TP n°2 : WebGL

Le but du TP est de découvrir les bases de WebGL : VBO, shaders et transformations. C'est un mélange entre observation des sources fournis et ajout de fonctionnalités.

Téléchargez l'archive du TP2 qui est sur Moodle et décompressez-la. Il est possible que les raccourcis du dossier libs (liens logiques entre dossiers) ne fonctionnent pas. Vous devrez alors soit les refaire, soit recopier les dossiers directement.

Il faut vous assurer que votre navigateur implémente correctement JavaScript 6 (classes, variables locales affectées avec let) et WebGL2 : https://webglreport.com/?v=2. Au pire, il faut que WebGL1 marche sur votre PC : https://webglreport.com/?v=1.

1. Débuts avec JavaScript 6 : 00-JavaScript

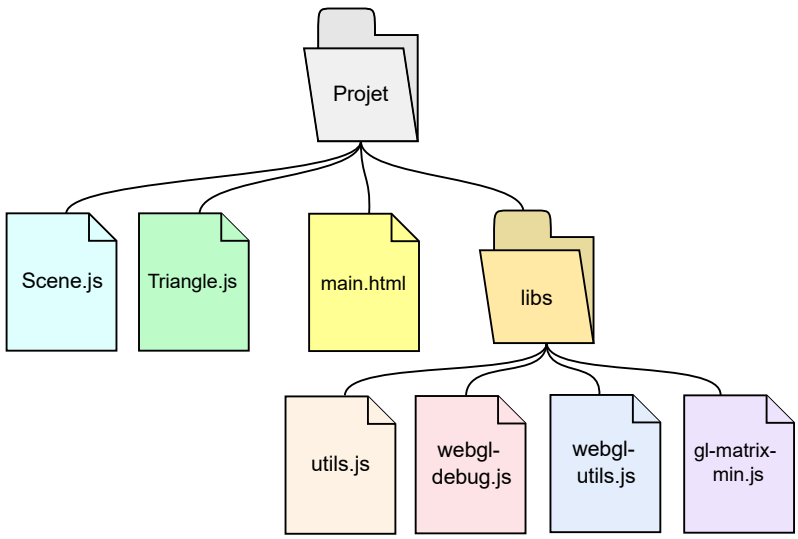
Pour commencer, un mini-projet pour découvrir JavaScript 6 et en particulier le mode d'emploi des classes. Le dossier comporte un source HTML à ouvrir avec le navigateur et deux sources JavaScript qui sont inclus par le HTML.

Ouvrez la console (F12, puis onglet Console sur Firefox). À noter que le débogueur va assez peu servir. La mise au point se fera surtout avec les messages émis sur la console : console.log(…), console.info(…), console.error(…)

Les manipulations sur cet exemple seront définies en séance, en fonction des questions.

2. Bases du dessin avec OpenGL : 01-Dessin2D-1triangle

Chaque projet WebGL est composé de cette manière :



- Le dossier `libs` de chaque projet est un raccourci (lien logique) vers le dossier du même nom au dessus, afin de ne pas dupliquer les fichiers qu'il contient. Dans le chapitre 3, il y aura un dossier `data` similaire.

  Sur Windows, il faudrait remplacer ces deux dossiers par des raccourcis.

  Mais ensuite, selon les systèmes d'exploitation, le navigateur peut déclencher une erreur « cross-origin resource sharing », voir

https://developer.mozilla.org/fr/docs/Web/HTTP/CORS. Pour éviter cette erreur, il y a un script Python à lancer dans le dossier du projet, `server.py` et qui consiste en un serveur HTTP sur http://localhost:8000 — cette page correspond à `main.html`, et toutes les références à `libs` se font dans le dossier au dessus. Au pire, si ce script ne fonctionne pas, vous devrez supprimer tous les raccourcis et recopier le dossier `libs` dans chacun des projets (mais penser à retirer ces dossiers du ZIP pour rendre le TP).

- Le fichier html5 `main.html` organise l'affichage du projet : initialisation WebGL et création d'une instance de `Scene`, puis affichage.
- `Scene` et `Triangle` sont les deux classes essentielles du projet. Il y a énormément de concepts à découvrir. En règle générale, le constructeur crée des ressources OpenGL qui subsistent durant tout le logiciel. Des méthodes comme `onDraw` utilisent ces ressources pour dessiner. JavaScript ne connaît pas la notion de destructeur, mais il en faut quand même un, alors j'ai défini une méthode `destroy` à appeler explicitement quand l'instance n'est plus nécessaire.

