WebGL

Programmation Graphique

Abdelkader Gouaïch

2023/2024

Introduction

WebGL est une suite de spécifications développée par le consortium Web3D, permettant l'interaction avec les cartes graphiques 3D. Ces spécifications, qui sont un sous-ensemble des spécifications d'OpenGL et plus précisément de OpenGL ES (Embedded Systems), visent à faciliter l'interaction avec les cartes graphiques (GPU) dans des environnements contraints tels que les appareils mobiles ou les navigateurs web.

L'objectif principal de WebGL est d'autoriser un programme JavaScript, hébergé sur une page web par exemple, à interagir directement avec la carte graphique de l'ordinateur, et ainsi, à dessiner dans une zone dédiée de la fenêtre du navigateur.

Il est important de noter que, tout comme OpenGL, WebGL fournit une API permettant de créer des images 2D composées de pixels colorés à partir de données issues potentiellement d'un modèle géométrique 3D. En tant que programmeurs, notre tâche est précisément de fournir les fonctions permettant de projeter un modèle géométrique (3D ou 2D) sur l'image finale qui sera inévitablement en 2D, avec un système de coordonnées borné allant de -1 à 1 pour les deux axes.

Cette cartographie est réalisée à l'aide de fonctions écrites dans un langage spécialisé de *shader*: le GLSL. Il existe deux types de shaders :

- Le vertex shader, qui gère la transformation du modèle géométrique source (3D ou 2D) vers les coordonnées de l'image générée, située dans l'espace $[-1, 1] \times [-1, +1]$.
- Le fragment shader, qui s'occupe de la coloration des fragments générés par le vertex shader. Il attribue à chaque vertex une couleur, élément de $[0, 1]^4$, représentant les trois couleurs primaires (rouge, vert, bleu) et une composante de transparence. De plus, les pixels situés dans le fragment délimité par les vertex seront colorés en utilisant une interpolation, c'est-à-dire des couleurs intermédiaires entre les couleurs fixées des vertex.

La Machine Virtuelle WebGL

WebGL définit une machine virtuelle comportant plusieurs composants ainsi que d'un état défini par des paramètres. Pour programmer, nous devons comprendre son architecture et son fonctionnement car contrairement à une API purement logicielle, certaines opérations nécessitent de réaliser une séquence d'appels qui modifient l'état de la machine. Les appels des fonctions de l'API ne sont pas donc pas indépendants et l'ordre des appels devient important.

La machine virtuelle est organisée avec les sections suivantes :

- Une section globale de variables et de paramètres.
- Une section pour stocker les informations nécessaires lors de la récupération des données d'un buffer mémoire vers les attributs utilisables par les programmes.
- Une section pour ranger les données relatives aux opérations d'effacement du buffer de rendu.
- Une section pour ranger les données des textures utilisées pour colorer les pixels des fragments géométriques.

Dans la suite, nous allons présenter uniquement les sections utiles pour dessiner des formes géométriques simples. Il s'agit de la section des paramètres globaux; l'effacement du buffer de rendu; et la section pour stocker les informations sur l'opération de récupération des attributs à partir d'un buffer mémoire.

Nous traiterons la section consacrée aux textures dans un prochain chapitre.

Les paramètres globaux :

Cette section regroupe les paramètres globaux suivants :

- `VIEWPORT` : ce paramètre définit les coordonnées en pixels de la surface allouée pour le dessin de l'image générée. Cette surface peut représenter qu'une sous-partie du canvas. Nous pouvons ainsi avoir plusieurs dessins dans un même canvas (une minimap par exemple)
- `ARRAY_BUFFER_BINDING` : ce paramètre fait référence à un buffer mémoire fréquemment utilisé comme source de données.
- `FRAMEBUFFER_BINDING` : ce paramètre fait référence à un tampon d'image hors écran ou utilisable ultérieurement comme texture.
- `CURRENT_PROGRAM` : ce paramètre fait référence au programme de shaders à utiliser.
- `VERTEX_ARRAY_BINDING` : ce paramètre fait référence à un tableau qui spécifie comment itérer sur une source de données pour générer les valeurs des attributs des vertex.
- `RENDERBUFFER_BINDING` : ce paramètre fait référence à une cible pour le rendu de l'image. Ces tampons ne peuvent pas être utilisés comme des textures, contrairement aux framebuffers.

Nous illustrerons l'utilisation de ces variables à travers des exemples.

