

Partie 2 : WebGL

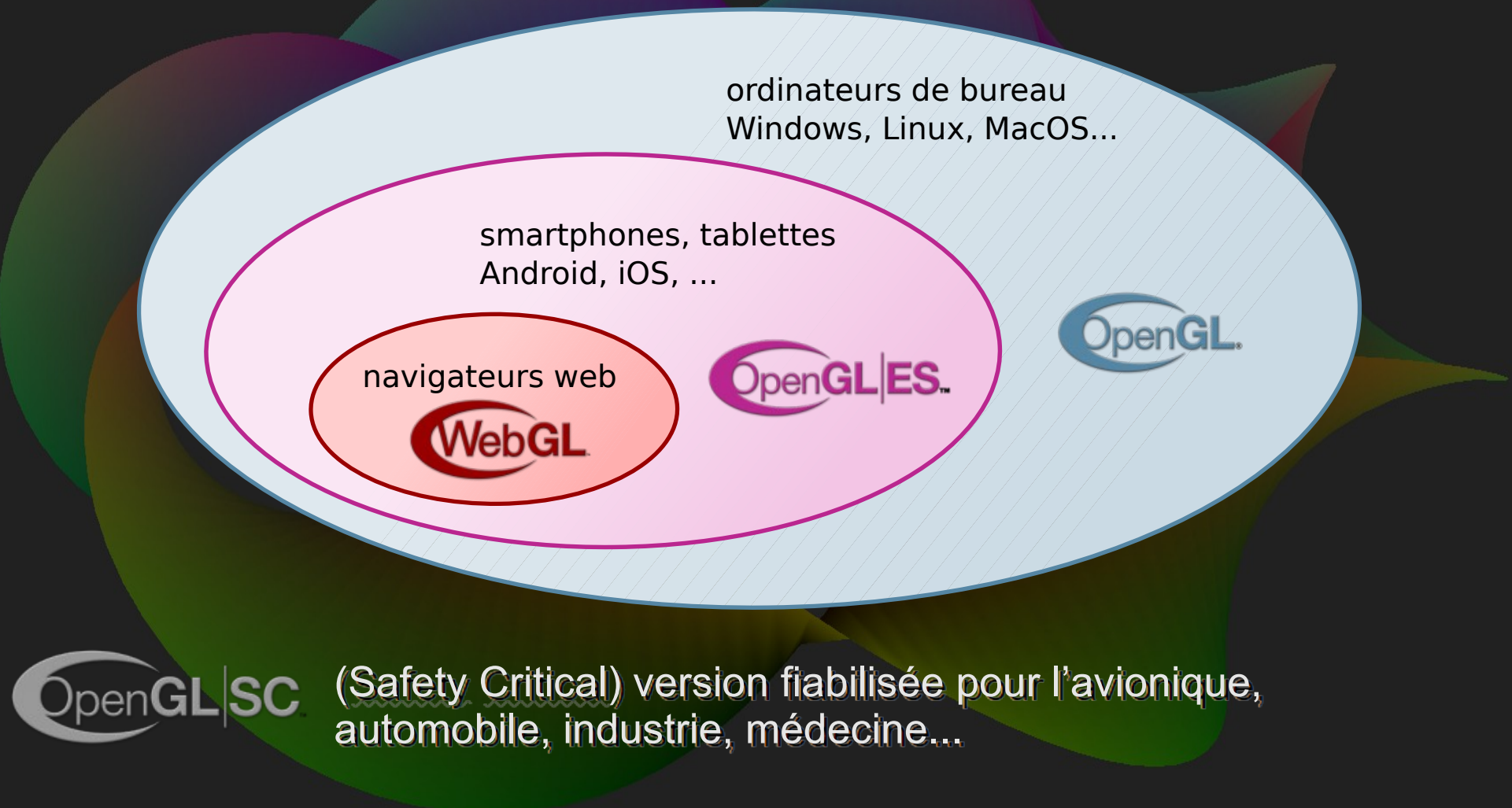
- OpenGL vs WebGL
- HTML5, JavaScript et WebGL
- VBO et Shaders
- Transformations, divers

2.1 - Qu'est-ce que OpenGL ?

