LABORATOIRE 7: WEBGL ET LES OBJETS 3D

Théorie

1. Le **3D** et **WebGL**

À l'intérieur d'un canevas, il est possible de dessiner en 3D. Pour ce faire, il faut utiliser le contexte **WebGL**. Le contexte **WebGL** est supporté par la très grande majorité des navigateurs.

WebGL est une librairie graphique que l'on peut utiliser à l'intérieur d'un canevas. Cette librairie graphique provient d'**OpenGL ES 2.0** mais certaines fonctions ont été modifiées et d'autres ont été ajoutées pour s'adapter au langage **Javascript** et à **HTML5**.

La grande force d'**OpenGL** (et par conséquent de **WebGL**), c'est que plusieurs fonctions sont directement exécutées par le processeur de la carte graphique (**GPU**). Cela fait en sorte que les animations en **OpenGL** sont très rapides.

Depuis 2012, la version de **WebGL 2.0** basée sur **OpenGL ES 3.0** est en construction.

2. Effacer le canevas en WebGL

Ouvrez la page Web **7-1 WebGl effacer.htm**. Cette page Web ne fait qu'effacer le canevas en noir.

Tout d'abord, sur la page Web, il y a un canevas. C'est à l'intérieur de ce canevas que nous allons dessiner.

```
<canvas id="monCanvas" width="640" height="640">
    Votre navigateur ne supporte pas la balise canvas
</canvas>
```

La fonction **démarrer** (appelée au chargement de la page Web), va chercher le canevas, initialise le contexte **WebGL** puis efface le canevas.

```
function demarrer() {
  var objCanvas = document.getElementById('monCanvas');
  var objgl = initWebGL(objCanvas);  // Initialise le contexte WebGL
  effacerCanevas(objgl);
}
```

La fonction qui initialise le contexte est située dans le fichier **WebGL.js**. Nous l'avons placé à part pour éviter d'encombrer la page Web.

```
function initWebGL(objCanvas) {
    var objgl = null;
    try {
        // Essayer de récupérer une des versions de WebGL.
        objgl = objCanvas.getContext("experimental-webgl2") ||
                objCanvas.getContext('webgl') ||
                objCanvas.getContext('experimental-webgl');
        objgl.enable(objgl.DEPTH_TEST); // Active le test de profondeur
        objgl.depthFunc(objgl.LEQUAL);  // Les objets proches cachent les objets lointains
    }
    catch(e) {
      alert('Impossible d\'initialiser le WebGL. Il est possible que votre navigateur ne supporte pas cette fonctionnalité.\n', e.message);
    }

    return objgl;
}
```

Dans cette fonction, tout d'abord, nous récupérons le contexte **WebGL** à partir du canevas (comme nous l'avons fait pour le contexte 2D). Jusqu'à date, il existe deux contextes **WebGL**. Le contexte nommé ***webgl*** est le **WebGL version 1.0**. Le contexte nommé ***experimental-webgl*** est le contexte qui prévalait avant la sortie de **WebGL version 1.0**.

L'instruction ***objgl = objCanvas.getContext('webgl') || objCanvas.getContext('experimental-webgl')*** signifie d'aller chercher le contexte ***webgl*** et, si ce contexte n'existe pas, d'aller chercher le contexte ***experimental-webgl***.

L'instruction ***objgl.enable(objgl.DEPTH_TEST)*** indique d'activer le test de la profondeur (par défaut, il ne l'est pas car il est possible de faire du 2D en **WebGL**). Le test de la profondeur consiste à vérifier si un voxel (un pixel 3D) est placé en arrière d'un autre sur l'axe des Z. L'instruction ***objgl.depthFunc(objgl.LEQUAL)*** signifie que si un voxel est placé en arrière d'un autre sur l'axe des Z, ce voxel n'est pas dessiné. Cela n'est pas important en 2D car il n'y a pas de profondeur mais c'est très important en 3D.

Ici, si une des instructions précédentes échouent, un message d'erreur s'affiche.

```
catch(e) {
  alert('Impossible d\'initialiser le WebGL. Il est possible que votre navigateur ne supporte pas cette fonctionnalité.\n', e.message);
}
```

La fonction **effacerCanevas** efface le canevas.

Tout d'abord, nous mentionnons, à **WebGL**, la couleur qui doit être utilisée pour effacer le canevas : ***objgl.clearColor(0.0, 0.0, 0.0, 1.0).***

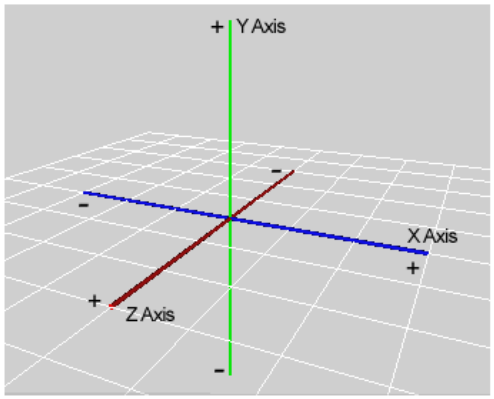
```
function effacerCanevas(objgl) {
    // Mettre la couleur d'effacement au noir et complétement opaque
    objgl.clearColor(0.0, 0.0, 0.0, 1.0);
    // Effacer en profondeur avec la couleur
    objgl.clear(objgl.COLOR_BUFFER_BIT | objgl.DEPTH_BUFFER_BIT);
}
```

Il est important de mentionner qu'une couleur est un quadruplet (**rouge, vert, bleu, alpha**). Chaque nombre du quadruplet doit varier entre 0.0 et 1.0. Par exemple, **(1.0, 1.0, 0.0, 1.0)** c'est du jaune clair opaque. Par exemple, **(0.5, 0.5, 0.0, 0.5)** c'est du jaune plus foncé à moitié opaque. Ici, **(0.0, 0.0, 0.0, 1.0)** représente la couleur noire opaque.

Finalement, nous effaçons le canevas : ***objgl.clear(objgl.COLOR_BUFFER_BIT | objgl.DEPTH_BUFFER_BIT).*** Cette instruction signifie d'effacer le canevas en utilisant la couleur qu'il y a dans ***clearColor*** et de l'effacer en profondeur.

3. Quelques notions reliées au 3D

Avant de poursuivre, il est très important d'expliquer quelques notions reliées au 3D. Cela va grandement aider à comprendre la suite.

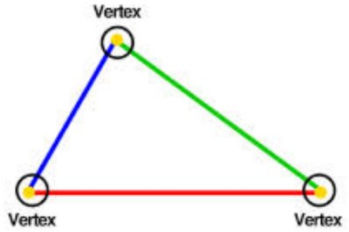


3.1 Le plan cartésien en 3D

En 3D, il y a trois axes sur le plan cartésien : l'axe des X (pour la largeur), l'axe des Y (pour la hauteur) et l'axe des Z (pour la profondeur). En **WebGL**, l'axe des Z positifs est situé vers l'avant.

