Apprendre la programmation graphique en 3D avec WebGL

Une carte graphique c'est quoi ? À quoi ça sert ?

Afin d'être sur que tout le monde parle le même langage nous allons commencer par revoir la définition de carte graphique. Qu'est-ce qu'une carte graphique et à quoi cela sert-il ?

Une carte graphique est un composant matériel dont le rôle est de convertir des données numériques brutes en données pouvant être affichées sur un terminal d'affichage tel qu'un écran par exemple. Son rôle ne se limite cependant pas à ça puisqu'elle décharge de plus en plus le processeur central des calculs complexes 3D et ce au moyen de diverses techniques que nous allons voir plus bas.

Une carte graphique est composée de quatre éléments principaux :

Le GPU (Graphical processing unit) :

C'est le processeur central de la carte graphique. Aujourd'hui les GPU possèdent des fonctions très avancées. Chaque génération de GPU apporte son lot d'innovations technologiques, qui sont plus ou moins utilisées dans les jeux. Le principal intérêt est de soulager le processeur central, d'augmenter la qualité des images tout en faisant chuter le moins possible les performances. Voici à quoi peut ressembler un GPU sans son ventirad (le ventirad est le couple radiateur + ventilateur) :

Le RAMDAC (Random access memory digital analog converter) :

cette puce est destinée à convertir les données numériques en données analogiques compréhensibles par les écrans cathodiques. La fréquence du RAMDAC nous renseigne sur le nombre maximal d'images par seconde que la carte peut afficher au maximum (même si sa puissance théorique est supérieure, elle sera limitée par le RAMDAC si celui-ci est cadencé trop faiblement). Rassurez-vous, les cartes ne sont jamais bridées par ce composant. On parle souvent de fréquence de rafraîchissement. La fréquence de rafraîchissement est mesurée en Hz (hertz) et va déterminer le nombre maximal d'images par seconde qui pourront être affichées. C'est pourquoi si vous avez un écran qui ne rafraîchit votre image que 60 fois par seconde, il est inutile d'avoir une carte graphique qui en débite 150, vous ne verrez pas la différence. Autant donc augmenter la qualité de l'image. Nous verrons plus bas quelles sont les techniques permettant d'augmenter la qualité des images.

La mémoire vidéo :

Variant généralement de 16 à 512 Mo, elle est utilisée pour stocker les textures (généralement sous formes d'images). Plus la taille de cette mémoire est importante, mieux c'est. Attention de ne pas non plus tomber dans l'inutile : 16 Mo (mémoire partagée, utilisée notamment par la technologie Turbo Cache que nous allons détailler plus bas) sont largement suffisants en utilisation bureautique et multimédia. Les jeux en revanche ne se satisferont pas d'une quantité aussi limitée, même si la mémoire vive vient en renfort avec cette technologie. La largeur du bus de la mémoire joue également beaucoup : un bus de 128 bit de largeur sera généralement moins performant qu'un 256 bit.

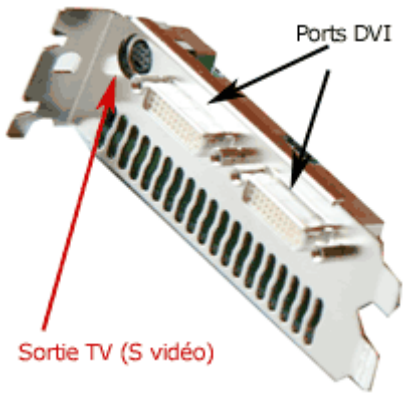
Aujourd'hui, il existe deux types de mémoire vidéo :

1°) La mémoire GDDR 2 (DDR pour Double Data Rate), exploitant les fronts montants et descendants de la mémoire, autrement dit la bande passante est doublée par rapport à la SD-RAM à même fréquence. On trouve plutôt ce type de mémoire sur les cartes graphiques bas de gamme.

2°) La mémoire GDDR 3 : presque identique à la GDDR 2, elle gagne en fréquence et sa tension d'alimentation diminue. Elle est toujours utilisée même dans les cartes graphiques les plus haut de gamme.

Les (éventuelles) entrées-sorties vidéos

VGA qui sert à relier les écrans cathodiques, DVI pour les écrans plats, et une éventuelle sortie TV généralement au format S Vidéo... La plupart des cartes possèdent maintenant une sortie TV, qui permet de regarder des DVD sur sa télé par exemple. La connectique de sortie est au format S-vidéo, avec, souvent, un adaptateur S-vidéo/composite. Quelques cartes possèdent aussi un tuner vidéo, qui permet de regarder la télé sur son PC, ou encore un port DVI en plus du port VGA. Le port DVI est numérique et ne nécessite pas la traduction des données par le RAMDAC. Équipée de telles cartes, l'ordinateur devient une véritable télé associée à un magnétoscope. Certaines cartes disposent de plusieurs sorties VGA pour pouvoir brancher deux écrans sur son ordinateur: L'intérêt du DUAL HEAD est de disposer d'un affichage plus étendu sous Windows. On peut ainsi ouvrir plus de programmes tout en se repérant mieux.



Performances graphiques :

Les performances d'une carte graphique dépendent en grande partie du processeur central. En effet, le processeur indique à la carte graphique tout ce qu'elle doit calculer. Cependant celui-ci devant aussi gérer les autres paramètres du jeu, il se peut qu'il ne fournisse pas assez d'informations, et alors la carte graphique doit attendre que le processeur ait terminé de mouliner. Il faut un processeur bien adapté à la carte vidéo afin d'éviter ce genre de piège. Mais il ne sert à rien d'avoir un processeur dernier cri avec une carte graphique d'entrée de gamme en utilisation ludique.
Plus la résolution augmente, plus le travail de la carte graphique sera important.

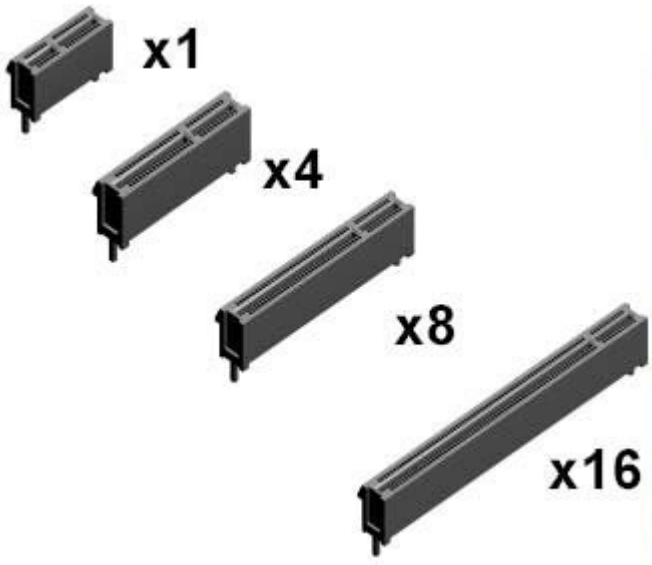
Les ports de connexion à la carte-mère :

La plupart des cartes graphiques se placent sur les ports AGP (pour Accelerated Graphic Port) et PCI Express. Le port AGP est apparu avec le Pentium II d'Intel en 1997. Celui-ci est totalement différent d'un port PCI tant physiquement que par sa vitesse. En effet le port AGP permet de partager la mémoire vive du PC quand la mémoire de la carte graphique est saturée. Cependant cette méthode d'accès est nettement plus lente que celle de l'accès à la mémoire de la carte graphique. On différencie trois normes d'AGP :

a) l'AGP 2X (500 Mo par seconde)
b) l'AGP 4X (1 Go par seconde)
c) l'AGP 8X (2 Go par seconde)

Les cartes graphiques PCI Express :

Le PCI Express est la norme qui a désormais remplacé l'AGP sur le marché des cartes graphiques. Plus rapide, existant en de nombreux formats, il remplace avantageusement les ports PCI et AGP. Il existe en différents débits.



Quelques technologies d'amélioration de qualité ou de performances :

Les constructeurs sont de plus en plus nombreux à vouloir ajouter des fonctionnalités supplémentaires à leurs produits. Cela va de la simple optimisation logicielle (au niveau des drivers) à des fonctions matérielles beaucoup plus puissantes. Nous allons en voir quelques unes et détailler pour chacune leurs avantages et inconvénients.

L'antialiasing :

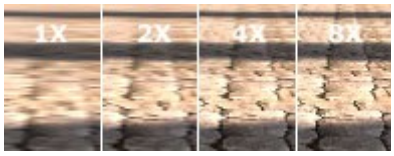
Sous ce terme un peu barbare se cache en fait une technologie destinée à améliorer la qualité de l'image, et notamment de ses arrêtes. Qui n'a en effet pas déjà pesté devant les images "en escaliers" ? l'antialiasing permet d'y remédier plus ou moins bien. Les cartes graphiques intègrent plusieurs niveaux d'antialiasing, ces niveaux peuvent aller de 2X à 32X. Plus le niveau choisi est élevé, meilleure est la qualité, mais plus les performances sont diminuées. Voici un aperçu de ce que peut donner une image sans puis avec antialiasing :

Sans antialiasing :

Avec antialiasing 4X

Le filtrage anisotrope :

L'antialiasing, vous pourrez le constater, provoque un léger effet de flou, surtout lorsque les arrêtes à retravailler sont lointaines. Le filtrage anisotrope vient palier à ce défaut. Il a été surtout conçu pour palier les défauts d'affichage des textures lointaines en les rendant plus nettes. Le filtrage anisotrope propose lui aussi différents niveaux de filtrage qui vont généralement de 2 à 16X. Plus le niveau de filtrage est élevé, plus les performances chutent (dans une moindre mesure qu'avec l'antialiasing toutefois). Voici ce que ça donne en images :

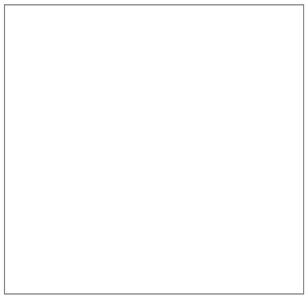


La technologie Turbo cache :

Implantée par NVIDIA sur ses cartes graphiques d'entrée de gamme, la technologie turbo cache est destinée à réduire le coût de production des cartes graphiques d'entrée de gamme mais aussi à récupérer des parts de marché sur les chipsets graphiques intégrés (qui offrent des performances généralement moindres que les cartes d'entrée de gamme). Cette technique consiste à n'implanter qu'une petite quantité de mémoire vive sur la carte graphique. Elle piochera alors d'abord sur sa mémoire disponible, et si elle en a besoin de plus, ira piocher dans la mémoire vive du PC. L'accès à la mémoire vive étant bien entendu nettement plus lent que l'accès à la RAM de la carte graphique. Il en résulte donc des performances moindres mais un coût qui l'est également. Cette technologie convient donc parfaitement pour ce à quoi elle est destinée, c'est à dire l'entrée de gamme. Les cartes équipées de la technologie turbo cache embarquent de 16 à 64 Mo de mémoire, mais attention aux emballages souvent estampillés "128 Mo" par exemple (les emballages affichant la taille de la mémoire que la carte peut s'accaparer au total).