VIEWPORT

VIEWPORT est utilisé pour spécifier la région du canvas où le rendu sera effectué. Cette région est définie par quatre valeurs : la position x, la position y, la largeur et la hauteur.

Voici un code illustrant une situation où nous souhaitons dessiner uniquement sur le quart inférieur du canvas :

```
// Obtenez le contexte de rendu WebGL
var gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
// Définissez la zone de la fenêtre d'affichage
var x = 0;
var y = 0;
var width = canvas.width / 2; // dessin uniquement sur le quart inférieure du canvas
var height = canvas.height / 2;
// Appliquez la zone de la fenêtre d'affichage
// la variable globale VIEWPORT de la machine WebGL va récupérer ces quatres valeurs sous la
gl.viewport(x, y, width, height);
```

ARRAY_BUFFER et ARRAY_BUFFER_BINDING

La machine WebGL nous permet de réserver de la mémoire dans l'espace GPU. La création d'un buffer de mémoire se fait avec la fonction `gl.createBuffer()` qui renvoie une référence vers le buffer créé.

Les opérations de WebGL ne manipulent pas directement les références des buffers créés par le programmeur. Elles consultent une variable prédéfinie appelée `ARRAY_BUFFER` (dans le jargon WebGL, nous parlons d'un slot) pour savoir sur quel buffer effectuer les opérations.

Il incombe donc au programmeur de lier (bind) le bon buffer mémoire avec le slot `ARRAY_BUFFER`. Cela se fait en utilisant l'opération `gl.bindBuffer`, dans laquelle nous spécifions que notre cible est le slot `ARRAY_BUFFER` et nous fournissons ensuite la référence vers notre buffer.

Le résultat de `gl.bindBuffer(ARRAY_BUFFER,mon_buffer)` est que le slot `ARRAY_BUFFER` pointe sur `mon_buffer`. Cela modifie l'état ou la configuration de la machine WebGL. Pour consulter le nouvel état et savoir quel est le buffer actuellement utilisé, nous pouvons consulter le paramètre global `ARRAY_BUFFER_BINDING`.

Voici un code qui illustre l'utilisation et vérifie que la valeur du paramètre est correcte:

```
// Obtenez le contexte de rendu WebGL
var gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
// Créez un nouveau tampon
var buffer = gl.createBuffer();
// Liez le tampon à la cible ARRAY_BUFFER
gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
// Maintenant, le tampon est lié à la cible ARRAY_BUFFER
// Cela se fait par la variable ARRAY_BUFFER_BINDING

// Obtenez une référence au tampon actuellement lié à ARRAY_BUFFER
var currentBufferBinding = gl.getParameter(gl.ARRAY_BUFFER_BINDING);

// Vérification que nous parlons du même buffer
console.log(currentBufferBinding === buffer); // Affichera true
```

FRAMEBUFFER et FRAMEBUFFER_BINDING

Le paramètre `FRAMEBUFFER_BINDING` dans WebGL est utilisé pour obtenir une référence au *tampon de trame* (framebuffer) actuellement lié.

Un tampon de trame est un objet qui contient une image utilisable comme texture de rendu.

Dans WebGL, nous pouvons aussi utiliser les tampons de trame pour réaliser un rendu hors écran, ce qui signifie que le rendu est effectué dans un tampon mémoire et non sur un canevas de dessin. Nous pouvons utiliser cela pour des rendus complexes nécessitant plusieurs itérations avant affichage; réaliser des rendus de contenu textuel sous format image ou créer à la volée des images et des textures par programmation.

L'utilisation du `FRAMEBUFFER` est similaire à celle du `ARRAY_BUFFER`. Nous devons modifier l'état de la machine WebGL avec la fonction `gl.bindFramebuffer`, en précisant le slot `FRAMEBUFFER` et la référence de notre framebuffer. L'état de WebGL est alors modifié et le paramètre `FRAMEBUFFER_BINDING` nous renseigne sur la cible pointée par le slot `FRAMEBUFFER`.

Voici un exemple simple d'utilisation :

```
// Obtenez le contexte de rendu WebGL
var gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
// Créez un nouveau tampon de trame
var framebuffer = gl.createFramebuffer();
// Liez le tampon de trame
gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
// Maintenant, le tampon de trame est lié
// Obtenez une référence au tampon de trame actuellement lié
```

```
var currentFramebufferBinding = gl.getParameter(gl.FRAMEBUFFER_BINDING);
console.log(currentFramebufferBinding === framebuffer); // Affichera true
```

L'exemple présenté dans `session3/etape1/framebuffer.html` illustre un cas plus utile d'utilisation d'un tampon de trame. Dans cet exemple, nous allons récupérer un texte écrit dans un canvas pour le transformer et l'utiliser comme une texture qui sera par la suite affichée sur la page. Dans cet exemple, pour pouvoir générer l'image à partir de la texture, nous avons créé un framebuffer sur lequel nous avons effectué des opérations en le pointant avec le slot `FRAMEBUFFER`.