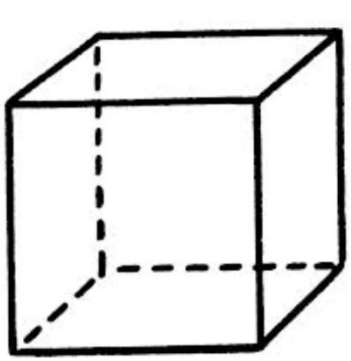
3.2 Les vertex

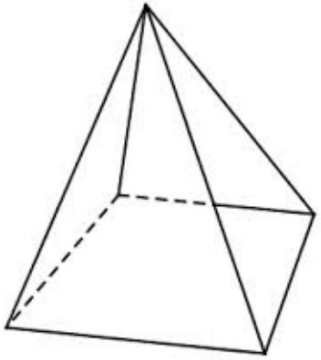
Dans sa forme la plus simple, un vertex est un triplet qui décrit une position 3D. Par exemple, le vertex **(3,-1,5)** décrit la position X=3, Y=-1, et Z=5.

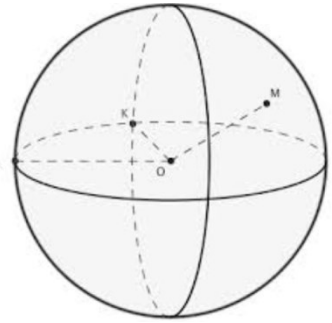


La majorité du temps, on utilise des vertex pour décrire la position des sommets d'une figure géométrique.

Par exemple, un triangle possède 3 sommets (donc 3 vertex), un rectangle possède 4 sommets (donc 4 vertex), un cube possède 8 sommets (donc 8 vertex), une pyramide possède 5 sommets (donc 5 vertex).





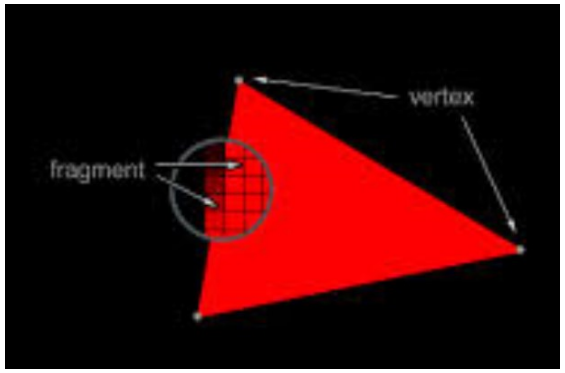


Mais on peut également l'utiliser pour décrire n'importe quelle autre position 3D. Par exemple, le centre d'une sphère.

3.3 Les *shaders*

WebGL, pour dessiner un objet, utilise toujours des ***shaders*** (ici, j'utilise le terme anglais car il n'y a pas de traduction française appropriée). Un ***shader*** est un programme qui s'exécute directement sur la carte graphique. Le but d'un ***shader*** est de « paramétrer et calculer » l'apparence et le rendu d'un objet 3D tel que ses positions 2D, ses couleurs, son éclairage, ses textures, etc.

En **WebGL**, il existe deux grandes catégories de ***shaders*** : le ***vertex shader*** et le ***fragment shader*** (aussi nommé ***pixel shader***).



Le ***vertex shader*** agit au niveau du vertex. Par exemple, supposons qu'on veut faire tourner un triangle. C'est le ***vertex shader*** qui va s'occuper de calculer les positions 2D du triangle. Il y a un appel au ***vertex shader*** pour chacun des vertex de l'objet. Par exemple, un triangle possède 3 vertex. Par conséquent, il y a 3 appels au ***vertex shader***.

Le ***fragment shader*** agit au niveau des pixels de l'objet. Par exemple, supposons qu'on colore un triangle. C'est le ***fragment shader*** qui va s'occuper de colorer chacun des pixels du triangle.

Par conséquent, la séquence est la suivante :

- Le ***vertex shader*** s'occupe de calculer la position 2D de chaque vertex de l'objet 3D et, possiblement, sa normale.
- Le ***fragment shader*** s'occupe du reste, c'est-à-dire de colorer chaque pixel de l'objet en fonction de ses couleurs, de son éclairage, de ses textures, etc.

3.4 Les matrices de transformation 3D

En **WebGL**, pour déplacer, pour faire tourner ou pour modifier la taille d'un objet, on utilise toujours une matrice de transformation. C'est à l'aide de cette matrice que les nouvelles positions d'un objet 3D sont calculées à la suite de sa transformation.

Nous avons déjà vu l'utilisation des matrices de transformation en 2D pour transformer le contexte 2D.

En 2D, la matrice de transformation est une matrice 3X3 qui a cette forme :

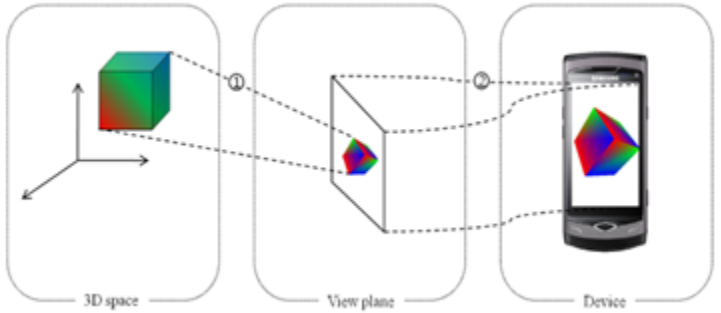
$$\begin{matrix} \mathbf{X_{11}} & \mathbf{X_{12}} & \mathbf{X_{13}} \\ \mathbf{X_{21}} & \mathbf{X_{22}} & \mathbf{X_{23}} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} \end{matrix}$$

En 3D, la matrice de transformation est une matrice 4X4 qui a cette forme :

$$\begin{matrix} X_{11} & X_{12} & X_{13} & X_{14} \\ X_{21} & X_{22} & X_{23} & X_{24} \\ X_{31} & X_{32} & X_{33} & X_{34} \\ 0 & 0 & 0 & 1 \end{matrix}$$

En **WebGL**, cette matrice de transformation porte le nom de **matrice du modèle** (***modelview matrix***).

En ***WebGL***, il existe une autre matrice de transformation qui porte le nom de **matrice de projection** (***projection matrix***). C'est également une matrice 4X4. Cette matrice sert à transformer les positions 3D en positions 2D car, au final, l'objet 3D va être dessiné sur une surface plane (en 2D). Le 3D n'est qu'une illusion.



