenant autant d'entrées qu'il y a de *vertices* dans une étoile, et remplir avec ces trois valeurs en fonction de la position du *vertex*;
- dans le *vertex shader*, il suffit de transmettre cette valeur au *fragment shader*;
- dans le *fragment shader*, attention à ne pas multiplier le canal `alpha`.

Copier le fichier `stars.js` en `stars-10.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.

Exercice 11 : BONUS : Animer la longueur des branches des étoiles de façon indépendante d'une étoile à l'autre



Objectif
- le même que pour l'exercice 8, en plus complexe.

Il s'agit cette fois-ci d'animer la longueur des branches des étoiles de façon indépendante et même non synchronisée. Pour cela,

- conserver dans le programme javascript un tableau contenant un coefficient multiplicateur pour chaque sommet de la scène totale (et non une entrée pour chaque sommet d'une étoile), dont toutes les entrées sont initialisées à 1.
- à un instant donné (correspondant à la periode définie dans l'exercice 8, par exemple 50 appels de la fonction `draw()`), seules les entrées de ce tableau correspondant à une étoile donnée sont mises à jour (par la même fonction d'aléa utilisée dans l'exercice 8).

Copier le fichier `stars.js` en `stars-11.js` afin de garder trace de cette étape et faciliter le suivi asynchrone de votre travail par l'enseignant.