Le SLI (Scalable Link Interface) :

Le SLI est une technique que l'on trouvait sur les très anciennes cartes 3DFX aujourd'hui disparues. Cette technique a depuis été remise au goût du jour par NVIDIA. Le SLI consiste à utiliser deux cartes graphiques qui vont se répartir le travail. Des algorithmes sont là pour répartir équitablement la charge entre les deux cartes, c'est à dire qu'une carte ne pourra calculer que 10% de la surface totale d'une image si celle-ci est gourmande en calculs, l'autre carte se chargeant des 90 % restants. Le bénéfice de cette technologie est surtout très important dès que l'on utilise les effets de qualité disponibles (antialiasing et filtrage anisotrope). Il permet de retarder la baisse de performances et permet au système d'être de plus en plus limité par le processeur du PC (l'évolution des processeurs graphiques étant généralement plus rapide).



Introduction aux vertices et à la géométrie de base en 3D

Qu'est-ce que les vertices ? A quoi ça sert ? Comment s'en servir ? C'est ce que nous allons voir dans cette partie dédiée à l'apprentissage de la géométrie de base en 3D.

Un vertex (singulier de vertices) est avant tout un point en programmation graphique, il peut contenir un certain nombre d'informations, par exemple, dans le cas d'un point 2d, un vertex contient au minimum 2 informations, les coordonnées x et y correspondant au point.

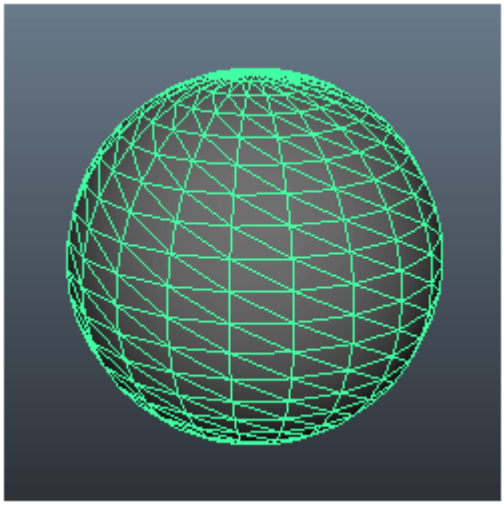
Dans un espace en 3d, un vertex va contenir « a minima » les coordonnées x,y et z correspondant au point. Deux points reliés ensemble forment une droite, ou une arrête, et plusieurs arrêtes reliées ensembles forment une face. Cette face peut avoir un nombre d'arrêtes totalement aléatoire par exemple :

- Une face a 3 arrêtes (ou côtés) est obligatoirement un triangle.
- Une face a 4 arrêtes (ou cotés) est obligatoirement un quadrilatère.
- Une face a 5 arrêtes (ou cotés) est obligatoirement un pentagone

Etc etc …

En programmation graphique, on utilise habituellement ce genre de face car la carte graphique les gère de manière native. Le triangle est la forme la plus répandue et sûrement la plus utilisée, on dit qu'on **utilise des faces de type triangle.**

Un ensemble de vertices nous permet donc de tracer des triangles en reliant chaque sommet, ce qui nous permet, en collant des triangles bout à bout de créer des formes plus complexes, comme une sphère en 3d par exemple.



Une sphère en 3d composée de triangles

Et bien ce genre de forme complexe porte un nom, en vérité toute suite de vertices permettant de créer des faces (triangulaires ou non) en vue de former un objet plus complexe s'appelle un **Mesh** ce qui signifie maillage en anglais. En effet, une suite de vertices peut s'apparenter à un maillage, comme celui des filets de pêche. Bien entendu, nous pouvons former des primitives telles que les sphères, cercles, cube, pyramide, triangles etc ... Mais également des formes beaucoup plus complexes telles d'un visage, une voiture, un avion, ou même un monde virtuel dans son ensemble !

Dans le but de comprendre comment cela fonctionne, il nous faut maintenant nous intéresser à la programmation, et en l'occurrence, à l'implémentation de ces concepts en Javascript avec WebGL.
Pour cela nous allons maintenant analyser et détailler un peu de code source.

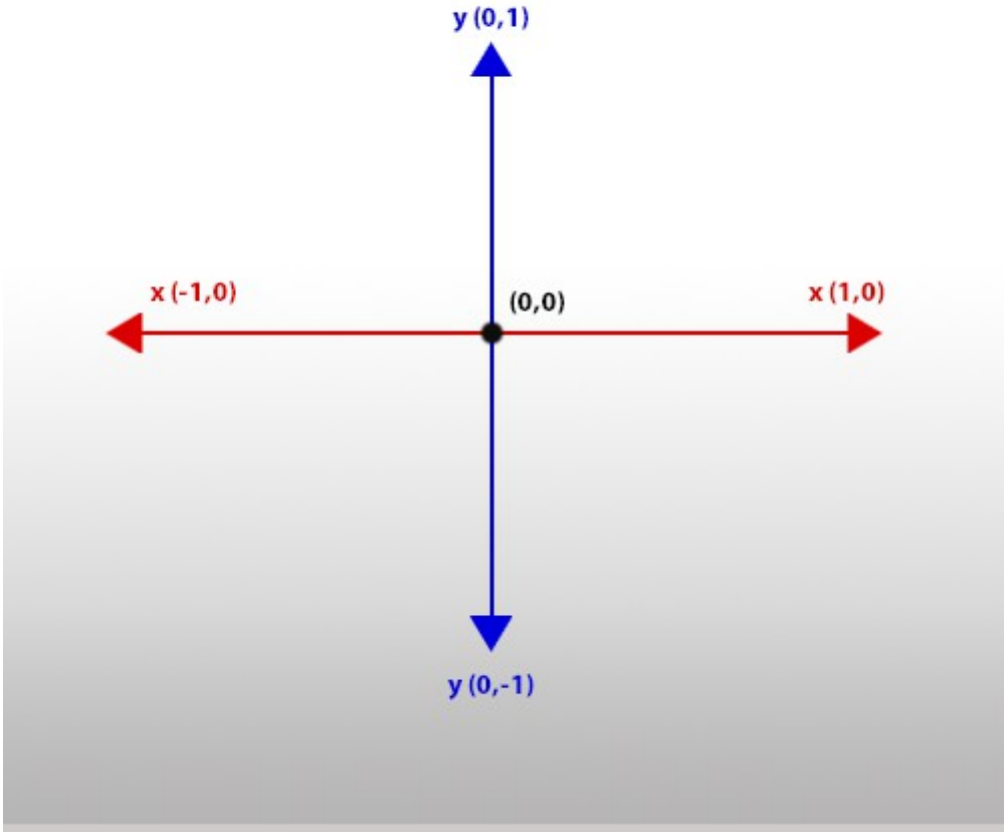
Rendez-vous dans le dossier "exercices" et ouvrez le dossier "exo1", vous y trouverez les fichiers suivants:

- index.html
- js/app.js
- js/glMatrix-0.9.5.min.js
- shaders/vertexshader.glsl
- shaders/pixelshader.glsl

Ouvrez-les nous allons les étudier un peu plus en détail, intéressons-nous à ce bout de code, que vous pourrez trouver dans le fichier app.js:

```
var vertices = [
            0.0,  1.0,  0.0,
           -1.0, -1.0,  0.0,
            1.0, -1.0,  0.0
];
```

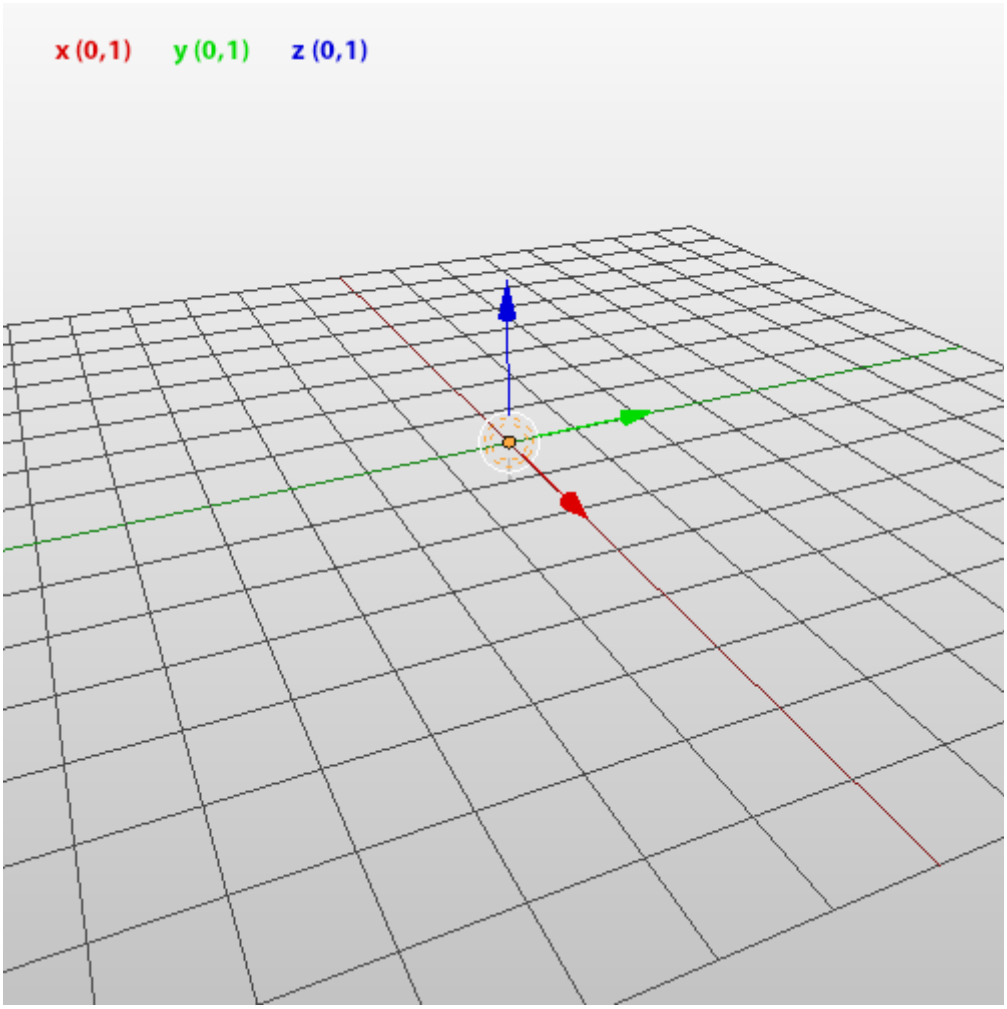
Comme vous pouvez le voir, nous avons crée un tableau Javascript à 9 élements. Ces 9 élements doivent être lues 3 par 3 (c'est d'ailleurs pour cela que nous avons indenté le code de cette manière).
Chaque groupe de 3 correspond aux coordonnées x,y,et z d'un point dans l'espace en 3 dimensions par défaut de WebGL que voici illustré ci-dessous:



Comme vous pouvez le voir, les coordonnées sont comprises entre -1 et 1 et le centre du repère se situe par défaut au centre du *viewport* ce qui correspond à l'heure actuelle au centre de notre canvas. Les coordonnées sont donc **normalisées** nous verrons plus tard ce que cela veut dire.