Voici maintenant une description des appels à OpenGL dans chacune des classes.

2.1. Classe Scene

a) Constructeur

`gl.clearColor(rouge, vert, bleu, alpha)` : configure la couleur d'effacement de l'écran. Les composantes sont entre 0 et 1. L'effacement n'a lieu que lors d'un appel à `gl.clear(gl.COLOR_BUFFER_BIT)`. Cette méthode remplit le « color buffer » avec la couleur prédéfinie par `gl.clearColor`. Le color buffer est un *pixmap* constituant l'image qu'on voit à l'écran. D'autres buffers sont également associés à l'écran, mais ils sont invisibles ; on les étudiera au fur et à mesure.

b) Méthode onSurfaceChanged

Cette méthode est appelée lors de l'affichage initial du canvas, ou si vous avez prévu que le canvas puisse s'agrandir. Dans un programme C ou Android, elle serait également appelée lors du redimensionnement de la fenêtre, par exemple lors d'une rotation de l'écran sur un smartphone.

Son rôle est de spécifier la zone de dessin OpenGL : le « viewport » à l'aide de `gl.viewport(x0, y0, largeur, hauteur)`. Un programme OpenGL peut dessiner dans plusieurs rectangles sur le color buffer, mais la plupart du temps, on ne définit qu'un seul viewport.

Dans un prochain exemple, nous verrons qu'on peut aussi définir la projection en perspective ; elle a besoin de la taille de la fenêtre pour garder le même champ de vision.

Le nom de cette méthode ainsi que la suivante proviennent de l'API Android, ce sont ces noms qu'on doit leur donner.

c) Méthode onDrawFrame

C'est la méthode qui est appelée par `main.html` pour dessiner la scène.

d) Destructeur

Il doit supprimer toutes les ressources OpenGL allouées dans le constructeur, et de préférence dans l'ordre inverse de leur création. Sur vos classes, il faut appeler la

méthode `destroy` récursivement, tandis que sur les objets OpenGL (buffers, shaders), il faut appeler une fonction OpenGL.

2.2. Classe Triangle

C'est la classe la plus complexe car elle fait appel à de nombreux concepts d'OpenGL. Dans les premières versions d'OpenGL, il était beaucoup plus simple de dessiner, mais maintenant, on doit utiliser des VBO et des shaders.

a) Constructeur

Il y a deux parties : créer les shaders et créer les VBO. Les VBO contiennent les coordonnées des sommets. Cela peut être des couples (x,y), des triplets (x,y,z) ou des quadruplets (x,y,z,w). Dans ce premier projet, ce sont des couples car on dessine en 2D.

- Les shaders sont un couple *vertex shader* et *fragment shader*. Le premier s'occupe des coordonnées des sommets du triangle ; le second s'occupe du calcul de la couleur.

  Pour chacun d'eux, on crée une chaîne de caractères JavaScript en utilisant la syntaxe `` `...` ``. Cette syntaxe permet d'inclure des expressions dynamiques avec la syntaxe `${expression}`. Le mot-clé `dedent` permet de créer une chaîne multi-ligne indentée, mais ramenée à la marge, c'est à dire sans les espaces du début de chaque ligne. C'est pour une meilleure présentation dans les logs de la console.

  À noter qu'il y a un problème lorsque le navigateur n'implémente pas correctement WebGL2 : les shaders écrits pour WebGL2 ne passent pas en WebGL1. J'ai donc prévu deux versions (V1 et V2), mais c'est très lourd. Si votre navigateur accepte le WebGL2, vous pourrez supprimer ce qui concerne la version V1.

  NB : l'an dernier, la totalité des PC acceptaient WebGL2, donc j'ai enlevé les shaders V1 des TP suivants.