En gros, en 3D, la séquence pour dessiner un objet 3D est la suivante :

- On définit l'objet 3D ainsi que le type de dessin. C'est le programmeur qui doit s'en occuper.
- On fait passer cet objet à travers la **matrice du modèle** (par exemple, lorsqu'on veut déplacer l'objet). Cela va définir des nouvelles positions 3D pour cet objet. C'est le rôle du ***vertex shader***.
- On fait passer cet objet transformé à travers la **matrice de projection**. Cela va définir les positions 2D correspondantes pour cet objet. C'est le rôle du ***vertex shader.***
- On colore chacun des pixels de l'objet. C'est le rôle du ***fragment shader***.
- On dessine chacun des pixels de l'objet. C'est la carte graphique qui s'en occupe.

4. Le dessin d'un carré blanc

Ouvrez la page Web **7-2-1 WebGL Dessiner carré blanc.htm**. Cette page Web efface le canevas en noir et dessine un carré blanc.

En 2D, cela ne demanderait que quelques lignes de code.

Il en va autrement en **WebGL**. En **WebGL**, cela demande plusieurs lignes de code. Par contre, ces lignes de code vont être réutilisées plus tard pour dessiner des objets beaucoup plus complexes qu'un simple carré blanc.

Voyons voir à quoi ressemble la fonction **demarrer** (qui s'exécute au chargement de la page Web).

```
function demarrer() {
  var objCanvas = document.getElementById('monCanvas');
  var objgl = initWebGL(objCanvas);  // Initialise le contexte WebGL
  var objProgShaders = initShaders(objgl); // iniatilise les sahaders
  var objCarre = creerCarre(objgl); // Définir le carré
  effacerCanevas(objgl);
  dessiner(objgl, objProgShaders, objCarre); // Le dessiner
}
```

Les fonctions **initWebGL** et **effacerCanevas** ont déjà été vus. Voyons voir à quoi ressemble la fonction **initShaders** qui sert à initialiser les ***shaders***. Cette fonction a été programmée dans **ShaderBlanc.js**.

4.1 La programmation des *shaders*

Tel que déjà énoncé, un ***shader*** est un programme qui est exécuté par le processeur de la carte graphique.

On ne peut pas programmer des ***shaders*** en **Javascript**. La raison principale, c'est que le langage **Javascript** n'a pas une syntaxe assez développée pour que ce soit possible. Il aurait été pensable d'ajouter une librairie de fonctions qui aurait donné cette possibilité. Mais cela aurait beaucoup ralenti la performance d'exécution (le 3D requiert habituellement beaucoup de performance).

En **WebGL**, nous devons programmer les **shaders** dans un langage de programmation nommé **GLSL (*OpenGL Shading Language).*** Les programmes écrits dans ce langage doivent être compilés. Une fois compilés, ces programmes sont directement exécutés par le processeur de la carte graphique (**GPU**) et non pas par le processeur de l'ordinateur (**CPU**).

C'est pour cette raison que les animations 3D en **WebGL** sont très performantes. C'est le processeur de la carte graphique qui s'occupe d'exécuter les tâches qui demandent le plus de temps à s'exécuter.

Voyons à quoi ressemble la programmation d'un ***vertex shader*** en **GLSL**.

```
attribute vec3 vertexPos;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main(void) {
  gl_Position = projectionMatrix * modelViewVMatrix * vec4(vertexPos, 1.0);
}
```

Tel que déjà mentionné, un des rôles du ***vertex shader*** est de calculer la position de chaque vertex par rapport à l'objet 3D. Ici, la variable nommée **vertexPos** contient le vertex, la variable nommée **modelViewMatrix** contient la matrice 4X4 du **modèle** et la variable nommée **projectionMatrix** contient la matrice 4X4 de **projection.**

Ce programme multiplie ces trois variables et mets le résultat dans la variable **gl_Position**. Après calcul, cette variable contient la position **2D** du vertex. La carte graphique va utiliser ce résultat pour « dessiner » l'objet.

Il est inutile de s'attarder là-dessus. Tout ce que vous devez comprendre, c'est que ce ***shader*** calcule la position **2D** d'un vertex après transformations.

Voyons à quoi ressemble la programmation du ***fragment shader*** en **GLSL**.

```
void main(void) {
  gl_FragColor = vec4(1.0, 1.0, 1,0, 1.0);
}
```

Tel que déjà mentionné, un des rôles du ***fragment shader*** est de colorer chacun des pixels de l'objet. La couleur est placée dans la variable **gl_FragColor**. Ici, tous les objets dessinés vont être blancs (c'est la raison pour laquelle le carré est blanc).

Nous allons ajouter d'autres couleurs plus tard dans un autre programme. Pour l'instant, nous allons nous contenter de la couleur blanche.

Maintenant passons à la compilation de ces ***shaders***. C'est ce que fait la fonction **creerShader**.

Tout d'abord, nous vérifions de quel type est le ***shader*** qu'il faut compiler (dans **strType**). S'il s'agit d'un ***shader*** de type '**fragment**', nous créons un ***shader*** de ce type sinon, s'il s'agit d'un ***shader*** de type '**vertex**', nous créons un ***shader*** de ce type.

Par la suite, nous vérifions si le ***shader*** a bien été créé. Si ce n'est pas le cas, nous affichons un message d'erreur sinon nous attachons, à ce ***shader***, le code source (**strSource**) puis nous compilons ce ***shader***. Si la compilation est un échec (probablement dû à des erreurs de syntaxe), nous affichons un message d'erreur.

```
function creerShader(objgl, strSource, strType) {
    var objShader = null;

    if (strType == 'fragment') {
        objShader = objgl.createShader(objgl.FRAGMENT_SHADER);
    } else if (strType == 'vertex') {
        objShader = objgl.createShader(objgl.VERTEX_SHADER);
    }

    if (!objShader) {
        alert('Impossible de créer le ' + strType + 'shader');
    }
    else {
        objgl.shaderSource(objShader, strSource);
        objgl.compileShader(objShader);
        if (!objgl.getShaderParameter(objShader, objgl.COMPILE_STATUS)) {
            alert('Impossible de compiler le ' + strType + ' shader');
        }
    }
    return objShader;
}
```

Finalement, nous retournons le ***shader.*** A cette étape-ci, le ***shader*** est compilé et exécutable.

Ce qu'il reste à faire, c'est d'intégrer ces ***shaders*** compilés à notre **Javascript**. C'est ce que fait la fonction **initShaders**.