Pour l'instant retenez juste que si vous disposez d'un canvas d'une largeur de 500px par exemple, et que vous souhaitez placer un vertex à une position **x=125px**, sa valeur dans ce repère serait de **x=-0,5.**

Voici une autre représentation de cet espace en 3 dimensions :



Pour en revenir à nos vertices, nous aurions tout aussi bien pu créer une face à 4 arrêtes sans effort, il aurait suffit de changer quelques petites choses dans le code. Nous aurions pu ainsi afficher un quadrilatère. Nous allons maintenant nous intéresser plus en détail au code en introduisant le concept de **buffer** et de **programmes webgl**.

L'object WebGLContext :

Avant toute chose, il nous faut parler de l'objet *WebGLContext,* celui-ci représente notre sésame vers l'api WebGL. En effet, un objet de type WebGLContext agit comme une passerelle vers l'API WebGL, sans cet objet, programmer avec l'API WebGL est impossible.

Un objet de type WebGLContext se récupère à partir de l'appel à la méthode *getContext* d'un objet de type *HTMLCanvasElement* qui correspond ni plus ni moins à l'objet du DOM représentant votre balise <canvas></canvas>, intéressons-nous maintenant à ce code:

```
this._glContext = canvas.getContext("experimental-webgl");
```

Nous voyons bien que nous récupérons un objet de type *WebGLContext* en passant en paramètre la string **"experimental-webgl",** en effet, il est possible de récupérer un contexte 2d qui ne dispose pas de l'accélération matérielle et qui donc, n'a rien à voir avec **WebGL**.

Cet objet va nous servir aussi longtemps que l'on utilisera WebGL, en effet, comme dit précédemment, c'est un objet d'une importance cruciale.

Il se trouve que certains navigateurs ne gèrent pas WebGL, c'est pour cela que nous tentons de récupérer le contexte dans un bloc try catch, si le contexte WebGL ne peut être récupéré, nous stoppons l'application et nous affichons l'erreur à l'aide d'une popup « alert ».

L'objet WebGLBuffer :

Nous allons maintenant introduire la notion de **buffer**, que nous pouvons traduire en français par « **tampon** ». Un buffer est un élément qui comme son nom l'indique, va nous permettre de stocker des données au sein d'un tampon, un peu à la manière d'un tableau en Javascript (qui est techniquement un buffer), les lecteurs à l'aise avec le langage C seront probablement plus familiers avec cette notion. En Javascript avec WebGL, un buffer est une instance de l'instance *WebGLBuffer*.

Les données stockées seront ainsi passées à la carte graphique par le biais de méthodes fournies par l'API WebGL. Pour ce faire, nous allons devoir créer les buffers en question ainsi qu'un objet qui se trouve être **un programme webgl**. Nous allons d'abord nous intéresser aux buffers avant d'aborder les programmes webgl plus en détail et pour cela, regardons le code suivant qui se trouve dans notre code d'exemple :

```
Main.prototype.initBuffers          = function()
{
        var vertices = [
                 0.0,  1.0,  0.0,
                -1.0, -1.0,  0.0,
                 1.0, -1.0,  0.0
        ];

        this._buffer = this._ctx.createBuffer();
        this._ctx.bindBuffer(this._ctx.ARRAY_BUFFER, this._buffer);
        this._ctx.bufferData(this._ctx.ARRAY_BUFFER, new Float32Array(vertices),
this._ctx.STATIC_DRAW);
        this._custom.itemSize = 3;
        this._custom.numItems = 3;
}
```

Comme vous pouvez le constater, nous créons un objet de type WebGLBuffer à l'aide de la méthode "createBuffer" de l'objet WebGLContext dont voici la signature:

```
WebGLBuffer gl.createBuffer();
```

Pour stocker des données dans ce buffer, nous devons au préalable préciser au contexte webgl quel buffer nous allons utiliser et ce, à l'aide de la méthode, "bindBuffer" dont voici la signature:

```
void gl.bindBuffer(target, buffer);
```

Cette méthode prend 2 arguments en compte, le premier "target", correspond à ce que la documentation nous décrit comme le "binding point", sans rentrer dans de grandes considérations, il s'agit, à peu de choses près, du "type" de buffer que l'on souhaite manipuler. Ce paramètre est de type GL_ENUM, c'est à dire que seules quelques valeurs sont acceptables, en l'occurrence les valeurs suivantes:

a) gl.ARRAY_BUFFER: utilisé pour les buffers qui contiennent des vertices, des attributs (nous verrons plus tard de quoi il s'agit) des coordonnées uv (nous verrons également cela plus tard) ou des couleurs de vertices.

b) gl.ELEMENT_ARRAY_BUFFER: Qui est utilisée pour les buffers contenant des indices de vertices.

Et enfin, pour pousser les données dans le buffer, nous faisons appel à la méthode bufferData dont voici l'une des signatures possibles:

```
void gl.bufferData(target, data, usage);
```

- Le paramètre "target" est le même que celui de la méthode précédente
- Le paramètre "data" correspond aux données que l'on souhaite pousser dans le buffer, comme vous pouvez le voir, nous convertissons notre tableau de vertices, en Float32Array, pour des raisons de performances et surtout pour être sur et certain que chaque élement sera un entier codé sur 32 bits.
- Le paramètre "usage" indique la façon dont on va se servir du buffer, les 3 cas suivants sont possibles:

a) gl.STATIC_DRAW: Indique que le contenu du buffer est susceptible d'être souvent utilisé mais très peu actualisé, ce qui correspond bien à notre utilisation du buffer.

b) gl.DYNAMIC_DRAW: Indique que le contenu du buffer est susceptible d'être utilisé et actualisé souvent.

c) gl.STREAM_DRAW: Indique que le contenu du buffer est susceptible de ne pas être beaucoup utilisé

Pour des raisons de performances, il est important de bien choisir la valeur que l'on donne à ce dernier paramètre. En fait il est très important de faire attention à à peu près tout lorsqu'on code en WebGL, cela nous permettra de conserver des performances optimales, et ce quelque soit le type d'application que l'on souhaite créer (dans les limites du langage bien entendu).

L'objet WebGLProgram et les Shaders:

Un programme webgl est une instance de l'objet *WebGLProgram* qui a pour seul et unique but de **passer des données à la carte graphique** et de lui préciser la façon dont elle va devoir les traiter. Cela peut être les données contenues dans les buffers,des shaders (sous forme compilée), ou des données numériques brutes que les shaders pourront utiliser. En fait, il s'agit ni plus ni moins de la combinaison de 2 shaders de types différents:

- 1 Pixel Shader (ou fragment shader)
- 1 Vertex Shader

Mais qu'est-ce qu'un shader et a quoi cela sert-il ? Un shader est un programme écrit dans un langage spécifique et compilé dans un pseudo assembleur compréhensible par la carte graphique. La principale fonction d'un shader est de manipuler des données numériques ou des pixels (suivant le type du shader) à l'aide de fonction mathématiques simples. Pourquoi ne pas le faire en amont me direz-vous ? Et bien il se trouve que la carte graphique est spécialisée dans ce type de traitement, ses composants matériels ont été optimisés et crées exprès pour agir des centaines de fois plus vite qu'un processeur classique, c'est pour cela que l'on parle courramment d'accélération matérielle, lorsqu'on utilise une API qui communique avec la carte graphique.

Il existe plusieurs langages de shaders comme HLSL, AGAL, GLSL etc... C'est d'ailleurs ce dernier langage qui est le plus utilisé, et ça tombe bien, car c'est celui que nous utilisons avec WebGL. Il est également indispensable de mentionner la différence entre un Fragment Shader (aussi appellé Pixel Shader) et un Vertex Shader.

Voici le contenu du fichier shaders/vertexshader.glsl écrit en GLSL:

```
attribute vec3 aVertexPosition;

uniform mat4 uMVMatrix;
uniform mat4 uPMatrix;

void main(void)
{
        gl_Position = uPMatrix * uMVMatrix * vec4(aVertexPosition, 1.0);
}
```

et le contenu du fichier shaders/pixelshader.glsl, écrit lui aussi en GLSL:

```
precision mediump float;

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

Maintenant que nous savons à quoi cela ressemble, voyons maintenant à quoi cela sert.

Le Vertex Shader:

C'est un shader spécialisé dans le traitement des données numériques et géométriques de type vertices, matrices etc... Ainsi que les données associées à ces derniers. En gros, c'est un programme qui va manipuler chaque composant de chaque vertex de manière individuelle, il peut également traiter certaines données numériques comme des valeurs scalaires, des matrices, des vecteurs simples et éventuellement les passer au pixel shader.

Dans notre exemple, il est le shader dont l'id est "shader-vs" et est de type "x-shader/x-vertex". Pour fonctionner, il a besoin de plusieurs paramètres que l'on se charge de passer au programme webgl. Dans un premier temps on déclare ce que l'on appelle un attribut, il s'agit d'un type de données qui est amené à varier souvent.

Ici on s'en sert pour stocker la position des vertices dans un tableau de données, que l'on peut donc appeler *Vertex Array.* Cet attribut est déclaré comme étant un conteneur de données de type "vec3", ce qui signie que les données à l'intérieur seront automatiquement interprétées comme des vecteurs à 3 dimensions.

Les deux autres variables utilisées sont ce que l'on appelle des uniform, çàd des données dont les données seront constantes. Au sein de notre shader, l'on déclare donc 2 uniform de type mat4, ce qui signifie que les données passées à ces deux "uniform" seront traitées comme des matrices 4x4.

On peut voir que ces matrices sont utilisées en combinaison avec les vertices dans le but de déterminer une position, en fait il s'agit de la position finale du vertex actuellement traité. Le même traitement sera appliqué à chacun des vertices.

Le Fragment Shader ou Pixel Shader:

C'est un shader spécialisé dans le traitement des couleurs appliqués à chacun des pixels, ce qui veut dire que le programme va être éxécuté pour chaque pixel que l'on souhaite afficher à l'écran. Il peut récupérer certaines variables que lui passe le vertex shader, opérer des transformations sur les pixels déjà présents, changer des couleurs, appeller des fonctions de texturing etc etc ...

Il est le shader dont l'id est "shader-fs" et est de type "x-shader/x-fragment". Pour l'instant, il se contente de définir une couleur unique à chaque pixel contenu dans les triangles formés les vertices. Les différents composants de cette couleur sont décrits en base 1 pour chaque canal (RGBA). Actuellement, l'on remplit chaque pixel de blanc, mais nous avions voulu changer la couleur, il nous aurait fallu opérer le traiement suivant:

```
vec4( r/255, g/255, b/255, a/255);
```

ou rgba sont les valeurs de chaque canal de couleur dont les valeurs sont traditionnellement comprises entre 0 et 255. On voit également qu'une ligne est dédiée à la *précision* nous y reviendrons plus tard, pour l'instant nous allons nous intéresser aux données que manipule le vertex shader à savoir les matrices et les vertex arrays.

Vertex Array

Un vertex Array est juste un tableau de vertices que l'on envoit à un objet de type WebGLProgram afin qu'il puisse les traiter au sein d'un vertex shader. Une fois traités, ces vertices serviront à définir les sommets de primitives, en général des triangles. Puis, une fois ces primitives dessinées, le pixel shader se charge de remplir chacun des pixels présents à l'intérieur de ces primitives (dans notre exemple chaque pixel de chaque primitive est blanc). Dans notre vertex shader, le vertex Array correspond à l'attribut "aVertexPosition".

Les matrices:

Les matrices sont des outils mathématiques utilisés dans beaucoup de domaines dont la 3D. On peut les représenter comme un tableau en deux dimensions contenant des valeurs scalaires (entières, réelles ou complexes).

On peut réaliser sur les matrices des opérations proches de celles que l'on applique sur les nombres réels comme l'addition, la soustraction, la multiplication ou l'inversion. Cependant, à cause de leur nature plus complexe, certaines de ces opérations peuvent être définies de plusieurs façons différentes. Par exemple la multiplication peut être membre à membre ou utiliser une formule qui multiplie les lignes de l'une contre les colonnes de l'autre.

Une matrice avec M lignes et N colonnes est appelée une matrice MxN. Le gros avantage des matrices, c'est qu'elle peuvent également nous servir à appliquer des transformations à des vecteurs à *n dimensions,* donc nous pouvons techniquement appliquer des transformations à des vertices, et c'est exactement ce que nous faisons au sein du vertex shader.

En effet, nous multiplions les deux matrices contenues dans les uniform entre elles, puis nous multiplions le résultat obtenu par le vecteur à 4 dimensions composés des valeurs x,y,z de chaque vertex + une quatrième coordonnées (traditionnellement notée "w") que l'on définit à 1,0. Une annexe un peu plus complète sur les matrices est disponible en fin de document.

Voyons d'ailleurs pourquoi webGL a besoin des matrices :

- Pour la projection : grâce à cela nous allons pourvoir "transformer" un monde 3D en un monde 2D.

- Pour les transformations : regroupant les transformations que je vous ai énumérées à l'instant.

Malheureusement, webGL ne nous procure aucune gestion native des matrices, nous allons donc utiliser une librairie écrite en javascript qui le fera pour nous (Même si nous pourrions en écrire une nous-mêmes), j'ai nommé, la librairie **glMatrix**. Mais avant de commencer à les manipuler nous allons faire un peu de théorie pour bien comprendre ce que nous faisons, nous aurons besoin de deux matrices:

- La matrice de projection : qui sert à transformer notre monde 3D en un monde 2D affichable sur l'écran.

- Une matrice qui sera la matrice principale, c'est sur elle que nous allons appliquer nos transformations, dans notre shader il s'agit de la matrice **uMVmatrix** (pour user Model View matrix).

Ce que le shader va faire, c'est multiplier la matrice de projection par la matrice model view, puis multiplier chaque vertex par le résultat obtenu, nous donnant ainsi une impression de 3 dimensions. A présent, voyons comment nous allons envoyer les données à nos shaders par le biais de l'objet webGLProgram:

```
Main.prototype.getShader        = function(srcURL, type)
{
        var str = null;
        var shader = this._ctx.createShader(type);
        var http = new XMLHttpRequest();

        http.open("GET",srcURL, false); // synchronous call
        http.send();

        str = http.responseText;

        this._ctx.shaderSource(shader, str);
        this._ctx.compileShader(shader);

        if (!this._ctx.getShaderParameter(shader, this._ctx.COMPILE_STATUS))
        {
                alert(this._ctx.getShaderInfoLog(shader));
                return null;
        }

        return shader;
};

Main.prototype.initShaders      = function()
{
        var fragmentShader = this.getShader("shaders/pixelshader.glsl", this._ctx.FRAGMENT_SHADER);
        var vertexShader = this.getShader("shaders/vertexshader.glsl", this._ctx.VERTEX_SHADER);

        this._prgm = this._ctx.createProgram();
        this._ctx.attachShader(this._prgm, vertexShader);
        this._ctx.attachShader(this._prgm, fragmentShader);
        this._ctx.linkProgram(this._prgm);

        if (!this._ctx.getProgramParameter(this._prgm, this._ctx.LINK_STATUS))
        {
                alert("Could not initialise shaders");
        }

        this._ctx.useProgram(this._prgm);

        this._custom.vertexPositionAttribute = this._ctx.getAttribLocation(this._prgm, "aVertexPosition");
        this._ctx.enableVertexAttribArray(this._custom.vertexPositionAttribute);

        this._custom.pMatrixUniform = this._ctx.getUniformLocation(this._prgm, "uPMatrix");
        this._custom._mvMatrixUniform = this._ctx.getUniformLocation(this._prgm, "uMVMatrix");
}
```

Comme vous pouvez le constater, nous obtenons nos shaders par le biais de le fonction "getShader" qui va dans l'ordre:

- Créer un objet de type shader à l'aide de la fonction "createShader" de l'objet de type webGLContext
- Récupérer le code source du shader dans le fichier dédié à l'aide d'un appel ajax.
- Attribuer le code source ainsi obtenu au shader à l'aide de la méthode "shaderSource" du contexte.
- Compiler le shader à l'aide de la méthode "compileShader", puis vérifier que la compilation s'est effectuée sans encombre.

De retour dans la fonction "initShaders", nous créeons un objet de type webglProgram à l'aide de la méthode "createProgram" du contexte webGL, puis nous attachons les shaders ainsi compilés. Une fois le programme lié au contexte, à l'aide de la méthode "linkProgram", nous précisons à ce dernier que nous comptons l'utiliser à l'aide de la méthode "useProgram".

Une fois le programme lié et attribué en tant que programme courant, nous appelons la méthode "getAttribLocation". Cette méthode va nous retourner un entier qui correspondra à un emplacement en mémoire dédié à la variable dont nous précisons le nom en second paramètre, en l'occurrence, la variable "**aVertexPosition**".

Une fois l'entier stocké (sur l'objet _custom qui nous servira à stocker pas mal de variables), nous appellons la méthode "enableVertexAttribArray" qui va nous permettre,comme son nom l'indique, d'activer l'attribut en tant que tableau... Mais qu'est-ce que cela veut dire ?

Par défaut, en webGL, **beaucoup de choses sont désactivées pour des raisons de performances**, dans le cas d'un attribut comme le nôtre, cela veut dire qu'il faut activer le fait que la variable correspondante contiendra un tableau de données, si nous ne passons pas par cette étape rien ne se produira, et une erreur sera levée, ils nous faut donc l'activer.

Nous passons ensuite aux matrices, nous procédons de la même façon que pour la variable "aVertexPosition" à ceci près que nous appellons la méthode "getUniformLocation", ce qui veut dire que les données qui seront envoyées à l'emplacement réservé en mémoire seront de type **uniform**. En GLSL, une variable de type uniform contient des données qui ne sont pas vouées à être changées, on peut assimiler cela à des **constantes**.

Maintenant que nos shaders sont initialisées et les variables des shaders créées, nous pouvons passer à la prochaine étape qui consiste à assigner les valeurs stockées dans nos buffers à ces variables, pour finalement enclencher l'étape finale, le dessin de la scène, examinons le code suivant:

```
Main.prototype.drawScene	= function()
{
	this._ctx.viewport(0, 0, this._custom.viewportWidth, this._custom.viewportHeight);
	this._ctx.clear(this._ctx.COLOR_BUFFER_BIT | this._ctx.DEPTH_BUFFER_BIT);

	mat4.perspective(45, this._custom.viewportWidth / this._custom.viewportHeight, 0.1, 100.0,
this._pMatrix);
	mat4.identity(this._mvMatrix);
	mat4.translate(this._mvMatrix, [-1.5, 0.0, -7.0]);

	this._ctx.bindBuffer(this._ctx.ARRAY_BUFFER, this._buffer);
	this._ctx.vertexAttribPointer(this._custom.vertexPositionAttribute, this._custom.itemSize,
this._ctx.FLOAT, false, 0, 0);