- Le *vertex shader* reçoit les coordonnées de chaque sommet sous la forme d'un `vec2`. C'est un type prédéfini du langage GLSL. Regardez rapidement la fiche `webgl20-reference-guide.pdf`. Cherchez le cadre intitulé *Types* sur la page 6. Il vous donne les noms des types prédéfinis. Dans le cadre *Qualifiers*, vous avez des mot-clés comme `in`, `out` et `uniform`. `in` indique que la variable provient d'un VBO – on l'appelle aussi *variable attribute*, `out` indique que la variable va vers le fragment shader (il n'y en a pas dans le premier exemple) et `uniform` signifie que la variable est affectée par le programme JavaScript (idem, aucune dans cet exemple).

  Comme vous voyez, le vertex shader se contente de recopier les coordonnées 2D qu'il reçoit dans `gl_Position`. Cette dernière est présentée dans la fiche, cadre VERTEX SHADER VARIABLES. Elle est de type `vec4`, c'est pourquoi le shader doit convertir les coordonnées 2D en 4D – on verra pourquoi 4 et non pas 3D.

  Dans un programme plus élaboré, c'est ici qu'il y a des calculs de projection en perspective sur l'écran. Dans ce premier projet, la projection est parallèle (cavalière) et les coordonnées 2D deviennent directement des coordonnées écran.

- Le fragment shader se contente de retourner une couleur constante dans `glFragColor`. Cette variable est définie par le qualificatif `out`. Dans GLSL version 1.0, la variable était prédéfinie et s'appelait `gl_FragColor`.

  Amusez-vous à changer la couleur dans le source du shader.

Ces deux sources sont compilés par OpenGL, à l'aide de la fonction `Utils.makeShaderProgram` dont vous pouvez aller voir le source. Avec Vulkan, on ne fournit plus les sources, mais du code intermédiaire déjà compilé avec les outils du SDK. Au final, on récupère un objet représentant le programme, couple des shaders.

Pour découvrir les messages d'erreurs, vous pouvez dès à présent rajouter une scorie dans l'un des shaders, par exemple enlevez le `_` de `gl_Position`. Affichez la console de mise au point JavaScript (firebug par exemple, onglet console). Selon les pilotes graphiques, vous aurez un message comme « Vertex Shader of Triangle: ERROR: 0:5: 'glPosition' : undeclared identifier ». Normalement, 0:5 signale une erreur ligne 5, mais vous aurez peut-être une autre syntaxe. Vous avez le source numéroté en dessous. Certaines erreurs ne sont pas signalées par une interruption du logiciel ; elles peuvent être très difficiles à trouver.

Après avoir compilé et lié les deux shaders, il faut effectuer une dernière opération : déterminer l'emplacement des variables d'entrée du vertex shader. Ces variables devant être liées à des VBO, il faut pouvoir indiquer à OpenGL : « connecte telle variable à tel VBO ». Pour cela, on fait appel à `gl.getAttribLocation(shader, nom_var)` qui retourne un objet identifiant la variable. Cet identifiant sera utilisé pour associé le VBO.

Ensuite, le constructeur crée un VBO, c'est à dire un tableau de coordonnées. Ce tableau doit avoir deux colonnes car la variable du shader qui lui sera liée est un vec2, et ce tableau doit avoir trois lignes, car il y a trois sommets dans un triangle. Il faut faire attention à cela quand on crée des VBO. Amusez-vous à rajouter une valeur dans le tableau : sans effet ; et à en enlever une : « Bound vertex attribute buffers do not have sufficient size for given first and count ».

Notez que l'identifiant du shader, de la variable et du VBO sont mémorisés dans des variables d'instance (`this.m_ShaderId`, `this.m_VertexLoc` et `this.m_VertexBufferId`) tandis que les variables temporaires ayant servi à les créer ne le sont pas.

b) Méthode onDraw

Son principe est simple : elle active le shader, elle active le VBO en l'associant à la variable glVertex du vertex shader, elle dessine un triangle et elle désactive le shader et le VBO. L'activation du shader et du VBO sont toujours les mêmes. Il y a plusieurs fonctions OpenGL à appeler. On aurait pu les caser dans utils.js.

Le dessin d'un ou plusieurs triangles se fait avec `gl.drawArrays(gl.TRIANGLES, 0, nb_sommets);`. Le nombre de sommets doit être un multiple de 3.

Changez la primitive de dessin : au lieu de `gl.TRIANGLES`, mettez `gl.POINTS` ou `gl.LINE_LOOP`. Cette dernière fait dessiner un contour fermé.