- OpenGL et ses sous-ensembles sont des interfaces de programmation (API) graphique 3D
 - Bibliothèques de fonctions graphiques 3D
 - « dessiner tel triangle de telle couleur à telle position 3D »
 - et aussi un pilote (*driver*) de cartes graphique
 - commandes de bas niveau pour réaliser les fonctions 3D
- OpenGL est initialement destinée à des programmes C ou C++.
 - Des liaisons (*wrappers*) ont été écrits pour d'autres langages, ex : Python, JavaScript

OpenGL = famille d'API

- Famille structurée en sous-ensembles :



OpenGL, suite

- C'est la base de très nombreux logiciels 3D : CAO, moteurs de jeu...
- Fonctionnement interne :
 - Succession d'étapes : projection, pixelisation, coloration... qui seront expliquées au fur et à mesure
 - Parallélisme SIMD de plus en plus massif pour accélérer les calculs
 - Avenir :
 - programmer à plus bas niveau (Vulkan)
 - Intégration du lancer de rayons et autres

Qualités et défauts d'OpenGL

- **Qualités :**

- Disponible sur des plate-formes et systèmes variés : PC (Mac, Windows, Linux...), Raspberry, smartphones,
- Stable, homogène, compréhensible

- **Défauts :**

- Uniquement synthèse d'images (pas de son...)
- Un peu « éloigné des transistors » donc moins efficace (**Vulkan** est destiné à s'en rapprocher)
- En retard sur DirectX

Histoire d'OpenGL

■ Fondations

- Silicon Graphics Inc 1982 (Univ. Stanford)
 - Stations de travail 3D avec IrisGL (API propriétaire)
- Début 1990 : ouverture de IrisGL => OpenGL
 - Arrivée des cartes graphiques sur PC
- Consortium d'une centaine de constructeurs
 - ARB devenu Khronos Group

■ Difficultés rencontrées

- Faillite de Silicon Graphics Inc en 2006
- Compétition avec DirectX de Microsoft

Versions d'OpenGL

- **L'API principale existe en plusieurs versions :**
 - 1.x (1997 à 2003) et 2.x (2006) : antiquités, mais encore fonctionnelles
 - 3.x (2008 à 2010) : changements fondamentaux (VBO et shaders obligatoires)
 - 4.x (à partir de 2010) : intégration des avancées de DirectX11, complexité croissante
 - **Les autres OpenGL suivent ces versions de loin**
 - Ex : WebGL 2.0 = OpenGL ES 3.0 = OpenGL 4.3
- NB : on utilisera WebGL2, nommé WebGL pour simplifier dans la suite

Principe général d'OpenGL

- **Lancer de rayons :**
 - Pour chaque pixel, quelle est sa couleur ?
 - Recherche de l'objet le plus proche rencontré par un rayon œil-pixel, puis calcul de couleur
- **OpenGL = « pixelisation », voir wikipedia**
 - Pour chaque objet de la scène, où se dessine-t-il ?
 - Objets = points, lignes, triangles appartenant à des *maillages* (voir prochaines semaines)
 - Projection mathématique sur l'écran => pixels concernés (éventuellement plusieurs fois pour différents objets)
 - Calcul de couleur comme en lancer de rayons, mais sans rayons récurifs

Niveau de travail avec OpenGL

- **OpenGL** fournit un certain niveau d'abstraction par rapport aux processeurs graphiques
 - On prépare des primitives graphiques très simples : points, lignes et triangles dans des tableaux (VBO)
 - On écrit de petits programmes appelés *shaders* pour positionner et colorer ces primitives
 - On configure OpenGL selon les effets voulus
- **Écrire un programme OpenGL ou WebGL « pur » est assez compliqué**