	// set matrices uniform
	this._ctx.uniformMatrix4fv(this._custom.pMatrixUniform, false, this._pMatrix);
	this._ctx.uniformMatrix4fv(this._custom._mvMatrixUniform, false, this._mvMatrix);

	this._ctx.drawArrays(this._ctx.TRIANGLES, 0, this._custom.numItems);
}
```

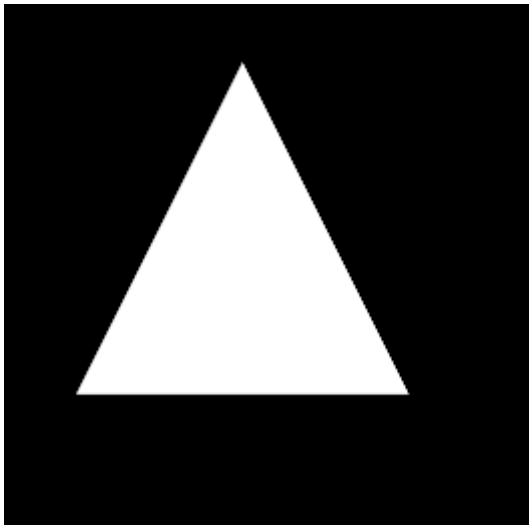
Nous avons déjà détaillé plus haut l'assignation des données aux buffers, lorsque nous avons introduit ceux-ci, ici, ce qui nous intéresse vraiment ici ce sont les appels aux méthodes "vertexAttribPointer" et "uniforMatrix4fv". Pour la première, il nous suffit de préciser que le buffer courant est le vertex buffer (à l'aide de la méthode bindBuffer du contexte) puis de préciser la forme qu'ont ces données ainsi que l'espace en mémoire auquel elles doivent être envoyées. Voici le rôle des différents paramètres que nous avons envoyés:

- **this._custom.vertexPositionAttribute :** l'entier qui pointe vers l'espace en mémoire dédiée à la variable "aVertexPosition", les données du buffer seront poussées à cet endroit en mémoire.

- **this._custom.itemSize:** la taille d'un élément dans le tableau de données, en l'occurrence 3. Nous précisons donc à l'API webGL que les données devront êtres lues 3 par 3.

- **this._ctx.FLOAT:** Il s'agit d'une constante de l'objet contexte qui précise que chaque élément du tableau devra être lu sous la forme d'un nombre flottant.

Nous ne nous intéresserons pas de suite aux derniers paramètres. L'appel à la méthode "*uniforMatrix4fv*" obéit à peu près à la même logique que la méthode précédente, à ceci près que puisque les valeurs que l'on va pousser à cet endroit sont constantes, on peut directement les envoyer sans passer par un quelconque buffer, en visualisant la page, voici ce que vous devriez obtenir:



Exercice:

A l'aide de l'annexe sur les matrices et du cours, essayez de manipuler le triangle dans tout les sens. Appliquez-lui des transformations de type translation, rotation, homothétie etc etc...

Introduction aux IndexBuffers:

Comme vous avez pu le voir, à l'heure actuelle, nous dessinons nos triangles à l'aide de la méthode "drawArrays" de l'objet de type webGLContext, ces triangles sont dessinés dans l'ordre dans lequel nous les présentons au sein du vertexBuffer, et aucun vertex n'est réutilisé pour l'affichage. Nous allons voir maintenant comment attribuer un index à chaque vertex, nous permettant ainsi d'en réutiliser certains et de dessiner les triangles dans un ordre totalement arbitraire.

Examinons le code suivant (extrait de l'exercice exo2):

```
Main.prototype.initBuffers          = function()
{
        var vertices = [
                //triangle 1
                 0.0,  1.0,  0.0,
                -1.0, -1.0,  0.0,
                 1.0, -1.0,  0.0,

                 //triangle 2

                2.0,  1.0,  0.0,
                1.5, -1.0,  0.0,
                2.5, -1.0,  0.0,
        ];

        var indices = [0,1,2,3,4,5];

        this._buffer = this._ctx.createBuffer();
        this._ctx.bindBuffer(this._ctx.ARRAY_BUFFER, this._buffer);
        this._ctx.bufferData(this._ctx.ARRAY_BUFFER, new Float32Array(vertices),
this._ctx.STATIC_DRAW);
        this._custom.itemSize = 3;
        this._custom.numItems = vertices.length / 3;

        this._indexBuffer = this._ctx.createBuffer();
        this._ctx.bindBuffer(this._ctx.ELEMENT_ARRAY_BUFFER, this._indexBuffer);
        this._ctx.bufferData(this._ctx.ELEMENT_ARRAY_BUFFER, new Uint16Array(indices),
this._ctx.STATIC_DRAW);
        this._custom.indexItemSize = 1;
        this._custom.indexNumItems = indices.length;
};
```

```
Main.prototype.drawScene	= function()
{
	this._ctx.viewport(0, 0, this._custom.viewportWidth, this._custom.viewportHeight);
	this._ctx.clear(this._ctx.COLOR_BUFFER_BIT | this._ctx.DEPTH_BUFFER_BIT);

	mat4.perspective(	45,
				this._custom.viewportWidth / this._custom.viewportHeight,
				0.1,
				100.0,
				this._pMatrix
			);

	mat4.identity(this._mvMatrix);
	mat4.translate(this._mvMatrix, [-1.5, 0.0, -7.0]);

	this._ctx.bindBuffer(this._ctx.ARRAY_BUFFER, this._buffer);
	this._ctx.vertexAttribPointer (		this._custom.vertexPositionAttribute,
						this._custom.itemSize,
						this._ctx.FLOAT,
						false,
						0,
						0
				);

	this._ctx.bindBuffer(this._ctx.ELEMENT_ARRAY_BUFFER, this._indexBuffer);

	this._ctx.uniformMatrix4fv(this._custom.pMatrixUniform, false, this._pMatrix);
	this._ctx.uniformMatrix4fv(this._custom._mvMatrixUniform, false, this._mvMatrix);

	this._ctx.drawElements(t	his._ctx.TRIANGLES,
					this._custom.indexNumItems,
					this._ctx.UNSIGNED_SHORT, 0);
};
```

Ce code est assez similaire au code précédent à quelques détails près:

- Nous créons un nouveau buffer de type ELEMENT_ARRAY_BUFFER, les données que l'on pousse à l'intérieur sont des entiers non signés codés sur 16bits, qui se situent donc entre 0 et 65535. Ces entiers représentent les indices des vertices, utilisés pour dessiner des triangles.

- Nous ne faisons plus appel à la méthode "drawArrays" mais à la méthode "drawElements", qui prend en paramètre le type de primitive que l'on dessine, le nombre d'items que le compte dessiner(ici nous dessinons l'ensemble des éléments représentés dans l'index Buffer) , le type des éléments un (entier court) donc un entier codé sur 16 bits.

Nous avons demandé à la carte graphique de dessiner les vertices dans un ordre précis, amusez-vous à changer cet ordre, à réutiliser les vertices, à tout déplacer. vous verrez c'est simple.

Exercice:

A l'aide du formateur et du cours, ajoutez un attribut au shader permettant de contenir une couleur codée représentée par un vecteur à 4 élements et dessinez deux triangles de couleur différentes.

Introduction aux textures:

Nous allons maintenant apprendre à utiliser des textures en lieu et place des couleurs pour remplir nos primitives. Mais tout d'abord définissons ensemble ce qu'est une texture.

Une texture est en jeu de données graphique que l'on envoit à la carte graphique afin qu'elle l'applique à une forme qu'elle dessinée précédemment. En règle générale, on emploie des images, mais cela pourrait être une vidéo, une image générée dynamiquement etc etc... Bref, tout ce qui peut être interprété comme étant une donnée graphique. Toutefois, dans nos exemples, nous utiliserons exclusivement des textures dont la source graphique seront des images.

Analysons le code source de l'exerice exo4, comme vous pouvez le constater nous avons effectué quelques petits changements notamment la fonction initTexture:

```
Main.prototype.initTexture = function()
{
	this._image 	= document.getElementById("texture");
	this._texture = this._ctx.createTexture();
	this._ctx.bindTexture(this._ctx.TEXTURE_2D, this._texture);
	this._ctx.pixelStorei(this._ctx.UNPACK_FLIP_Y_WEBGL, true);
	this._ctx.texImage2D(this._ctx.TEXTURE_2D, 0, this._ctx.RGBA, this._ctx.RGBA,
this._ctx.UNSIGNED_BYTE, this._image);
	this._ctx.texParameteri(this._ctx.TEXTURE_2D, this._ctx.TEXTURE_MAG_FILTER,
this._ctx.NEAREST);
	this._ctx.texParameteri(this._ctx.TEXTURE_2D, this._ctx.TEXTURE_MIN_FILTER,
this._ctx.NEAREST);
	this._ctx.bindTexture(this._ctx.TEXTURE_2D, null);
};
```

- Cette fonction crée une texture en appellant la méthode "createTexture" du contexte webGL.

- Cette texture est ensuite relié au contexte par le biais de la fonction méthode "bindTexture".

- Puis, nous précisons que nous voulons faire en sorte de "renverser" la texture sur l'axe des Y

- Enfin nous envoyons les données contenues dans l'image (récupérée depuis le DOM dans notre exemple) à l'aide de méthode texImage2D.

- Puis nous choisissons les filtres de texture que nous souhaitons voir utiliser lorsque la texture se retrouve étirée, en l'occurrence, nous choisissons des filtres qui font en sorte de prendre le pixel le plus proche en cas d'étirement de la texture ou de compression de la texture.

Ensuite, nous déclarons un nouveau buffer dans la fonction initBuffers dans lequel nous poussons des données. Ces données correspondent aux coordonnées uv qui seront utilisées au moment de texturer le pixel en cours dans le fragment shader. **Mais que sont les coordonnées UV et à quoi servent-elles** ?

Les coordonnées UV:

Il y a plusieurs façons de plaquer des textures en programmation graphique, mais le plaquage de texture à l'aides des coordonnées UV reste quand même celui qui offre le plus de liberté. En gros, cela revient à associer à un vertex un point de la texture, ce point étant bien entendu normalisé, çàd que si on souhaite représenter le point le plus à droite de la texture, celui-ci vaudra x = 1, le point le plus haut quand à lui vaudra y = 1.

Attention, j'ai bien parlé ici du point le **plus haut** cela veut dire que si l'on prend comme origine le point en haut à gauche, ses coordonnées uv seront x = 0 et y = 1. Les coordonnées uv prennent leur origine dans le coin en bas à gauche d'une texture.

Mais on peut faire en sorte de changer cela en inversant les données graphiques contenues dans la texture, à l'aide de l'option vu plus haut (UNPACK_FLIP_Y_WEBGL).

Pour le coup, si nous voulons associer des coordonnées uv à une texture, il nous faudra les envoyer à la carte graphique, et pour cela, un nouveau buffer est tout indiqué. Il nous faut aussi stocker et envoyer les données graphiques (l'image) à la carte graphique, pour cela un nouvel uniform un peu spécial sera tout indiqué (uSampler dans notre shader).

Et enfin, pour obtenir la couleur finale d'un pixel, nous appelons la fonction GLSL texImage2D à laquelle nous envoyons les données graphiques et les coordonnées uv, elle nous renverra un vecteur de type vec4 qui correspondra à notre couleur finale.

Maintenant que nous savons appliquer des textures à des formes, voyons comment traiter des objets transparents à l'aide de l'alpha blending.

L'alpha blending et le depth buffer ou tampon de profondeur

Lorsque vous dites à WebGL de dessiner quelque chose, cela passe par plusieurs étapes de traitement. D'un point de vue général, il :

1. exécute le vertex shader sur tous les vertex ;
2. effectue une interpolation linéaire entre les vertex, qui renseignent les fragments (que vous pouvez pour le moment considérer comme des pixels) qui ont besoin d'être dessinés ;
3. exécute sur chaque fragment le fragment shader pour déterminer sa couleur ;
4. l'écrit dans le frame buffer.

Pour le moment, le frame buffer est ce qui est finalement affiché. Mais que se passe-t-il si vous dessinez deux choses ? Par exemple, si vous dessinez un carré de centre (0, 0, -5) et un autre de même taille en (0, 0, -10) ? Vous ne voudriez pas que le second écrase le premier, parce qu'il est nettement plus loin et devrait être caché.

WebGL gère ceci avec le tampon de profondeur. Suite au traitement par le fragment shader, lorsque les fragments sont écrits dans le frame buffer, de même que les valeurs de couleurs RGBA, la valeur de la profondeur liée est aussi stockée. Ce n'est pas exactement la même valeur que la *Z value* associée au fragment. (Sans surprise, le tampon de profondeur est souvent aussi appelé le Z-buffer.)

Quentends-t'on par "liée ?" Eh bien WebGL aime que les valeurs de profondeur soient entre 0 et 1, 0 pour le plus proche et 1 le plus éloigné. Tout ceci est caché par la matrice de projection créée par l'appel à la méthode "perspective" au début de notre méthode "drawScene". Tout ce que vous devez savoir pour le moment est que plus la valeur du tampon de profondeur sera grande, plus l'élément sera éloigné.

Voilà pour le tampon de profondeur. maintenant voyons un peu cet extrait de code:

```
gl.enable(gl.DEPTH_TEST);
```

Il s'agit d'une instruction destinée au système WebGL indiquant ce qu'il faut faire lors de l'écriture d'un nouveau fragment dans la mémoire du frame buffer. Essentiellement, elle signifie « prend en considération le tampon de profondeur ». Il est combiné avec un autre paramètre WebGL, la fonction de la profondeur, font la valeur pa défaut est:

```
gl.depthFunc(gl.LESS);
```

Cela signifie « si notre fragment a une valeur de profondeur inférieure à la valeur actuelle, utilise la nouvelle plutôt que l'ancienne ». Ce test à lui seul, combiné au code pour l'activer suffit à nous donner un comportement logique : les objets proches dissimulent les plus éloignées. (Vous pouvez également utiliser d'autres valeurs pour la fonction de profondeur, mais je soupçonne qu'elles soient utilisées plus rarement.)

Le mélange est simplement une alternative à ce procédé. Avec le test de profondeur, la fonction de profondeur permet de faire le choix de remplacer ou non le fragment existant par le nouveau.
Lorsque nous effectuons le mélange, nous utilisons une fonction de mélange pour combiner les couleurs du fragment actuel et du nouveau pour en faire un tout nouveau, que nous écrivons alors dans le tampon.

Voyons maintenant le code suivant:

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE);
```

Les paramètres de cette fonction définissent la façon dont le mélange est fait, définissons tout d'abord deux termes :

- Le fragment source est le fragment que nous dessinons en ce moment
- Le fragment de destination est celui qui est déjà dans le frame buffer.

Le premier paramètre de la fonction détermine le facteur de source, et le second le facteur de destination. Ces facteurs sont utilisés dans la fonction de mélange, vous pouvez consulter une annexe à la fin du document précisant comment sont effectuées les combinaisons de couleur.

A noter cependant que **le mélange n'est pas la même chose que la transparence**, c'est juste une technique utilisée (parmi d'autres) pour obtenir des effets de transparence.

Autre point important, **nous devons désactiver le test de profondeur, pour que le mélange s'effectue.** Si nous ne le faisons pas, le mélange n'aura lieu que dans certains cas et pas dans d'autres. Par exemple, si nous dessinons une primitive se trouvant derrière une autre, alors lorsqu'elle sera tracée, elle sera écrite dans le frame buffer, puis celle située en avant sera mélangée par-dessus, ce qui est bien ce que nous voulons.

Cependant, **si nous dessinons la face de devant en premier, la face arrière sera rejetée par le test de profondeur avant d'arriver à la fonction de mélange**, de sorte qu'elle ne contribuera pas à l'image. Et ce n'est pas ce que nous voulons.

Il y a dans le mélange, **une forte dépendance de l'ordre** dans lequel vous dessinez les éléments que nous n'avions pas rencontrés dans les leçons précédentes. Finalement **comment obtenir une transparence correcte** ?

Eh bien la documentation nous dit que vous devez utiliser un facteur source SRC_ALPHA et un facteur destination de ONE_MINUS_SRC_ALPHA. Cela nous donne déjà une solution quand à la fonction de blending à utiliser mais qu'en est-il de l'ordre des primitives ?

La documentation nous dit que:

« *Lorsque vous utilisez un tampon de profondeur dans une application, vous devez faire attention à l'ordre de rendu des primitives. Les primitives totalement opaques doivent être rendues en premier, suivies par les partiellement opaques de l'arrière vers l'avant. Si vous ne respectez pas cet ordre, les primitives qui seraient normalement visibles au travers d'une primitive opaque, pourraient totalement échouer au test de profondeur*. »

Et voilà. La transparence par le mélange est assez délicate et fastidieuse, et cela demande beaucoup d'efforts pour quelque chose qui reste finalement peu utilisé dans les applications 3D modernes. En règle générale, le fait d'ordonner chaque triangle transparent est assez coûteux en termes de performances, ce qui nous obligera dans un premier temps à en utiliser peu. Maintenant que nous en avons fini avec le tampon de profondeur, passons à **l'ajout de lumière**.

Les lumières:

WebGL n'a absolument aucun support **intégré** pour l'éclairage. À la différence d'OpenGL, qui vous permet de spécifier au moins huit sources lumineuses, et gère tout à votre place, WebGL vous laisse tout faire vous-même. Mais si vous êtes à l'aise avec ce qui touche aux shaders, l'éclairage s'avère beaucoup plus simple à appréhender,

Notre objectif est de pouvoir simuler un certain nombre de sources de lumière dans la scène. Ces sources n'ont pas besoin d'être visibles elles-mêmes, mais elles ont besoin d'éclairer les objets 3D de façon réaliste, afin que le côté de l'objet exposé à la lumière soit brillant et le côté éloigné de la lumière soit sombre. Autrement dit, nous voulons être en mesure de spécifier un ensemble de sources de lumière, puis nous voulons travailler sur la façon dont l'ensemble des lumières affecte chaque partie de notre scène 3D.

Pour l'instant nous ne le ferons que pour une lumière, employer plusieurs lumières reviendra à répéter la même procédure pour chacune et additionner les résultats.

Note:
Puisque nous travaillons sur l'éclairage par vertex, les effets de la lumière sur les pixels situés entre les vertex seront calculés par l'interpolation linéaire habituelle. Cette technique est appelée "per vertex lighting" ou "éclaire par vertex".

Ce qui signifie que les espaces entre les vertex seront illuminés comme s'ils étaient à plat, pour les surfaces courbées, où vous voulez calculer les effets de l'éclairage pour chaque pixel de façon indépendante, vous pouvez utiliser une technique appelée *éclairage par fragment* (ou *par pixel*), que nous verrrons plus tard.

Notre objectif est d'écrire un shader réaliste qui nous permette de simuler un certain type de lumière, pour cela nous partir d'un modèle décrivant le comportement de certaines lumières, **le modèle de réflexion Phong.** Alors qu'il n'y a dans le monde réel qu'un type de lumière (la lumière se comporte toujours de la même façon à quelques exceptions près), en programmation graphique, on peut prétendre qu'il y en a deux pour les graphismes:

- 1 La lumière qui vient d'une direction spécifique qui n'éclaire que les objets situés dans cette direction. Nous l'appellerons lumière directionnelle,

- 2 La lumière qui vient de partout et éclaire tout de façon uniforme, indépendamment de la face exposée. On l'appelle lumière ambiante.

Lorsque la lumière frappe une surface, elle se reflète de deux façons:

- 1 **Diffuse** : c'est-à-dire indépendamment de l'angle d'attaque, elle rebondit de manière égale dans toutes les directions. Quel que soit l'angle d'observation, la brillance de la lumière reflétée est entièrement commandée par l'angle sous lequel la lumière frappe la surface. Plus l'angle d'incidence est élevé, plus la réflexion est sombre. La réflexion diffuse pourrait faire penser à un objet qui est éclairé,

- 2 **Spéculaire** : de manière semblable à un miroir. La partie de la lumière réfléchie rebondit sur la surface avec le même angle sous lequel elle l'a frappée. Dans ce cas, la brillance de la lumière réfléchie sur le matériau dépend de si vous vous situez ou non dans la ligne dans laquelle elle a été réfléchie. C'est à-dire qu'elle ne dépend pas seulement de l'angle sous lequel la lumière frappe la surface, mais aussi de l'angle entre votre ligne de mire et la surface.

Cette réflexion spéculaire est ce qui provoque les « reflets » ou la « surbrillance » sur les objets. L'intensité de la réflexion spéculaire peut évidemment varier d'un matériau à l'autre ; le bois brut aura probablement très peu de réflexion spéculaire alors que le métal fortement poli en aura beaucoup.

Le modèle de Phong ajoute une touche supplémentaire à ce système composé de quatre étapes, en considérant que toutes les lumières possèdent deux propriétés :

- 1 Les valeurs RVB de la lumière diffuse qu'elles produisent.
- 2 Les valeurs RVB de la lumière spéculaire qu'elles produisent.

Et que tous les matériaux en possèdent quatre :

- 1 Les valeurs RVB de la lumière ambiante qu'ils reflètent.
- 2 Les valeurs RVB de la lumière diffuse qu'ils reflètent.
- 3 Les valeurs RVB de la lumière spéculaire qu'ils reflètent.
- 4 La brillance de l'objet, qui détermine les détails de la réflexion spéculaire.

La couleur de chaque point de la scène est une combinaison de la couleur de la lumière qui l'éclaire, des couleurs du matériau et des effets de lumière. Donc pour définir complètement la lumière dans une scène selon le modèle de Phong, nous avons besoin de deux propriétés par lumière et quatre par point de la surface de notre objet.

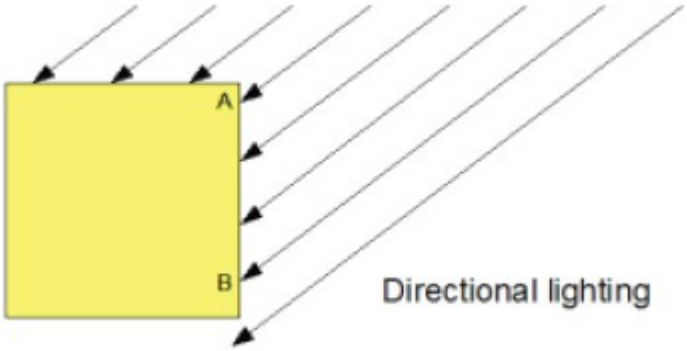
La lumière ambiante, de par sa nature même, n'est liée à aucune lumière particulière, mais nous avons aussi besoin d'un moyen de stocker son intensité globale dans la scène. Certaines fois il est plus facile de simplement spécifier un niveau ambiant pour chaque canal de source de lumière et les ajouter au sein d'un seul terme.