Tout d'abord, nous créons les ***shaders*** (et nous les compilons).

```
function initShaders(objgl) {
    var objProgShaders = null;

    // Créer les shaders à partir du code source
    var objFragmentShader = creerShader(objgl, strFragmentShaderSource, 'fragment');
    var objVertexShader = creerShader(objgl, strVertexShaderSource, 'vertex');
```

Observez que les codes sources des **shaders** écrits en **GLSL** ont été placés dans des chaînes de caractères.

```
var strVertexShaderSource =
      '    attribute vec3 vertexPos;\n' +
      '    uniform mat4 modelViewMatrix;\n' +
      '    uniform mat4 projectionMatrix;\n' +
      '    void main(void) {\n' +
      '        gl_Position = projectionMatrix * modelViewMatrix * \n' +
      '            vec4(vertexPos, 1.0);\n' +
      '    }\n';

 var strFragmentShaderSource =
      '    void main(void) {\n' +
      '    gl_FragColor = vec4(1.0, 1.0, 1.0, 1.0);\n' +
      '}\n';
```

Nous aurions pu placer ces codes sources dans des fichiers et lire ces fichiers (cela est très compliqué en ***Javascript***). Nous aurions également pu placer ces codes sources dans des zones de texte cachés sur la page Web. Étant donné que les codes sources ne sont pas longs, nous avons simplement choisi de les placer dans des chaînes de caractères. C'est la solution la plus simple.

À la suite de la création des ***shaders*** compilés, il faut les placer à l'intérieur d'un programme. Le fait de les placer à l'intérieur d'un programme va jouer un double-rôle. D'une part, ce programme va transmettre ces ***shaders*** à la carte graphique et, d'autre part, ce programme va nous permettre d'avoir accès aux variables qui ont été déclarées dans les ***shaders*** et ce, à partir du **Javascript**.

```
// Créer le programme dans lequel seront ajoutés les shaders
objProgShaders = objgl.createProgram();
objgl.attachShader(objProgShaders, objVertexShader);
objgl.attachShader(objProgShaders, objFragmentShader);
objgl.linkProgram(objProgShaders);

if (!objgl.getProgramParameter(objProgShaders, objgl.LINK_STATUS)) {
    alert('Impossible de lier les shaders');
```

Le code suivant crée un programme (**objProgShaders**) et ajoute les ***shaders*** dans ce programme. Il envoie un message d'erreur si cela ne fonctionne pas.

Il ne nous reste plus qu'à nous donner la possibilité d'avoir accès aux variables qui ont été déclarées dans les ***shaders*** et ce, à partir du **Javascript**.

```
else {
    // Lier les variables des shaders à nos propres variables
    objProgShaders.posVertex = objgl.getAttribLocation(objProgShaders, 'vertexPos');
    objgl.enableVertexAttribArray(objProgShaders.posVertex);

    objProgShaders.matProjection = objgl.getUniformLocation(objProgShaders, 'projectionMatrix');
    objProgShaders.matModeleVue = objgl.getUniformLocation(objProgShaders, 'modelViewMatrix');

    objgl.useProgram(objProgShaders);
}
```

Observez l'instruction : ***objProgShaders.posVertex = objgl.getAttribLocation(objProgShaders, 'vertexPos').*** Cela signifie de lier **objProgShaders.posVertex** à la variable **vertexPos** qui a été déclaré dans le ***shader***.

```
attribute vec3 vertexPos;

uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main(void) {
  gl_Position = projectionMatrix * modelViewVMatrix * vec4(vertexPos, 1.0);
}
```

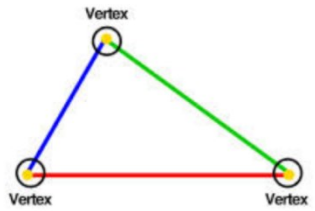
Il en va de même pour les autres variables.
objProgShaders.matProjection = objgl.getUniformLocation(objProgShaders, 'projectionMatrix');
objProgShaders.matModeleVue = objgl.getUniformLocation(objProgShaders, 'modelViewMatrix');

Ici, **.posVertex**, **.matProjection** et **.matModeleVue** sont des noms de propriétés que nous avons inventés. Nous aurions pu inventer n'importe quel autre nom.

La seule chose qu'il faut retenir, c'est que **.posVertex** fait référence à la position du vertex déclarée dans le ***shader***, que **.matProjection** fait référence à la matrice de **projection** déclarée dans le ***shader*** et que **.matModeleVue** fait référence à la matrice du **modèle** déclarée dans le ***shader***.

Observez l'instruction suivante : ***objgl.enableVertexAttribArray(objProgShaders.posVertex);***
Cela signifie que les vertex peuvent être transmis à l'intérieur d'un tableau de vertex.

Cela va simplifier beaucoup notre tâche lorsqu'on objet va contenir plusieurs vertex. Par exemple, un triangle contient 3 vertex. Au lieu de transmettre chacun des vertex un après l'autre au ***vertex shader***, il va suffire de placer ces trois (3) vertex à l'intérieur d'un tableau et de les transmettre en un seul coup.



L'avant-dernière instruction de la fonction **initShaders** est l'instruction qui indique à **WebGL** d'activer le programme dans lequel ont été placés les ***shaders***. Sans cette instruction, les ***shaders*** ne sont pas utilisables.

```
    objgl.useProgram(objProgShaders);
}
```

A la fin, ce programme est retourné.

```
        return objProgShaders;
    }
```

En **WebGL**, la mise en place des ***shaders*** est un processus long, pénible et aride. Par contre, ce processus n'est exécuté qu'une seule fois. Par la suite, s'il y a des changements à faire au niveau des ***shaders***, ceux-ci vont être mineurs.

4.2 La création du carré

Maintenant, nous sommes prêts à créer notre premier objet 3D: un simple carré.

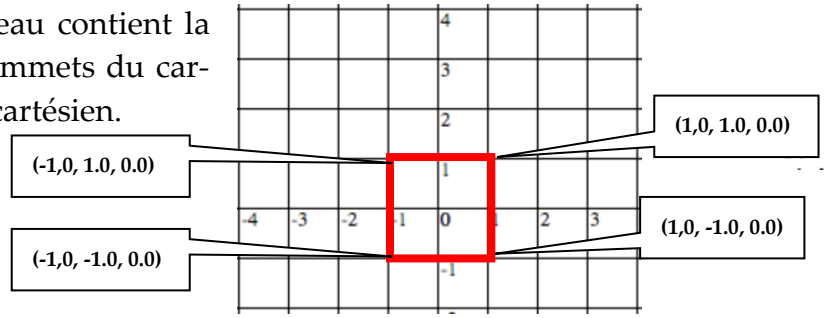
En **WebGL**, tous les objets 3D doivent être placés dans un tampon (un tampon est un bloc de données contiguës). L'avantage de placer les objets 3D dans un tampon est que cela permet de transmettre rapidement des données. Par exemple, ici, lorsque le carré va être transmis au ***vertex shader***, c'est le tampon au complet qui va être transmis. Cela va se faire en un seul coup. L'instruction ***var objCarre = objgl.createBuffer()*** sert à créer un nouveau tampon et à l'affecter à la variable **objCarre**.

```
//Pour créer le carré
function creerCarre(objgl) {
    var objCarre = objgl.createBuffer();
    var tabVertex = [
         1.0, 1.0, 0.0,
         1.0, -1.0, 0.0,
        -1.0, -1.0, 0.0,
        -1.0, 1.0, 0.0
    ];
```

L'instruction suivante sert à définir les vertex du carré. Un carré possède 4 vertex. Comme il est possible de constater ces vertex ont été placés dans un tableau.

```
//Pour créer le carré
function creerCarre(objgl) {
    var objCarre = objgl.createBuffer();
    var tabVertex = [
         1.0, 1.0, 0.0,
         1.0, -1.0, 0.0,
        -1.0, -1.0, 0.0,
        -1.0, 1.0, 0.0
```

En fait, ce tableau contient la position des sommets du carré sur un plan cartésien.