`CURRENT_PROGRAM`

Ce paramètre référence le programme des shaders actuellement utilisé. Un programme shader dans WebGL est un programme exécutable créé en liant un shader vertex et un shader fragment. Il est utilisé pour spécifier comment les données de vertex et de fragment doivent être traitées pour générer l'image finale. Nous pouvons préparer plusieurs programmes shaders lors de la phase d'initialisation et, lors de la phase de rendu, choisir le programme à utiliser avec la fonction `gl.useProgram(program)`.

Cette fonction modifie l'état de la machine WebGL et le paramètre `CURRENT_PROGRAM` nous permet de récupérer le programme actuellement utilisé avec `gl.getParameter(gl.CURRENT_PROGRAM)`.

Voici un programme qui illustre l'utilisation de ces fonctions :

```
// Obtenez le contexte de rendu WebGL
var gl = canvas.getContext("webgl") || canvas.getContext("experimental-webgl");
// Créez des shaders et un programme
var vertShader = gl.createShader(gl.VERTEX_SHADER);
var fragShader = gl.createShader(gl.FRAGMENT_SHADER);
var program = gl.createProgram();
gl.attachShader(program, vertShader);
gl.attachShader(program, fragShader);
gl.linkProgram(program);
// Utilisez le programme
gl.useProgram(program);
// Obtenez une référence au programme shader actuellement en cours d'utilisation
var currentProgram = gl.getParameter(gl.CURRENT_PROGRAM);
console.log(currentProgram === program); // Affichera true
```

`VERTEX_ARRAY_BINDING`

Le paramètre `VERTEX_ARRAY_BINDING` dans WebGL permet de récupérer la référence au Vertex Array Object (VAO) actuellement lié.

Un Vertex Array Object est un objet qui stocke les informations de configuration pour calculer la valeur des attributs à partir des données provenant d'un tampon mémoire. C'est une fonctionnalité de WebGL 2.0, qui n'est pas disponible dans WebGL 1.0.

Fonctionnement du VAO Pour mieux comprendre le rôle du VAO, prenons un exemple.

Le VAO est un tableau où chaque ligne correspond à un attribut du programme. Il est donc important de récupérer la localisation d'un attribut dans le programme, c'est-à-dire son index dans la séquence de tous les attributs du programme, et utiliser cette même localisation comme index dans le tableau VAO.

Pour un attribut particulier, nous devons l'activer ou le désactiver. Par défaut, toutes les lignes sont désactivées et nous devons activer une ligne avant de l'utiliser.

Ensuite, nous renseignons des informations pour lier le tampon mémoire qui sera la source des données et les attributs de chaque vertex. Par exemple, si un attribut est de type vecteur composé de 2 flottants pour une position 2D, alors nous indiquerons dans le VAO qu'il faut lire 2 cases dans la mémoire tampon pour cet attribut et cela pour chaque vertex. Il est aussi possible de commencer la lecture des données avec un décalage (offset) et de réaliser des sauts entre les valeurs. Nous pouvons aussi demander une normalisation du vecteur des valeurs pour obtenir des valeurs entre $[0, 1]$.

Voici un extrait de code tiré du fichier `vertexarray.html` :

```
var coordLoc = gl.getAttribLocation(shaderProgram, "

coordinates");

gl.bindBuffer(gl.ARRAY_BUFFER, vertexBuffer);
gl.enableVertexAttribArray(coordLoc);
gl.vertexAttribPointer(coordLoc, 2, gl.FLOAT, false, 0, 0);
```

Dans ce code, nous avons récupéré la localisation (l'index) de l'attribut "coordinates" déclaré dans notre programme shader. Nous avons configuré l'état de WebGL pour que `ARRAY_BUFFER` pointe sur un tampon mémoire `vertexBuffer` qui contient un tableau de flottants où les coordonnées 2D de nos vertex sont stockées sous la forme $[x_0, y_0, x_1, y_1, ...]$. Nous activons la ligne du VAO qui correspond à notre attribut en utilisant sa localisation. Cette ligne devient active pour la suite des opérations.