Quoi qu'il en soit, une fois que nous avons ces informations, nous pouvons travailler sur les couleurs associées aux réflexions ambiante, directionnelle et spéculaire de la lumière sur tous les points, puis les ajouter ensemble pour obtenir la couleur globale. Tout ce que notre shader doit faire est de calculer les contributions sur chaque vertex des couleurs rouge, vert et bleu des lumières ambiante, diffuse et spéculaire.

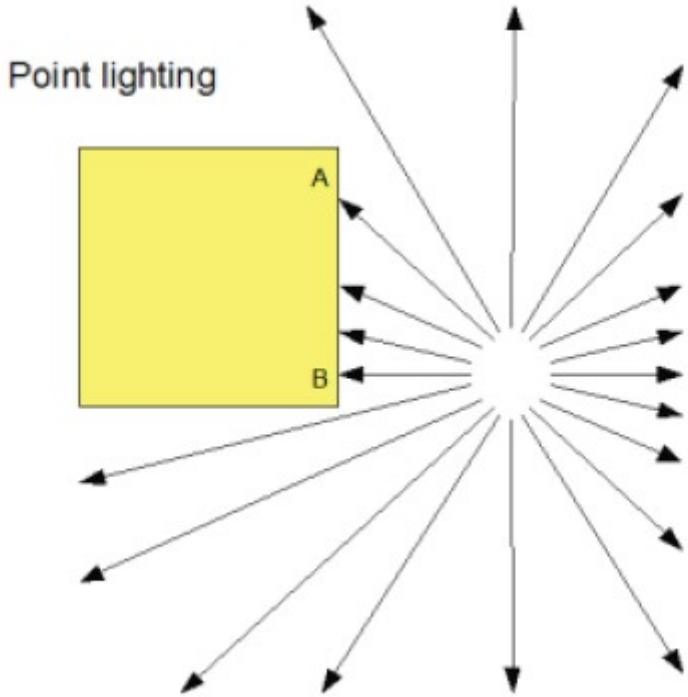
Puis il doit les utiliser pour pondérer les composantes RVB, les additionner ensemble et renvoyer le résultat. Maintenant, et par soucis de simplicité, nous allons commencer par considérer les lumières diffuses et ambiantes en ignorant la spéculaire. Nous allons supposer que les couleurs de la texture sont les valeurs des réflexions diffuse et ambiante.

Pour commencer, nous considèrerons seulement un type de lumière diffuse, **la lumière directionnelle.**
La lumière arrivant vers une surface selon une direction peut être de deux types

- Une simple lumière directionnelle qui est dans la même direction à travers toute la scène, comme une lumière provenant d'une source lumineuse éloignée.



- La lumière qui vient d'un seul endroit de la scène, une lumière ponctuelle donc, vue sous un angle différent dans différents endroits donc.



Pour l'éclairage directionnel simple, l'angle d'incidence de la lumière au niveau des vertex sur une face donnée - aux points A et B sur le schéma - est toujours le même. Pensez à la lumière du soleil : tous les rayons sont parallèles, et bien ici, c'est pareil.

Si, au contraire, la lumière arrive d'un point de la scène, l'angle de la lumière sera différent selon chaque vertex au point A dans ce second schéma, l'angle est d'environ 45 degrés, alors qu'au point B, il est d'environ 90 degrés par rapport à la surface.

Cela signifie que **pour l'éclairage ponctuel, nous devons travailler sur la direction d'où vient la lumière pour chaque vertex**, alors que pour un éclairage directionnel nous avons seulement besoin d'une valeur pour la source.Nous allons donc décrire le fonctionnement de la lumière la plus simple pour commencer.

L'éclairage directionnel

Nous savons que toutes les lumières de notre scène vont provenir d'une direction particulière, direction qui ne changera pas d'un vertex à l'autre. Nous pouvons donc la mettre dans une variable uniforme, à laquelle le shader peut accéder. Nous savons aussi que l'effet de la lumière sur chaque vertex sera déterminé par l'angle formé avec la surface de notre objet en ce vertex, nous avons donc besoin de représenter l'orientation de la surface d'une quelconque manière.

La meilleure façon de faire ceci en 3D consiste à spécifier **le vecteur normal** à la surface au niveau du vertex, ce qui nous permet de caractériser la direction face à la surface par trois nombres. Une fois que nous avons la normale, il reste un dernier élément nécessaire avant de pouvoir écrire notre shader.

Étant donné le vecteur normal d'une surface en un vertex et le vecteur décrivant la direction de provenance de la lumière, nous devons travailler sur la quantité de **lumière réfléchie** par la surface de façon **diffuse**. Elle s'avère être **proportionnelle au cosinus de l'angle entre ces deux vecteurs**.

Si la normale est de 0 degré (autrement dit, la lumière frappe pleinement la surface, à 90 degrés par rapport à la surface dans toutes les directions), alors nous pouvons dire qu'elle reflète toute la lumière. Si l'angle de la lumière à la normale est de 90 degrés, rien n'est réfléchi.

Tout ce qui se trouve entre suit la courbe du cosinus. (Si l'angle est supérieur à 90 degrés, alors nous obtiendrions en théorie des quantités négatives de la lumière réfléchie, nous utilisons la plus grande des valeurs entre le cosinus et zéro.)

Calculer le cosinus de l'angle entre deux vecteurs est assez aisé, si les deux vecteurs sont normalisés, on calcule leur **produit scalaire** et nous aurons la valeur recherchée. En GLSL, vous pouvez calculer directement un produit scalaire à l'aide de la fonction *dot,* à laquelle vous devez envoyer 2 vecteurs. Maintenant que nous avons étudié la théorie, je vous invite à ouvrir l'exercice "exo7" et à analyser le code.

ANNEXES

Les matrices:

Nous avons brièvement vu ce qu'étaient les matrices, nous allons donc maintenant détailler un peu plus la façon dont on s'ee sert. Comment les manipuler ? Comment se comportent-elles ? Comment les combiner ? En webGL, nous allons essentiellement travailler avec les 3 transformations suivantes (mais d'autres sont possibles avec certains types de matrices):

- La translation : consiste à faire "glisser" un ensemble de points sur un "rail" (un vecteur).
- La rotation : consiste à faire pivoter un ensemble de points d'un angle Thêta par rapport à un point.
- L'homothétie : consiste simplement à agrandir ou à réduire une forme géométrique.

Une transformation est donc grosso-modo une modification apportée à un ensemble de points ou repère dans un espace donné (soit 2D, soit 3D, etc ...).

Ces transformations vont nous servir à placer tous nos objets (personnages, arbres, maisons, ...) dans notre monde 3D. A chaque fois que nous voudrons placer un objet nous utiliserons les transformations. Elles seront essentiellement utilisées sur le repère.

Mais qu'est-ce qu'une matrice? C'est un tableau de nombres ordonnés en lignes et en colonnes entourés par des parenthèses. Sa syntaxe est semblable à celle d'un vecteur mais avec plus de nombres :

$$\begin{pmatrix} 7 & 3 & 10 \\ 6 & 0 & 6 \\ 4 & 8 & 12 \end{pmatrix}$$

Une matrice n'est pas forcément un tableau de 9 cases, elle peut en contenir jusqu'à l'infini. Cependant ce n'est pas qu'un simple tableau, c'est une sorte de super-vecteur qui permet de faire pas mal de choses intéressantes. Un vecteur est en général utilisé en 3D pour gérer les points, les directions, les normales... Les matrices permettent de faire bien plus que ça. Elles servent principalement à convertir des données géométriques en données numériques. Commençons donc par introduire certains types de matrices.

Matrice carrée

Une matrice carrée est une matrice ayant le même nombres de colonnes et de lignes :

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

matrice carrée 3x3

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

matrice carrée 4x4

Nous pouvons voir ici deux exemples de matrices carrées, une matrice carrée 3x3 et une matrice carrée 4x4. Avec webGL, nous n'utiliserons que les matrices carrées de ce type. Bien entendu il est tout à fait possible d'avoir des matrices carrées de type 1x1, 2x2, 5x5 etc etc ...

Matrice d'identité

Il s'agit de la matrice la plus simple qu'il soit. Elle a la particularité d'avoir toutes ses valeurs égales à 0 hormis les valeurs de sa diagonale qui, elles, sont égales à 1, reprenons nos matrices vues plus haut:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

matrice identité carrée 3x3

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

matrice identité carrée 4x4

Comme vous pouvez le voir, ces matrices correspondent pile poil à la définition de la matrice identité.
La caractéristique principale de la matrice identité, hormis son aspect, est qu'elle correspondant à une **transformation nulle,** ce qui veut dire que si nous multiplions un vecteur par une telle matrice, le vecteur ne sera absolument pas changé.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 5 \\ 4 \end{pmatrix} = \begin{pmatrix} 3 \\ 5 \\ 4 \end{pmatrix}$$

multiplication d'un vecteur à 3 dimensions par une matrice 3x3

Nous allons étudier la multiplication matricielle dans un instant. Mais vous pouvez déjà retenir ce principe, **si un vecteur, ou une matrice, est multiplié(e) par une matrice d'identité, alors il/elle ne sera pas modifié(e).**

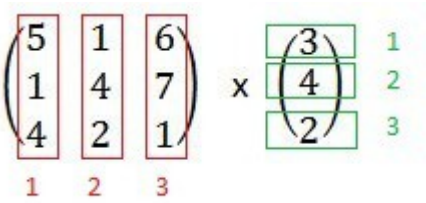
Multiplication d'un vecteur par une matrice

Partie 1 : la vérification

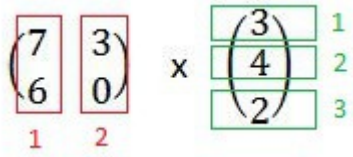
Dans ce premier exemple, je vais prendre une matrice carrée d'ordre 3 pour simplifier les choses. Sachez cependant que le principe est le même quelle que soit la taille de la matrice.

$$\begin{pmatrix} 5 & 1 & 6 \\ 1 & 4 & 7 \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} 31 \\ 33 \\ 22 \end{pmatrix}$$

Voyons comment arriver à ce résultat.
Premièrement, il faut vérifier que le nombre de colonnes de la matrice soit égal au nombre de coordonnées du vecteur. Si ce n'est pas le cas alors la multiplication est impossible.