Dans cet exemple, le shader et le VBO pourraient être activés définitivement dans le constructeur, au lieu d'à chaque appel de `onDraw`. Mais dans un projet impliquant plusieurs objets, il faut correctement commuter de l'un à l'autre sans rien laisser traîner. Dans OpenGL, tout est persistant, comme le mode de calcul des angles dans une calculatrice.

3. Interpolation vertex-fragment shaders : 02-Dessin2D-gradient

Le deuxième projet propose une variante. Seule la classe `Triangle` a été modifiée : un second VBO contenant des couleurs pour chaque sommet a été rajouté.

Cet ajout oblige à modifier le calcul des couleurs. La couleur étant définie par sommet, elle doit être transmise et interpolée du vertex shader vers le fragment shader. Pour cela, on fait appel à une variable spéciale, identique dans les deux shaders, et qualifiée avec le mot-clé `out` dans le vertex shader et `in` dans le fragment shader.

Constatez que les couleurs sont interpolées exactement comme les coordonnées.

Exercice : reprendre le premier exemple et au lieu d'interpoler des couleurs entre les sommets, interpoler un coefficient 0..1 à multiplier aux composantes de la couleur fixe.

4. Dessin de plusieurs objets : 03-Dessin3D-2triangles

On s'intéresse au dessin de plusieurs objets. Ce projet définit deux classes de triangles – on aurait pu paramétrer la création d'un triangle et l'instancier deux fois différemment.

OpenGL dessine les objets séquentiellement, le plus récemment dessiné écrase le précédent. Modifiez l'ordre de dessin dans la méthode `onDraw` et constatez.

Pourtant, cette fois, les triangles sont définis en 3D. Chaque sommet a une coordonnée z supplémentaire. Regardez ce que vaut cette coordonnée et quelle superposition on aurait systématiquement dû obtenir.

Maintenant, on va activer un dispositif appelé *depth buffer*. Il s'agit d'un tableau qui mémorise la profondeur (distance z) de chaque pixel dessiné sur l'écran. En configurant un test interne de la carte graphique, un pixel ne pourra être dessiné que s'il est plus proche que ce qui a déjà été dessiné au même endroit. La mise en place de ce test est actuellement en commentaires dans le constructeur de `Scene`, décommentez-la, puis recommencez à intervertir les lignes qui dessinent les triangles. À noter que les deux dernières instructions sont inutiles car c'est la configuration par défaut.

Modifiez les deux lignes en : `gl.depthFunc(gl.GREATER); gl.clearDepth(0.0);` et constatez que le test de profondeur a été inversé. Remettez comme c'était.

Maintenant, modifiez les triangles pour leur donner une inclinaison en profondeur : au lieu d'un z constant, mettez +0.5, 0.0 et -0.5 aux différents coins – il faut que les triangles s'entremêlent.

5. Transformation géométrique : 04-Dessin3D-transformation_glsl

Dans ce projet, on rajoute une transformation géométrique. Elle est définie par une matrice construite par le vertex shader et appliquée aux coordonnées 2D des sommets. La matrice définit une rotation. Le prochain cours présentera les matrices et leurs relations avec les points et vecteurs.

Une chose à noter est l'emploi d'une variable *uniform*. C'est une sorte de paramètre défini par le programme JavaScript avant le dessin de primitives. Ainsi tous les sommets reçoivent la même valeur. Ici, c'est le nombre de secondes écoulées depuis le lancement de la page qui permet de paramétrer l'angle de rotation. De ce fait, le triangle est animé.

Exercices :

- Remplacer la rotation par une translation en x qui produit un balancement du triangle
- Remplacer la rotation par une mise à l'échelle qui fait pulser le triangle : il devient petit, puis grand, puis à nouveau petit…
- Au lieu de modifier la géométrie du triangle, faites en sorte que le temps altère les couleurs du triangle : plus sombre alternativement plus clair…

- Question de réflexion : et s'il fallait dessiner plusieurs triangles animés tous de la même manière… Dans le même registre : en fait, chaque sommet fait définir une matrice qui est exactement la même que celle des autres sommets. Et qu'en est-il des transformations 3D ?