Pour finir, nous donnons les informations sur comment importer les données de `vertexBuffer` pour créer les attributs de chaque vertex :

- Nous lirons deux valeurs du tableau x et y.
- Le type de chaque valeur est `gl.FLOAT`.

- Nous ne souhaitons pas de normalisation.
- Nous souhaitons commencer la lecture au début du tableau (offset = 0).
- Nous ne souhaitons pas de décalage entre chaque traitement pour un vertex.

Utilisation de VAO Au démarrage, WebGL dispose d'un VAO par défaut, mais nous pouvons en créer un nouveau avec la fonction `gl.createVertexArray()`. La liaison avec un VAO se fait avec la fonction `gl.bindVertexArray`. Pour finir, le paramètre `gl.VERTEX_ARRAY_BINDING` permet de récupérer le VAO actuellement utilisé avec `gl.getParameter(gl.VERTEX_ARRAY_BINDING)`.

Voici un exemple simple montrant comment ces différentes fonctions sont utilisées :

```
// Obtenez le contexte de rendu WebGL 2
var gl = canvas.getContext("webgl2");
// Créez un nouveau Vertex Array Object
var vao = gl.createVertexArray();
// Liez le Vertex Array Object
gl.bindVertexArray(vao);
// Maintenant, le Vertex Array Object est lié
// Obtenez une référence au Vertex Array Object actuellement lié
var currentVertexArrayBinding = gl.getParameter(gl.VERTEX_ARRAY_BINDING);
// currentVertexArrayBinding est maintenant une référence au Vertex Array Object
// Vous pouvez vérifier l'égalité des références pour confirmer que le VAO est bien lié
console.log(currentVertexArrayBinding === vao); // Affichera true
```

Transmission des Données vers le Programme Shader WebGL

Il est impératif de rappeler que le programme Shader s'exécute sur la GPU et qu'il doit disposer des données dans la mémoire de la GPU. Nous sommes responsables de transmettre explicitement les données de l'espace mémoire de la CPU (notre programme JavaScript) vers l'espace mémoire de la GPU. Cette section présente les divers moyens de réaliser ce transfert de données.

WebGL propose quatre moyens pour qu'un programme shader reçoive des données.

Attributs et Buffers

Les buffers sont des tableaux de données binaires que nous téléchargeons sur le GPU. Ils contiennent des éléments tels que les positions, les coordonnées de

texture, et les couleurs de sommet. Bien que ce soient les cas usuels, nous sommes libres d'y placer les données que nous estimons nécessaires pour nos besoins applicatifs.

Les attributs, déclarés au sein du programme shader (le vertex shader plus précisément), sont associés à chaque vertex. Nous sommes responsables de l'association entre les données du buffer et les attributs. Cette association est réalisée grâce au VAO (Vertex Array Object), qui informe WebGL sur la manière d'extraire les informations du buffer pour les attribuer à un attribut d'un vertex donné.

Par exemple, vous pourriez stocker des positions dans un buffer sous la forme de trois flottants de 32 bits par position. Vous indiqueriez à un attribut particulier quel buffer doit être utilisé pour extraire les positions, quel type de données il doit extraire (3 composants de nombres flottants de 32 bits), à quel décalage dans le buffer les positions commencent, et combien d'octets obtenir d'une position à la suivante.

Les buffers ne permettent pas un accès aléatoire. Au lieu de cela, un shader de sommet est exécuté un nombre spécifié de fois. Chaque fois qu'il est exécuté, la valeur suivante de chaque buffer spécifié est extraite et attribuée à un attribut.

Uniformes

Les uniformes sont simplement des variables globales que nous définissons avant l'exécution de notre programme shader.

Elles sont accessibles durant la phase de traitement de tous les vertex et leur valeur ne peut pas changer durant un cycle d'exécution du programme sur tous les vertex.

Exemple

Voici un exemple pour transmettre une couleur unie à un fragment shader via un uniforme.

Le fragment shader va déclarer un uniform de type vec4 nommé `u_color`:

```
precision mediump float;
uniform vec4 u_color;
void main(void) {
    gl_FragColor = u_color;
}
```

Pour renseigner la valeur de cet uniform nous devons simplement récupérer sa localisation (son index) et y copier les données.

```
var u_color = gl.getUniformLocation(program, "u_color");
gl.uniform4fv(u_color, new Float32Array([0.0, 0.0, 1.0, 1.0])); // copier des valeurs dans
```

Le code complet de cet exemple est disponible dans le fichier `code/exemples/exmpleUniform.html`

Textures

Les textures sont des tableaux de données auxquels vous pouvez accéder de manière aléatoire dans votre programme shader. De plus, elles permettent l'application de fonctions prédéfinies, appelées filtres, pour modifier les valeurs du tableau.