Observez que toutes les coordonnées en Z sont égales à 0. Vous devez visualiser le carré comme un objet 3D qui n'a aucune profondeur (comme une feuille de papier ultra mince).

L'endroit où l'on place nos objets 3D sur le plan cartésien n'a pas réellement d'importance. Dans le prochain laboratoire, nous allons transformer les objets que nous allons avoir créés. C'est grâce à ces transformations que nous allons déplacer ces objets 3D, les tourner et/ou modifier leur taille.

Bien que l'endroit où l'on place nos objets 3D sur le plan cartésien n'ait pas d'importance, il est habituel de les centrer autour du point d'origine du plan cartésien. Cela facilite beaucoup les transformations par la suite.

Par la suite, on sélectionne ce tampon (il faut toujours sélectionner un tampon avant de l'utiliser) puis on place le tableau **tabVertex** dans ce tampon; **objgl.ARRAY_BUFFER** indique que, dans ce tampon, il y a un tableau; **new Float32Array(tabVertex)** crée un tableau de type **float (32 bits)** à partir de **tabVertex** et insère ce tableau dans le tampon; **objgl.STATIC_DRAW** signifie que ce tableau contient un objet qui sera dessiné et que les vertex de cet objet ne seront pas modifiés. L'avantage d'utiliser un objet statique (par rapport à un objet dynamique), c'est que les vertex de cet objet sont conversés tel quel en mémoire et ne sont pas détruits par la suite. Cela optimise « le dessin » de l'objet si l'objet se dessine plusieurs fois en cours d'exécution.

```
objgl.bindBuffer(objgl.ARRAY_BUFFER, objCarre);
objgl.bufferData(objgl.ARRAY_BUFFER, new Float32Array(tabVertex), objgl.STATIC_DRAW);
```

Toutes ces informations semblent anodines mais il ne faut pas oublier que ce tampon est transmis au ***vertex shader*** et ce dernier utilise ces informations.

La dernière information est une information que nous allons utiliser plus tard lorsque nous allons dessiner le carré. Elle indique le type de dessin. Ici, le type de dessin est ***LINE_LOOP***.

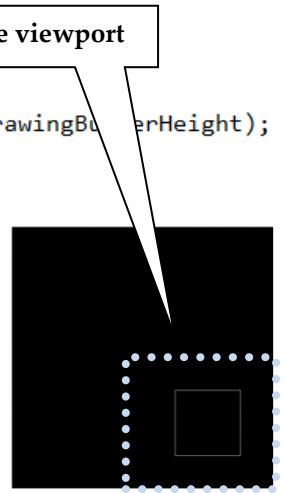
```
objCarre.typeDessin = objgl.LINE_LOOP;

return objCarre;
```

4.3 Le dessin du carré

Il ne nous reste plus qu'à dessiner le carré.

```
// Pour dessiner le carré
function dessiner(objgl, objProgShaders, objCarre) {
    // La vue sur le canevas
    objgl.viewport(0, 0, objgl.drawingBufferWidth, objgl.drawingBufferHeight);
```

Tout d'abord, nous devons décider dans quelle section du canevas (le *viewport*) nous allons dessiner. Ici, nous utilisons toute la zone dessinable du canevas ***(objgl.drawingBufferWidth, objgl.drawingBufferHeight).***

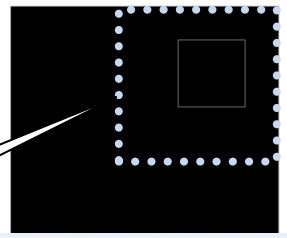


Mais nous pourrions utiliser une section plus petite du canevas pour dessiner. Par exemple, l'instruction suivante utilise la section inférieure droite du canevas pour dessiner.

objgl.viewport(objgl.drawingBufferWidth/2, 0,
objgl.drawingBufferWidth/2, objgl.drawingBufferHeight/2);

Par exemple, l'instruction suivante utilise la section supérieure droite pour dessiner.

objgl.viewport(objgl.drawingBufferWidth/2, objgl.drawingBufferHeight/2,
objgl.drawingBufferWidth/2, objgl.drawingBufferHeight/2);



Le fait de pouvoir choisir dans quelle section du canevas nous allons dessiner nous donne la possibilité de dessiner à l'intérieur d'un médaillon comme cela existe dans certains jeux 3D.

Dans cette image, le dessin des tuyaux est situé à l'intérieur d'un médaillon (à l'intérieur d'un *viewport*).

Par la suite, nous créons la matrice de **projection** et nous transmettons cette matrice au ***vertex shader***.

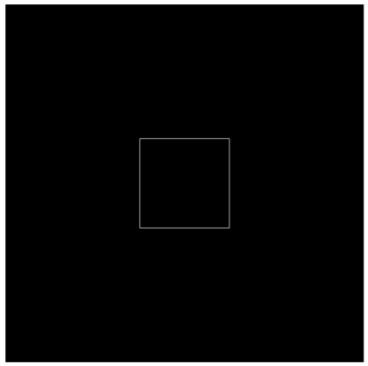
```
// Matrice de projection
var matProjection = mat4.create();
mat4.ortho(-2, 2, -2, 2, -5, 5, matProjection);
// Relier la matrice de projection aux shaders
objgl.uniformMatrix4fv(objProgShaders.matProjection, false, matProjection);
```

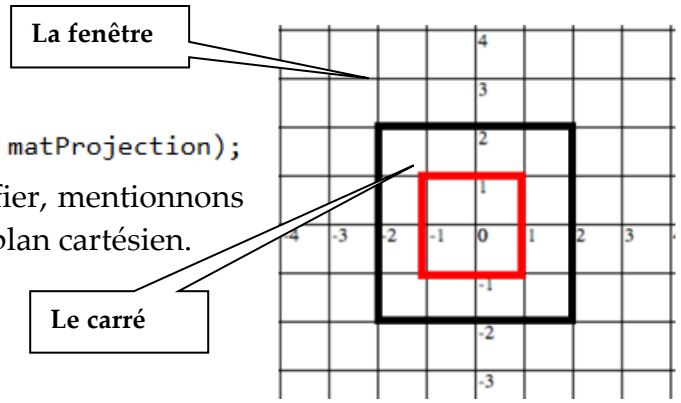
La méthode ***.ortho*** est une projection orthographique. Pour simplifier, mentionnons que la projection orthographique est comme une fenêtre 3D sur le plan cartésien.