6. Transformation géométrique : 05-Dessin3D-transformation_js

Cet exemple est une variante du précédent afin de répondre à la question de réflexion. La matrice de rotation est produite par le programme JavaScript et transmise au vertex shader par une variable *uniform*.

On commence par regarder la méthode `onDrawFrame` de la classe `Scene`. Elle utilise la bibliothèque `gl-matrix` (voir le dossier docs). Le principe de cette librairie est assez inhabituel : on crée une matrice qu'on passe en paramètre d'entrée et/ou de sortie à des fonctions qui l'altèrent comme `rotateY`, `translate`… On reviendra sur cette bibliothèque au prochain cours. Ici, on crée une rotation qui est fournie aux méthodes `onDraw` des triangles. Comme précédemment, cette rotation est paramétrée par le temps écoulé.

Normalement toutes les matrices et tous les vecteurs doivent être créés dans le constructeur de la classe qui les manipule. C'est partiellement fait ici : la matrice est créée mais pas les vecteurs vec3. Ça ne l'est pas dans les projets suivants, parce que c'est assez contraignant. Pourtant, c'est une clé de rapidité des programmes WebGL.

Chaque triangle possède le même vertex shader : il définit une variable `uniform mat4 matrix` destinée à recevoir la matrice de rotation. Cette matrice est appliquée aux coordonnées 3D venant du VBO.

L'emplacement des variables *uniform* et *attribute* (`in` du vertex shader) est mémorisé car il faut les associer l'une à la matrice, l'autre au VBO, dans la méthode `onDraw`.

La fourniture de la matrice se fait avec la fonction `mat4.glUniformMatrix(loc, mat)` qui n'est pas dans gl-matrix mais dans utils.js.

Notez qu'il y a une sorte de distorsion de l'image, car il n'y a pas de perspective.

Décommentez maintenant la translation et commentez la rotation. Regardez comment la translation est construite à partir d'un vec3 (consultez la documentation).

Maintenant, le but est de comprendre quelle transformation est construite quand on décommente les deux : translation et rotation. Elles sont composées dans un certain ordre. Intervertissez les deux instructions et comparez les résultats. À méditer.

Encore une expérience : il y a deux lignes commentées dans le constructeur : `gl.enable(gl.CULL_FACE)` et la suivante. Ces lignes font disparaître les triangles qui sont vus de dos. C'est intéressant pour les objets ayant des facettes opaques de tous les côtés. Ici, pour qu'on voit quelque chose des triangles vus de dos, j'ai rajouté le dessin d'un LINE_LOOP.

Dernière expérience, ne laissez que la première rotation en Y qui entraîne les deux triangles ensemble, et modifiez les profondeurs des deux triangles dans leurs VBO : mettre la même profondeur de 0.0 à tous les sommets. Les clignotements qui apparaissent sont appelés *depth fighting* (combat en z). Ils sont dus à des instabilités d'arrondi des calculs et au parallélisme dans la carte graphique.

7. Primitives indexées : 06-Dessin3D-index

On arrive à une autre manière de construire des objets, à utiliser dès que les mêmes sommets sont utilisés dans plusieurs triangles. Au lieu de recopier plusieurs fois les mêmes données, on construit un VBO supplémentaire contenant des numéros : ceux des sommets à employer pour chaque primitive (triangle, segment…). La fonction de dessin est un peu différente : `gl.drawElements(primitive, nombre, type, 0)` le premier paramètre indique ce qu'il faut dessiner. Vous pouvez mettre `gl.LINES` à la place de `gl.TRIANGLES` dans la méthode `onDraw` du tétraèdre. Le deuxième paramètre indique combien il y a de données = nombre de sommets * nombre de réels par sommet. Le troisième paramètre donne le type des numéros de sommets.

Exercice : dessiner un cube en primitives indexées en vous inspirant du tétraèdre. Attention, chaque face est un carré comprenant deux triangles. La bonne manière de faire consiste à faire un schéma numérotant les sommets.