Niveau de travail, suite

- **Choix possibles avec OpenGL ou WebGL :**
 - Programmation intégrale, directe : valable uniquement pour des programmes minuscules
 - Utilisation de Unity, Ogre, CryEngine, Unreal Engine... niveau supérieur : objets complexes, matériaux, éclairages et effets.
 - Utilisation d'une bibliothèque facilitante, ex : **Three.js**. Son niveau est entre WebGL de base et les systèmes précédents. Les programmes sont plus simples à écrire, mais pas portables.
- **Dans ce cours : programmation d'une bibliothèque personnelle, à but pédagogique**

2.2 - Divers, inclassable

- Rappels sur JavaScript 6
- Lien entre JavaScript et OpenGL/WebGL

(Rappels) JavaScript ECMA 6

- **Définition d'une variable locale**

```
const nb = 25; let dx = 0.15;
```

- Le type est spécifié par l'affectation

- **Définition d'un tableau**

```
const couleurs = [  
    1.0, 0.0, 0.5,  
    0.0, 0.0, 1.0  
];  
console.log(couleurs.length);  
for (let c in/of couleurs) {  
    console.log(c);  
}
```

JavaScript, suite

- Définition d'une classe

```
class Accum {  
  constructor(valinit) {  
    this.m_Nombre = valinit;  
  }  
  ajouter(inc) {  
    this.m_Nombre += inc;  
  }  
}
```

- Utilisation

```
let acc = new Accum(3);  
acc.ajouter(2);  
console.log(acc.m_Nombre);
```

Dépannage

- Il est indispensable de savoir déboguer un programme JavaScript
 - Navigateur : afficher la console
 - Étudier les messages d'erreurs
 - Erreur de syntaxe, erreur de saisie du programme ?
 - Erreur dans les résultats : algorithmes, données ?
 - Ajouter des instructions d'affichage pour vérifier la progression : `console.log("v=", v);`

OpenGL en JavaScript

- Un objet appelé `gl` représente l'API OpenGL
 - Les fonctions OpenGL sont des méthodes de cet objet ; les constantes sont ses membres. Il y a juste quelques petites modifications syntaxiques :

C++ : `glEnable(GL_DEPTH_TEST);`

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

C++ : `glClearColor(0.4, 0.4, 0.4, 1.0);`

JS : `gl.clearColor(0.4, 0.4, 0.4, 1.0);`

C++ : `glBindBuffer(GL_ARRAY_BUFFER, 0);`

JS : `gl.bindBuffer(gl.ARRAY_BUFFER, null);`

2.3 – WebGL

- **WebGL = OpenGL ES natif dans un navigateur**
 - C'est le client qui effectue les calculs d'affichage 3D à l'aide de sa carte graphique
 - Attaques pirates possibles, dénoncées par Microsoft (mais sûrement l'entreprise la plus mal placée pour en parler)
 - Les instructions OpenGL sont spécifiées en **JavaScript** dans le source HTML5, dans une balise `<script>`
 - On doit prévoir une balise `<canvas>` pour afficher l'image 3D dans la page

Structure d'un projet WebGL

- **Plusieurs fichiers :**
 - Le fichier HTML principal : `main.html`
 - Des inclusions de bibliothèques, dossier `libs` :
 - Calculs mathématiques : `glMatrix.js`
 - Utilitaires : `utils.js`
 - Des scripts supplémentaires :
 - Scène : 1 script, `scene.js` définissant la classe `Scene`
 - Objets de la scène : 1 script par type d'objets, chacun définissant une classe pour ce type d'objet
- **TD/TP : examiner le premier projet WebGL**