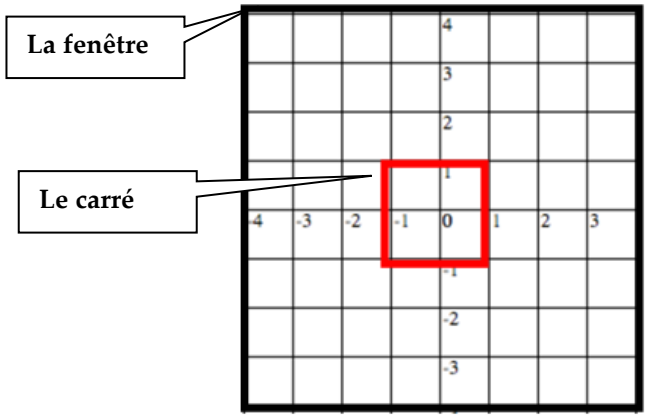
Si nous agrandissons la fenêtre 3D, le carré va paraître plus petit.

Par exemple, voici une autre projection orthographique.

mat4.ortho(-4, 4, -4, 4, -5, 5, matProjection);

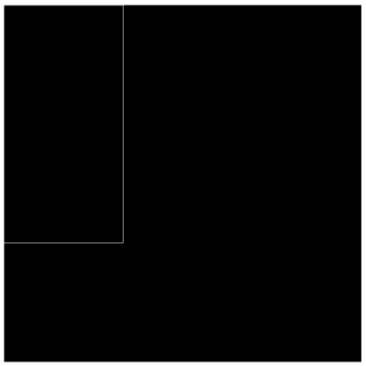


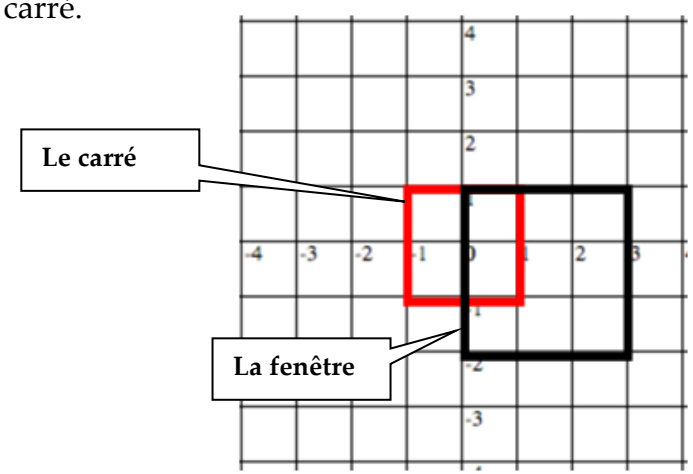




Il est possible que la projection orthographique cache une partie du carré.

mat4.ortho(0, 3, -2, 1, -5, 5, matProjection);





Observez, dans la projection orthographique, que, sur l'axe des Z, la fenêtre 3D va de -5 à 5. En fait, ici, cela n'a pas d'importance pour l'instant car le triangle n'a aucune profondeur. Plus tard, cela va avoir de l'importance.

Il est à noter que l'objet **mat4** (qui représente une matrice 4X4) est situé dans la librairie **glMatrix-0.9.5.min.js.** Cette librairie contient plusieurs opérations sur des vecteurs et sur des matrices. Cette librairie est facilement trouvable sur Internet car elle est très utilisée. Pour avoir de l'aide sur cette librairie, consultez http://glmatrix.net/docs/2.2.0/symbols/mat4.html.

L'instruction suivante est très importante. Elle sert à affecter la matrice de projection à la variable **projectionMatrix** qui a été déclarée dans le ***vertex shader*** :
objgl.uniformMatrix4fv(objProgShaders.matProjection, false, matProjection).

N'oubliez pas, lorsque nous avons initialisé les ***shaders***, que nous avons relié **objProgShaders.matProjection** à la variable **projectionMatrix** déclaré dans le ***vertex shader***.
objProgShaders.matProjection = objgl.getUniformLocation(objProgShaders, 'projectionMatrix');

Écrire cette instruction, c'est comme écrire : ***projectionMatrix = matProjection*** directement dans le ***vertex shader***. Mais on ne peut pas faire cela car le ***vertex shader*** n'est pas été programmé en **Javascript**.

Par la suite, nous créons la matrice du **modèle** et nous transmettons cette matrice au ***vertex shader***.

```
// Matrice du modèle
var matModeleVue = mat4.create();
mat4.identity(matModeleVue);
// Relier la matrice du modèle aux shaders
objgl.uniformMatrix4fv(objProgShaders.matModeleVue, false, matModeleVue);
```

Ici, la matrice du modèle que nous utilisons est la matrice identité. La particularité de la matrice identité, c'est qu'elle ne transforme rien. C'est un élément neutre (comme le 0 dans l'addition ou le 1 dans la multiplication). Par conséquent, les 4 vertex du carré sont dessinés exactement aux mêmes positions dans le plan cartésien que nous les avons définis.

L'instruction suivante est très importante. Elle sert à affecter la matrice du modèle à la variable **modelViewMatrix** qui a été déclarée dans le ***vertex shader***.
objgl.uniformMatrix4fv(objProgShaders.matModeleVue, false, matModeleVue).

N'oubliez pas que, lorsque nous avons initialisé les ***shaders***, nous avons relié **objProgShaders.matModeleVue** à la variable **modelViewMatrix** déclaré dans le ***vertex shader***.
objProgShaders.matModeleVue = objgl.getUniformLocation(objProgShaders, 'modelViewMatrix');

Écrire cette instruction, c'est comme écrire : ***modeleViewMatrix = matModeleVue*** directement dans le ***vertex shader***.

La dernière chose que nous devons transmettre au ***vertex shader***, ce sont les 4 vertex du carré. C'est ce que font les deux instructions suivantes :

```
// Relier les vertex aux shaders
objgl.bindBuffer(objgl.ARRAY_BUFFER, objCarre);
objgl.vertexAttribPointer(objProgShaders.posVertex, 3, objgl.FLOAT, false, 0, 0);
```

Ici, nous sélectionnons le tampon qui contient les 4 vertex du carré puis nous passons par ***objProgShaders.posVortex*** pour affecter chacun des vertex à la variable **vertexPos** qui a été déclarée dans le ***vertex shader***.

Ici, **3** indique la taille de chaque élément à l'intérieur d'un vertex et **objgl.FLOAT** indique que chaque élément est de type **float**. Les trois derniers paramètres n'ont pas d'importance pour le moment.