8. Dessin en perspective : 07-Dessin3D-perspective

Pour finir, voici un début de mise en place de la caméra. Tout sera expliqué la semaine prochaine. Le principe général est de transformer les objets par une matrice qu'on décompose en trois transformations :

- *Projection* : une projection perspective sur l'écran. Elle est constante pour toute la scène, et définie dans la méthode `onSurfaceChanged` de `Scene`. La matrice dépend des dimensions de la vue OpenGL, c'est pour ça qu'on l'initialise dans cette méthode. Les deux derniers paramètres de `mat4.perspective` sont la plus proche distance acceptée, et la plus lointaine. Les objets plus proches ou plus loin seront tronqués. Amusez-vous à changer les valeurs pour voir les effets.
- *View* : une combinaison de translations et de rotations représentant la position et l'orientation de la caméra. Cette matrice est fixe, elle pourrait être affectée
- *Model* : une combinaison de translations et de rotations représentant la position et l'orientation de l'objet par rapport à la caméra.

Trouvez les différents calculs de positionnement de l'objet à la fois dans la classe `Scene` et dans la classe `Tetraedre`. Vous devez comprendre que la position écran de chaque sommet est : P*V*M*sommet et c'est le vertex shader qui la calcule.

Il faut savoir que dans certains cas, on ne fournit qu'une seule matrice, P*M*V au vertex shader, afin qu'il ait le moins possible de calculs à faire (un produit de matrice est coûteux et ici, il est constant pour tout l'objet). Dans d'autres cas, il faut impérativement fournir P et M*V séparément au vertex shader.

Le fragment shader du tétraèdre comprend toute une partie commentée : décommentez-la et supprimez l'instruction qui était active à la place. Voici l'un de vos premiers effets : une brume basée sur la distance du fragment à la caméra.

Pour mettre au point cet effet (régler la fusion des couleurs), décommentez cette instruction : `gl_FragColor = vec4(dist-9, 9-dist, 0, 1);return;` Elle colore en rouge lorsque la distance dépasse 9 et en vert lorsque la distance est plus petite que 9. L'autre instruction similaire colore en fonction du brouillard. Il est maximal quand c'est dessiné en blanc.

9. Exercices

Voici maintenant une rafale d'exercices à faire (et à déposer sur Moodle) en se basant sur toutes ces notions. Copiez chaque fois l'un des projets, probablement le dernier, et renommez-le selon l'exercice.

9.1. Formes

- Dessiner une pyramide à base carrée, ses côtés font 1 unité, sa hauteur est de 2 unités et ses couleurs très vives (comme le tétraèdre).
- Dessiner un quadrillage plan, une sorte de grille formée de lignes. Dans cet exercice, les sommets ne sont pas fournis de manière constante, mais générés par une itération. Il y a une boucle en X et une boucle en Z, chacune par exemple de -2 à +2, tous les 0.5. Les dimensions et pas de la grille sont fournis en paramètres au constructeur. On utilise `push` pour ajouter un élément en fin de tableau.
- Dessiner un disque. Pour cela, il faut créer des triangles rayonnant autour du centre. On fournit le nombre de triangles au constructeur. Les couleurs des sommets peuvent être définis à l'aide d'une équation, ou avec la fonction hsv2rgb de la bibliothèque utils.js. Il suffit de fournir un vec3 ayant la composante x variant de 0 à 1, et les composantes y et z fixes, valant 0.5 par exemple pour générer un arc en ciel.
- Dessiner un cône. C'est à peine plus complexe. C'est un disque dont le centre n'est pas coplanaire avec les autres points. Et il y a un deuxième disque plan dessous.

9.2. Scènes

- Dessiner une scène comprenant une grille. La caméra doit avoir une vue plongeante sur cette grille. Ensuite, il faut positionner quelques objets dessus comme si c'était une petite forêt. Vous pouvez prendre la pyramide à base carrée et/ou le cône.
- Dessiner une horloge à l'aide de 12 cônes disposés en cercle et de deux pyramides formant des aiguilles, petite et grande, et bougeant en fonction du temps.

Déposer un fichier zip contenant votre dossier du TP2, avec chacun des projets clairement nommé (Pyramide, Cube, Grille, Disque, Cône, Forêt, Horloge).

Important : faites extrêmement attention aux chemins des dossiers libs présents dans l'archive. Vous devez faire en sorte que votre projet fonctionne tel quel, sorti de l'archive, sur une autre machine et système que le vôtre. Le correcteur ne peut pas consacrer 10 minutes à éditer chacun de vos sources pour qu'il fonctionne.