Il est courant de stocker une image dans une texture, mais les textures, n'étant que des conteneurs de données, peuvent tout aussi bien contenir autre chose que des couleurs.

Ce qui caractérise les textures c'est l'accès non séquentiel aux données via un échantillonnage. Ainsi, nous pouvons récupérer une donnée en utilisant un système de coordonnées spécifique appelé le repère UV défini dans $[0, 1] \times [0, 1]$.

Un autre aspect important concerne la possibilité d'utiliser des filtres prédéfinis pour modifier les valeurs de la texture. Nous pouvons par exemple utiliser des fonctions des filtres de minimisation et de maximisation sur des portions de la texture.

Exemple

Nous allons présenter un exemple où les couleurs d'un carré sont échantillonnées d'une texture que nous allons créer. Le code complet de l'exemple se trouve dans le fichier `code/exemples/exempleTexture.html`.

Voici le code des shaders utilisés :

```
<script id="vertex-shader" type="x-shader/x-vertex">
  attribute vec2 a_position;
  varying vec2 v_texCoord;
  void main() {
      gl_Position = vec4(a_position, 0, 1);
      v_texCoord = a_position * 0.5 + 0.5; // transformation linéraire de repère [-1,1]-> [0
  }
</script>
<script id="fragment-shader" type="x-shader/x-fragment">
  precision mediump float;
  varying vec2 v_texCoord;
  uniform sampler2D u_texture;
  void main() {
      gl_FragColor = texture2D(u_texture, vec2(v_texCoord.x, v_texCoord.y));
```

```
  }
</script>
```

Les deux shaders partagent un varying nommé `texCoords`. En plus d'être partagé un varying peut être interpoler pour les pixels. Le vertex shader récupère les coordonnées du vertex et copie ces coordonnées dans le varying qui représente les coordonnées dans la texture.

Le fragment shader utilise les coordonnées récupérées dans le varying et effectue la transformation suivante:

La transformation de l'intervalle $[-1, +1]^2$ du *clip space* vers $[0, 1]^2$ l'espace de texture est donnée par la fonction :

$$f(x, y) = (\frac{x+1}{2}, \frac{y+1}{2})$$

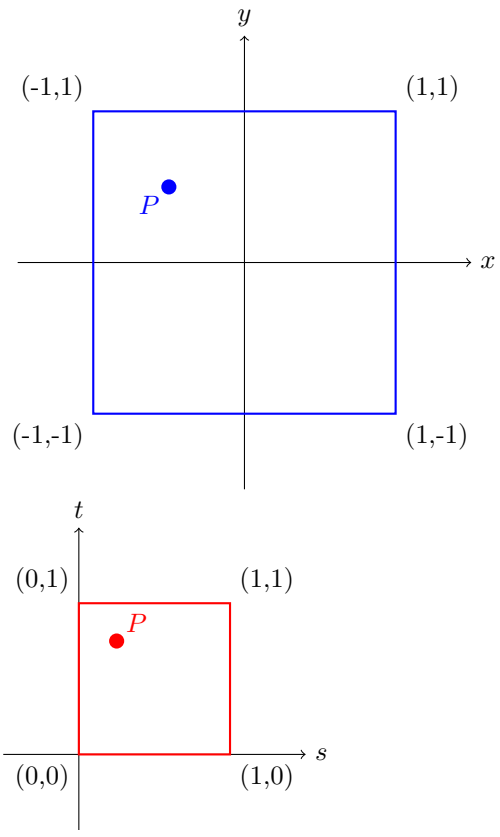

Figure 1: Transformation entre l'espace de dessin et l'espace de texture

Comme nous pouvons le voir avec la figure ci-dessus, cette transformation va

nous permettre de changer les coordonnées entre l'espace de dessin et celui des textures. Chaque point de l'espace de dessin va pouvoir récupérer une couleur de l'espace de texture qui lui correspond.

L'étape suivante est la création du buffer pour la texture et le téléchargement de ses données.

```
var texture = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, texture);
```

Ces deux lignes créent et bind le slot `TEXTURE_2D` vers la nouvelle texture.