Maintenant que le ***vertex shader*** possède toutes les informations nécessaires, il ne reste plus qu'à dessiner le carré.

```
// Dessiner le carré
var intNbVertex = (objgl.getBufferParameter(objgl.ARRAY_BUFFER, objgl.BUFFER_SIZE) / 4) / 3;
objgl.drawArrays(objCarre.typeDessin, 0, intNbVertex);
```

L'instruction ***intNbVertex = (objgl.getBufferParameter(objgl.ARRAY_BUFFER, objgl.BUFFER_SIZE) / 4 / 3)*** calcule le nombre de vertex qu'il y a dans le tampon. Ici, il y en a quatre (4).

L'instruction ***objgl.drawArrays(objCarre.typeDessin, 0, intNbVertex)*** signifie « dessine ***intNbVertex*** (4) vertex et ce, à partir du vertex #0 en utilisant le type de dessin mentionné ».

Ici le type de dessin est ***LINE_LOOP***. Cela signifie que le vertex #0 va être relié au vertex #1 par une droite, que le vertex #1 va être relié au vertex #2 par une droite, que le vertex #2 va être relié au vertex #3 par une droite et que le vertex #3 va être relié au vertex #0 par une droite. C'est de cette manière que nous dessinons le carré.

Cette manière de procéder pour dessiner peut sembler aride à première vue mais le premier but de **WebGL**, c'est la performance et non pas la simplicité.

Pour que ce soit réellement performant, il faut que le cœur du programme soit exécuté par le processeur de la carte graphique (GPU) et non pas par le processeur de l'ordinateur (CPU). C'est une des raisons pour laquelle les *shaders* sont programmés en **GLSL** et non pas en **Javascript**. Toute la complexité du 3D en **WebGL** s'explique par l'existence des *shaders* (par exemple, l'obligation de relier les variables **Javascript** aux variables qui ont été déclarées dans les *shaders*).

Par contre, le concept est toujours le même pour tous les programmes 3D. Une fois qu'on a bien compris le concept on s'habitue.

5. D'autres dessins

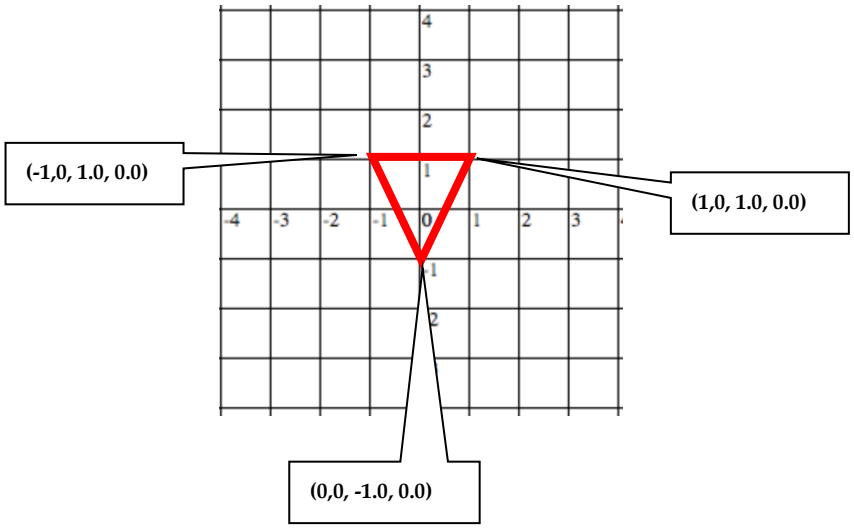
Dans cette section, nous allons dessiner d'autres objets.

5.1 Le dessin d'un triangle blanc

Ouvrez la page Web **7-2-2 WebGl Dessiner triangle blanc.htm**. Cette page Web efface le canevas en noir et dessine un triangle blanc.

Il n'y a pas grand-chose à modifier pour dessiner un triangle. En fait, la seule chose que l'on doit modifier, ce sont les vertex dans la création du triangle…

```
var objTriangle = objgl.createBuffer();
var tabVertex = [
      1.0, 1.0, 0.0,
     -1.0, 1.0, 0.0,
      0.0, -1.0, 0.0
 ];
```



Le reste du programme est identique (à part du nom du tampon).

5.2 Le dessin d'un triangle plein blanc

Ouvrez la page Web **7-2-3 WebGL Dessiner triangle plein blanc.htm**. Cette page Web efface le canevas en noir et dessine un triangle plein blanc.

Il n'y a pas grand-chose à modifier pour dessiner un triangle plein par rapport au triangle vide. En fait, la seule chose que l'on doit modifier, c'est le type de dessin. On doit indiquer que le triangle est plein dans le dessin.

```
objTriangle.typeDessin = objgl.TRIANGLES;
```

Ici, on utilise ***TRIANGLES*** au lieu de ***LINE_LOOP***.

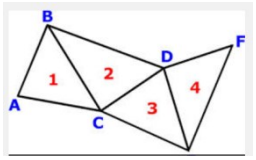
5.3 Le dessin d'un carré plein blanc

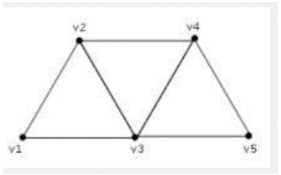
Ouvrez la page Web **7-2-4 WebGL Dessiner carré plein blanc.htm**. Cette page Web efface le canevas en noir et dessine un carré plein blanc.

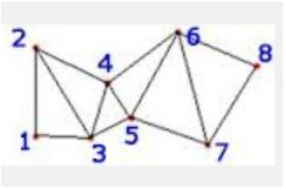
En **WebGL**, une manière de dessiner des figures géométriques pleines est de juxtaposer des triangles pleins en utilisant **TRIANGLE_STRIP**.

```
objCarre.typeDessin = objgl.TRIANGLE_STRIP;
```

L'utilisation de la juxtaposition de triangles pleins nous donne une variété de figures géométriques pleines comme l'illustre les images suivantes.





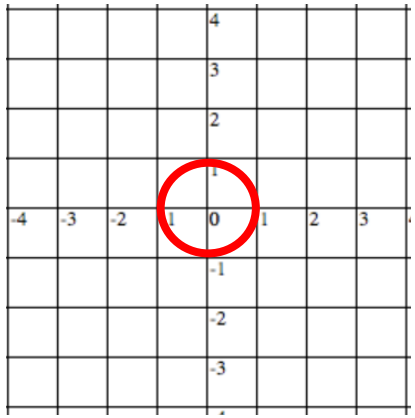


Il faut faire attention à l'ordre des vertex lorsqu'on juxtapose des triangles pleins car la dernière droite du triangle sert de base pour le prochain triangle plein.