La matrice possède 3 colonnes et le vecteur possède 3 coordonnées. La multiplication est donc possible.
Attention ! La matrice sera toujours à GAUCHE de la multiplication et le vecteur sera toujours à DROITE !
Prenons un autre exemple :



La matrice possède 2 colonnes et le vecteur 3 coordonnées. La multiplication est impossible.
Vous avez compris le principe ? Bien, passons à la suite.

Partie 2 : La multiplication

La multiplication d'une matrice et d'un vecteur s'effectue comme ceci : pour une seule coordonnée (x, y, ou z) du vecteur résultat, nous allons faire la somme des multiplications de chaque nombre d'une ligne de la matrice par chaque nombre correspondant du vecteur.

Commençons déjà par la première ligne :

$$\begin{pmatrix} 5 & 1 & 6 \\ \square & \square & \square \\ \square & \square & \square \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} (5*3)+(1*4)+(6*2) \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 15+4+12 \\ \square \\ \square \end{pmatrix} = \begin{pmatrix} 31 \\ \square \\ \square \end{pmatrix}$$

Il faut multiplier les nombres de même couleur sur le schéma, puis additionner les différents résultats. Une ligne de la matrice ne donnera qu'une seule coordonnée du vecteur résultat, ici la coordonnée x.

Passons à la deuxième ligne :

$$\begin{pmatrix} \square & \square & \square \\ 1 & 4 & 7 \\ \square & \square & \square \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} \square \\ (1*3)+(4*4)+(7*2) \\ \square \end{pmatrix} = \begin{pmatrix} \square \\ 3+16+14 \\ \square \end{pmatrix} = \begin{pmatrix} \square \\ 33 \\ \square \end{pmatrix}$$

La première ligne nous a donné la première coordonnée, la deuxième ligne nous donne la deuxième coordonnée. Nous multiplions chaque ligne de la matrice par toutes les valeurs du vecteur.

Passons à la troisième ligne :

$$\begin{pmatrix} \square & \square & \square \\ \square & \square & \square \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} \square \\ \square \\ (4*3)+(2*4)+(1*2) \end{pmatrix} = \begin{pmatrix} \square \\ \square \\ 12+8+2 \end{pmatrix} = \begin{pmatrix} \square \\ \square \\ 22 \end{pmatrix}$$

La troisième ligne de la matrice nous donne la troisième coordonnée. Si la matrice avait eu 4 lignes (et donc 4 colonnes) alors il y aurait eu une quatrième opération du même type pour la quatrième coordonnée.

Récapitulons:

$$\begin{pmatrix} 5 & 1 & 6 \\ 1 & 4 & 7 \\ 4 & 2 & 1 \end{pmatrix} \times \begin{pmatrix} 3 \\ 4 \\ 2 \end{pmatrix} = \begin{pmatrix} (5*3)+(1*4)+(6*2) \\ (1*3)+(4*4)+(7*2) \\ (4*3)+(2*4)+(1*2) \end{pmatrix} = \begin{pmatrix} 15+4+12 \\ 3+16+14 \\ 12+8+2 \end{pmatrix} = \begin{pmatrix} 31 \\ 33 \\ 22 \end{pmatrix}$$

Le principe est le même, et ce quelle que soit la taille de la matrice. Maintenant que nous savons mutliplier un vecteur et une matrice, nous allons nous attaquer à la multiplication de 2 matrices entre elles.

Multiplication de deux matrices

La multiplication de deux matrices peut sembler plus compliquée à première vue mais si vous avez compris ce que l'on a fait avant, alors vous savez déjà multiplier deux matrices.

Partie 1

Avant tout, pour pouvoir multiplier deux matrices carrées entre-elles il faut absolument que les deux matrices soient de la même taille (donc du même ordre), c'est-à-dire qu'elles doivent avoir le même nombre de lignes et le même nombre de colonnes.

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix}$$

Les deux matrices ont la même taille, on peut donc les multiplier, prenons un autre exemple :

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \times \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

Ici, les deux matrices n'ont pas la même taille nous ne pouvons pas les multiplier. Pour pouvoir multiplier deux matrices carrées, nous allons couper la deuxième matrice en 3 vecteurs (ou 4 selon la taille), puis nous appliquerons la multiplication que nous avons vue à l'instant pour chacun de ces vecteurs.

Prenons deux matrices cobayes :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix}$$

Attention, l'ordre des matrices est important ! M1 x M2 ne sera pas forcément égal à M2 x M1. Faites attention à l'ordre de votre multiplication !
Bien, coupons la seconde matrice en 3 vecteurs :

$$\begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix} \Rightarrow \begin{pmatrix} 5 \\ 3 \\ 1 \end{pmatrix} \quad \begin{pmatrix} 4 \\ 0 \\ 2 \end{pmatrix} \quad \begin{pmatrix} 3 \\ 1 \\ 5 \end{pmatrix}$$

Maintenant il nous suffit d'appliquer la multiplication d'une matrice et d'un vecteur pour chaque vecteur que nous venons de créer. Voici ce que ça donne pour le premier :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 5 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} (3*5)+(7*3)+(2*1) \\ (1*5)+(5*3)+(7*1) \\ (4*5)+(2*3)+(8*1) \end{pmatrix} = \begin{pmatrix} 15+21+2 \\ 5+15+7 \\ 20+6+8 \end{pmatrix} = \begin{pmatrix} 38 \\ 27 \\ 34 \end{pmatrix}$$

Au tour du deuxième :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 4 \\ 0 \\ 2 \end{pmatrix} = \begin{pmatrix} (3*4)+(7*0)+(2*2) \\ (1*4)+(5*0)+(7*2) \\ (4*4)+(2*0)+(8*2) \end{pmatrix} = \begin{pmatrix} 12+0+4 \\ 4+0+14 \\ 16+0+16 \end{pmatrix} = \begin{pmatrix} 16 \\ 18 \\ 32 \end{pmatrix}$$

On fait la même chose avec le dernier vecteur :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 3 \\ 1 \\ 5 \end{pmatrix} = \begin{pmatrix} (3*3)+(7*1)+(2*5) \\ (1*3)+(5*1)+(7*5) \\ (4*3)+(2*1)+(8*5) \end{pmatrix} = \begin{pmatrix} 9+7+10 \\ 3+5+35 \\ 12+2+40 \end{pmatrix} = \begin{pmatrix} 26 \\ 43 \\ 54 \end{pmatrix}$$

Nous obtenons au final 3 vecteurs fraichement calculés. Il nous suffit ensuite de les assembler **dans l'ordre de la division de la matrice** !

$$\begin{pmatrix} 38 \\ 27 \\ 34 \end{pmatrix} \quad \begin{pmatrix} 16 \\ 18 \\ 32 \end{pmatrix} \quad \begin{pmatrix} 26 \\ 43 \\ 54 \end{pmatrix} \Rightarrow \begin{pmatrix} 38 & 16 & 26 \\ 27 & 18 & 43 \\ 34 & 32 & 54 \end{pmatrix}$$

Au final, nous obtenons :

$$\begin{pmatrix} 3 & 7 & 2 \\ 1 & 5 & 7 \\ 4 & 2 & 8 \end{pmatrix} \times \begin{pmatrix} 5 & 4 & 3 \\ 3 & 0 & 1 \\ 1 & 2 & 5 \end{pmatrix} = \begin{pmatrix} 38 & 16 & 26 \\ 27 & 18 & 43 \\ 34 & 32 & 54 \end{pmatrix}$$

Rappel: l'ordre de la multiplication compte, M1 x M2 est différent de M2 x M1.

Évidemment, la taille de la matrice ne change toujours pas le principe. Nous maintenant faire passer les transformations de la géométrie à l'algèbre (les nombres). Nous allons maintenant voir comment utiliser des matrices pour appliquer des transformations.

La matrice de translation

La translation permet de déplacer un ensemble de points ou une forme dans un espace donné. En gros, on prend un vecteur qui servira de "rail" puis on fera glisser nos points sur ce rail. La forme finale ne sera pas modifiée, elle sera juste déplacée.

Une translation en 3 dimensions se traduit par la matrice suivante :

$$\begin{pmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ou X, Y et Z représentent la translation sur les axes x,y,z, vous remarquerez que cette matrice ressemble beaucoup à la matrice d'identité, il n'y a que les coordonnées du vecteur en plus.

La matrice d'homothétie

Une homothétie permet d'agrandir ou de réduire une forme géométrique, voici la matrice correspondante :

$$\begin{pmatrix} X & 0 & 0 & 0 \\ 0 & Y & 0 & 0 \\ 0 & 0 & Z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Ou X, Y, Z représentent les facteurs mutliplicatifs des axes x,y et z.

La matrice de rotation:

Il s'agit de la matrice la plus compliquée des 3, il est inutile de la retenir, elle est trop complexe nous la verrons jamais directement :

$$\begin{pmatrix} xx(1-\cos(\theta)) + \cos(\theta) & xy(1-\cos(\theta)) - z\sin(\theta) & xz(1-\cos(\theta)) + y\sin(\theta) & 0 \\ xy(1-\cos(\theta)) + z\sin(\theta) & yy(1-\cos(\theta)) + \cos(\theta) & yz(1-\cos(\theta)) - x\sin(\theta) & 0 \\ xz(1-\cos(\theta)) - y\sin(\theta) & yz(1-\cos(\theta)) + x\sin(\theta) & zz(1-\cos(\theta)) + \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dans notre cas, nous utiliserons la méthode correspondante de la librairie glMatrix, sachez juste que theta représente l'angle **en radians** et les coordonnées x,y,z **le vecteur correspondant à l'axe de rotation**.

Nous n'allons pas aller plus loin dans notre étude mathématique des matrices, nous disposons déjà de suffisamment de connaissances.

Le mélange des couleurs lors de l'alpha blending

Exemple:

Si vous utilisez SRC_COLOR pour spécifier la couleur de la source, vous vous retrouvez avec des facteurs de source distincts pour le rouge, le vert, le bleu et l'alpha, dont chacun est égal aux composantes RGBA originales. Imaginons maintenant que WebGL essaye de calculer les couleurs du fragment ayant des valeurs RGBA (Rd', Gd', Bd', Ad) à partir des valeurs (Rs', Gs', Bs', As) du fragment source.

Posons de plus les facteurs RGBA source (Sr', Sg', Sb', Sa) et destination (Dr', Dg', Db', Da), WebGL calculera chaque composante couleur comme suit :

- Rresult = Rs * Sr + Rd * Dr
- Gresult = Gs * Sg + Gd * Dg
- Bresult = Bs * Sb + Bd * Db
- Aresult = As * Sa + Ad * Da

Dans notre cas donc, nous avons (juste pour la composante rouge afin de garder les choses simples) :

- **Rresullt = Rs * As + Rd**