Ensuite nous téléchargeons les données de la texture. Notre texture est très simple: elle se compose de 3 pixels de couleurs rouge, vert et bleu.

```
var textureData = new Uint8Array([
  255,
  0,
  0,
  255, // Red
  0,
  255,
  0,
  255, // Green
  0,
  0,
  255,
  255, // Blue
]);
gl.texImage2D(
  gl.TEXTURE_2D,
  0,
  gl.RGBA,
  3,
  1,
  0,
  gl.RGBA,
  gl.UNSIGNED_BYTE,
  textureData
);
```

Pour charger les données dans la GPU, nous devons renseigner:

- le niveau de détail, nous mettons 0 pour dire que c'est une texture de niveau 0 (le niveau le plus détaillé)
- le format est donné en composantes couleurs plus une transparence (RGBA)
- chaque donnée est de type octet non signé
- et pour finir nous renseignons la source des données

Nous allons également configurer certains paramètre de la texture.

```
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

Ces lignes gèrent les points suivants:

- Si la coordonnées de l'axe S dépasse la limite 1 alors nous récupérons la couleur du pixel de bord (donc avec s=1)
- Si la coordonnées de l'axe T dépasse la limite 1 alors nous récupérons la couleur du pixel de bord (donc avec t=1)
- Pour générer une texture *minifiée* nous appliquons un filtre `le plus proche`
- Pour générer une texture *maximisée* nous appliquons un filtre `le plus proche`

Pour finir, nous renseignons l'index de la texture (0) dans l'uniform `u_texture`.

```
var u_texture = gl.getUniformLocation(program, "u_texture");
gl.uniform1i(u_texture, 0);
```

Cet index sera utilisé par la fonction d'échantillonnage du fragment shader pour lire la couleur dans la bonne texture:

```
gl_FragColor = texture2D(u_texture, vec2(v_texCoord.x, v_texCoord.y));
```

Varyings

Les varyings fournissent un moyen pour un vertex shader de transmettre des données à un fragment shader. En plus d'être des variables partagées entre le vertex shader et le fragment shader, les varyings possèdent une propriété particulière : leurs valeurs sont interpolées pour un pixel qui se situe entre des vertex. Ainsi, selon ce qui est rendu - points, lignes ou triangles - les valeurs définies sur un varying par un vertex shader seront interpolées lors de l'exécution du shader de fragment sur les pixels.

Procédure de Rendu Géométrique avec WebGL

WebGL opère en fournissant deux services fondamentaux :

- Il permet le dessin en *2D* de formes géométriques (points, lignes, et triangles) à partir d'une séquence de sommets positionnés dans un système de coordonnées, nommé le *clip space*. Ce système est bidimensionnel avec une base orthogonale, et ses coordonnées sont *limitées* à l'intervalle $[-1, +1] \times [-1, +1]$. Le point $(0, 0)$ constitue le centre du clip space.

- Il est apte à attribuer une couleur à chaque pixel du *clip space*.

Il incombe au programmeur de spécifier les coordonnées des sommets dans le *clip space* en utilisant un *vertex shader*, et également de déterminer la couleur à attribuer à chaque pixel au moyen d'un *fragment shader*.

La fusion du vertex shader et du fragment shader forme un programme shader qui, une fois compilé et chargé dans le GPU, est exploitable par WebGL.

Bien que la procédure de rendu sur WebGL soit aisé à appréhender, sa mise en œuvre technique peut requérir plusieurs étapes.

Nous subdivisons ce processus en deux phases :

- **Phase d'Initialisation** : Elle englobe le chargement des données, la préparation des textures, la compilation et le chargement des programmes shader, la localisation des attributs, et le chargement des uniformes.
- **Phase de Rendu** : Elle consiste à sélectionner les sources de données initialisées ainsi que les programmes, avant de les exécuter pour générer des images.

Nous allons ensuite élaborer ces deux phases en illustrant le dessin d'un triangle avec WebGL.

Étude Pratique : Dessiner un Triangle

La Phase d'Initialisation

Indubitablement, un vertex shader est requis pour déterminer les coordonnées de chaque sommet du triangle. Un triangle ayant trois sommets, ce programme sera exécuté précisément trois fois. Pour chaque sommet, un attribut `a_position` est prévu, lequel est un vecteur bidimensionnel contenant les coordonnées x et y du sommet concerné.

WebGL attend la position du sommet dans la variable `gl_Position`. Dans ce programme, nous nous contentons de copier les valeurs de l'attribut 'a_position.'