5.4 Le dessin d'un cercle blanc

Ouvrez la page Web **7-2-5 WebGL Dessiner cercle blanc.htm**. Cette page Web efface le canevas en noir et dessine un cercle blanc.

En **WebGL**, il n'existe pas de manière directe pour dessiner une courbe. La seule manière de dessiner une courbe, c'est de créer plusieurs vertex rapprochés entre eux et de les joindre par des droites.



Ici, pour dessiner le cercle, nous avons utilisé les équations trigonométriques du cercle (voir laboratoire 9).

$$x = x_c + cos(\alpha) * r$$
$$y = y_c + sin(\alpha) * r$$

Dans notre dessin, le rayon du cercle dessiné est de 1 et le centre est (0,0). Par conséquent, la formule est la suivante.

$$x = cos(\alpha)$$
$$y = sin(\alpha)$$

Dans la création du cercle, nous créons un tableau de 360 vertex : (**cos(α), sin(α), 0.0**) où **α** est l'angle en radians.

```
function creerCercle(objgl) {
 var objCercle = objgl.createBuffer();
 var tabVertex = [];
   for (var i = 0; i < 360; i++) {
     tabVertex = tabVertex.concat([Math.cos(i*Math.PI/180),Math.sin(i*Math.PI/180),0.0]);
 }
 objgl.bindBuffer(objgl.ARRAY_BUFFER, objCercle);
 objgl.bufferData(objgl.ARRAY_BUFFER, new Float32Array(tabVertex), objgl.STATIC_DRAW);

 // Rejoindre par des droites
 objCercle.typeDessin = objgl.LINE_LOOP;

 return objCercle;
}
```

Ces vertex sont rejoints par des droites.

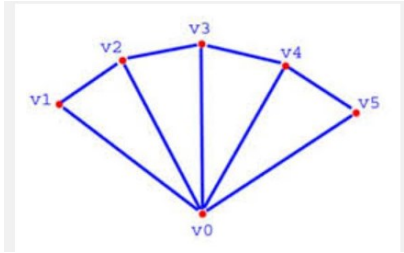
5.5 Le dessin d'un cercle plein blanc

Ouvrez la page Web **7-2-6 WebGL Dessiner cercle plein blanc.htm**. Cette page Web efface le canevas en noir et dessine un cercle plein blanc.

Pour dessiner un cercle plein, nous utilisons des triangles pleins en forme d'éventail (***TRIANGLE_FAN***).

```
objCercle.typeDessin = objgl.TRIANGLE_FAN;
```

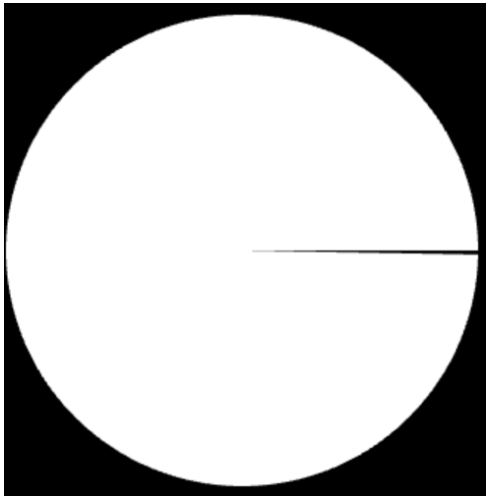
Dans les triangles pleins en forme d'éventail, nous avons un vertex de départ (**v0**). Tous les autres vertex sont reliés à ce vertex de départ par des triangles pleins.



Ici, le vertex de départ est le centre du cercle.

```
var tabVertex = [0,0,0] // Le centre du cercle;
  for (var i = 0; i <= 360; i++) {
    tabVertex = tabVertex.concat([Math.cos(i*Math.PI/180),Math.sin(i*Math.PI/180),0.0]);
}
```

Attention : Pour que l'éventail soit bien dessiné, il faut inclure le 360ième degré sinon il va manquer une petite pointe (voir illustration). Par conséquent, il y a 362 vertex en tout dans le dessin du cercle plein.



Prenez note que nous aurions pu utiliser des triangles pleins en forme d'éventail au lieu d'utiliser des triangles pleins juxtaposés pour dessiner le carré plein.

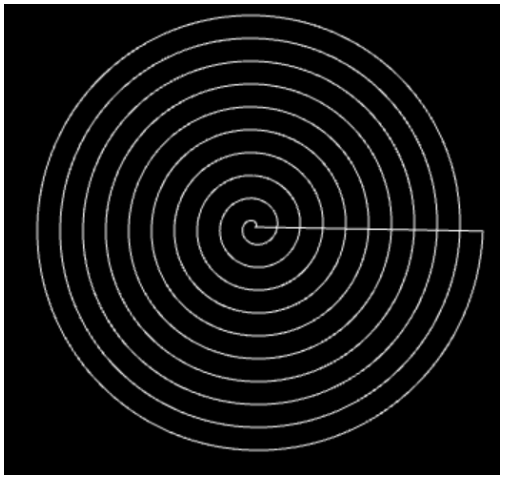
5.6 Le dessin d'une spirale blanche

Ouvrez la page Web **7-2-7 WebGL Dessiner spirale blanche.htm**. Cette page Web efface le canevas en noir et dessine une spirale blanche.

Pour créer une spirale, nous procédons de la même manière que pour créer un cercle sauf que nous augmentons le rayon de manière progressive. Au début, le rayon est 0; à la fin, le rayon est 1. Cela a déjà été expliqué dans le laboratoire 9.

```
function creerSpirale(objgl) {
   var objSpirale = objgl.createBuffer();
   var intNbCirconvolutions = 10;
   var tabVertex = [];
   for (var i = 0; i < 360 * intNbCirconvolutions; i++) {
       var fltRayon = i / (360 * intNbCirconvolutions);
       tabVertex = tabVertex.concat([Math.cos(i * Math.PI / 180) * fltRayon, Math.sin(i * Math.PI / 180) * fltRayon, 0.0]);
   }
    objgl.bindBuffer(objgl.ARRAY_BUFFER, objSpirale);
    objgl.bufferData(objgl.ARRAY_BUFFER, new Float32Array(tabVertex), objgl.STATIC_DRAW);

    objSpirale.typeDessin = objgl.LINE_STRIP;

    return objSpirale;
}
```

Observez ici, qu'il y a 3600 vertex (**360 * intNbCirconvolutions**). Cela signifie que, pour dessiner la spirale, le ***vertex shader*** s'exécute 3600 fois. Cela n'aurait pas été très performant si ce ***shader*** avait été programmé en **Javascript**.



Observez également que, pour dessiner la spirale, nous utilisons ***LINE_STRIP*** et non pas ***LINE_LOOP***. ***LINE_STRIP*** ne rejoint pas le dernier vertex au premier.

Si nous avions utilisé ***LINE_LOOP***, cela aurait dessiné une spirale qui ressemble à cela.