```
// an attribute will receive data from a buffer
attribute vec2 a_position;
// all shaders have a main function
void main() {
  // gl_Position is a special variable a vertex shader
  // is responsible for setting
  gl_Position = vec4(a_position, 0, 1);
}
```

Nous avons également besoin d'un fragment shader pour attribuer une couleur à chaque pixel. Il est important de noter que le vertex shader s'exécute pour

tous les sommets, tandis que le fragment shader s'exécute sur tous les pixels de l'espace de dessin.

WebGL attend la couleur dans une variable spécifique `gl_FragColor`. Pour notre exemple, nous avons choisi d'attribuer une couleur constante à tous les pixels définissant la région ou le fragment de notre forme géométrique.

```
// nous devons préciser la précision de nos float
precision mediump float;

void main() {
  // gl_FragColor est la variable de résultat
  gl_FragColor = vec4(1, 0, 0.5, 1); // une couleur rouge
}
```

Il est nécessaire de compiler ces deux shaders afin de créer un programme shader. Voici le code d'une fonction générique permettant de compiler un shader selon son type, spécifié en paramètre.

```
function createShader(gl, type, source) {
  var shader = gl.createShader(type);
  gl.shaderSource(shader, source);
  gl.compileShader(shader);
  var success = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
  if (success) {
    return shader;
  }
  console.log(gl.getShaderInfoLog(shader));
  gl.deleteShader(shader);
}
```

Nous pouvons désormais utiliser cette fonction pour nos deux shaders et créer le programme, en supposant que nos shaders sont rédigés dans des balises `script` ayant comme identifiants "#vertex-shader-2d" et "#fragment-shader-2d" :

```
//définition d'une fonction de création de programme
function createProgram(gl, vertexShader, fragmentShader) {
  var program = gl.createProgram();
  gl.attachShader(program, vertexShader);
  gl.attachShader(program, fragmentShader);
  gl.linkProgram(program);
  var success = gl.getProgramParameter(program, gl.LINK_STATUS);
  if (success) {
    return program;
  }

  console.log(gl.getProgramInfoLog(program));
  gl.deleteProgram(program);
}
```

```
//récupération du code des shaders comme un string
var vertexShaderSource = document.querySelector("#vertex-shader-2d").text;
var fragmentShaderSource = document.querySelector("#fragment-shader-2d").text;
//utilisation de la fonction de création des shaders
var vertexShader = createShader(gl, gl.VERTEX_SHADER, vertexShaderSource);
var fragmentShader = createShader(gl, gl.FRAGMENT_SHADER, fragmentShaderSource);
//appel de la fonction de création de programme.
var program = createProgram(gl, vertexShader, fragmentShader);
```

Un attribut a été défini pour nos sommets. Pour pouvoir l'utiliser, il est nécessaire de mémoriser son index ou sa localisation. Cette information sera utile ultérieurement, car WebGL ne référence pas les attributs par leurs noms, mais uniquement par l'index qui leur a été attribué après la compilation du programme.

```
var positionAttributeLocation = gl.getAttribLocation(program, "a_position");
```

Nous pouvons à présent référencer l'attribut "a_position" par son index `positionAttributeLocation`.

Il convient maintenant de se pencher sur les valeurs de l'attribut "a_position." Ces valeurs seront extraites d'une mémoire tampon contenant des données que nous allons fournir.

Créons donc cette mémoire tampon :

```
var positionBuffer = gl.createBuffer();
```

Nous allons copier les valeurs, définies dans un tableau JavaScript (résidant dans la CPU), vers la mémoire tampon (résidant dans la GPU).

Nous établissons notre tableau JS de points pour le triangle. Il contient six valeurs, car nous avons trois sommets, et chaque sommet requiert deux coordonnées.

```
// three 2d points
var positions = [0, 0, 0, 0.5, 0.7, 0];
```

Ensuite, nous devons associer le slot ARRAY_BUFFER à notre mémoire tampon.

```
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
```

Pour conclure

, nous lançons l'opération de copie vers notre mémoire tampon.

```
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(positions), gl.STATIC_DRAW);
```

Le tableau des positions est un tableau JavaScript. Cependant, WebGL requiert des données strictement typées. Ainsi, la partie new Float32Array(positions)

crée un nouveau tableau de nombres à virgule flottante de 32 bits et copie les valeurs depuis positions.

`gl.bufferData` copie ensuite ces données dans le `positionBuffer` sur le GPU, car nous l'avons associé au point de liaison `ARRAY_BUFFER` précédemment.

Le dernier argument, `gl.STATIC_DRAW`, donne une indication à WebGL sur la manière dont nous utiliserons les données, permettant à WebGL d'optimiser certaines choses. `gl.STATIC_DRAW` indique que nous ne sommes pas susceptibles de modifier ces données souvent.

Le code jusqu'à ce point constitue du code d'initialisation, qui s'exécute une fois lorsque nous chargeons la page. Le code suivant est du code de rendu, qui s'exécute chaque fois que nous souhaitons rendre/dessiner.

La Phase de Rendu (Dessin)

Après la phase d'initialisation, nous entamons la phase de rendu proprement dite.

Voici les étapes détaillées qui interviennent dans ce processus. Certaines de ces opérations ont déjà été présentées pour la phase d'initialisation, comme la sélection des ARRAY_BUFFER et la configuration des attributs de sommet. Nous les présentons dans cette phase également, car il est souvent utile de modifier la configuration de WebGL pour les besoins spécifiques d'un cycle de rendu particulier.

1. **Activation du Programme Shader** :
   - Le programme shader est activé en utilisant la fonction `gl.useProgram(program);`.
   - Une fois activé, le programme shader restera en usage jusqu'à ce qu'un autre programme soit activé ou jusqu'à ce que le programme soit désactivé.
2. **Binding des Buffers** :
   - Les buffers de sommet sont associés en utilisant `gl.bindBuffer`.
   - Cela permet à WebGL de savoir quelles données utiliser pour le rendu.

```
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
```

3. **Configuration des Attributs de Sommet** :
   - Les attributs de sommet sont configurés en utilisant `gl.vertexAttribPointer`.
   - Cela spécifie comment les données sont lues du buffer de sommet et passées aux attributs de sommet dans le vertex shader.

```
var positionLoc = gl.getAttribLocation(program, "a_position");
gl.vertexAttribPointer(positionLoc, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(positionLoc);
```

4. **Configuration des Uniformes** :

- Les valeurs des uniformes sont configurées en utilisant les fonctions `gl.uniform*` (par exemple, `gl.uniform1i`, `gl.uniformMatrix4fv`, etc.).
- Les uniformes sont des variables qui restent constantes pour tous les sommets et fragments durant le rendu.

```
var uMatrix = gl.getUniformLocation(program, "uMatrix");
var matrix = new Float32Array([
  /*...valeurs de la matrice...*/
]);
gl.uniformMatrix4fv(uMatrix, false, matrix);
```

5. **Binding des Textures** :

   - Si des textures sont utilisées, elles sont associées en utilisant `gl.bindTexture`.
   - Les textures sont ensuite accessibles dans les shaders via des samplers.

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture);
var uSampler = gl.getUniformLocation(program, "uSampler");
gl.uniform1i(uSampler, 0);
```

6. **Rendu des Primitives** :

   - Le rendu est effectué en utilisant soit `gl.drawArrays` pour les données de sommet sans index, soit `gl.drawElements` pour les données indexées.
   - Ces appels déclenchent le pipeline de rendu, exécutant d'abord le vertex shader sur chaque sommet, puis le fragment shader sur chaque fragment généré durant la rastérisation.

```
gl.drawArrays(gl.TRIANGLES, 0, 3);
```

7. **Tests et Blending** :

   - Des tests tels que le test de profondeur (Z-test) et le test d'occultation peuvent être effectués pour déterminer si un fragment doit être dessiné ou non.
   - Le blending peut être effectué si la transparence est impliquée, combinant les couleurs des nouveaux fragments avec celles des fragments existants dans le frame buffer.

```
gl.enable(gl.DEPTH_TEST); // Activation du test de profondeur
gl.depthFunc(gl.LEQUAL); // Configuration du test de profondeur
gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT); // Nettoyage des buffers
gl.enable(gl.BLEND); // Activation du blending
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA); // Configuration du blending
```

9. **Nettoyage** :

- Désactivation des attributs de sommet, des buffers, des textures, et du programme shader si nécessaire.
- Préparation pour le prochain cycle de rendu.

```
gl.disableVertexAttribArray(positionLoc);
gl.bindBuffer(gl.ARRAY_BUFFER, null);
gl.bindTexture(gl.TEXTURE_2D, null);
gl.useProgram(null);
```

Ces étapes incarnent l'essentiel du processus de rendu après l'initialisation des composants. Elles sont orchestrées de manière à garantir que les données de la scène sont correctement traitées et rendues à l'écran.