main.html simplifié

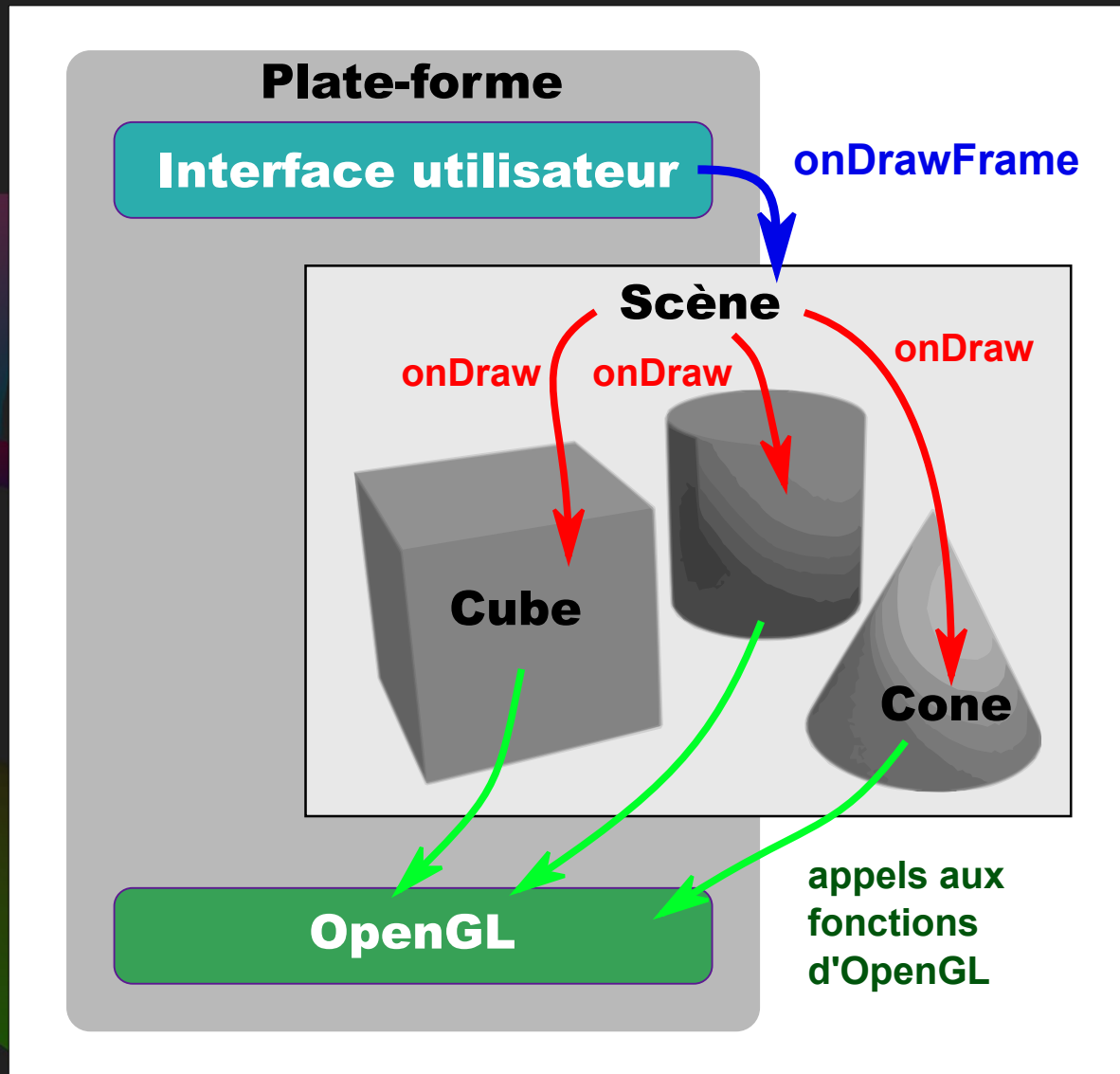
- **Entête = script JS d'initialisation**

- ```
function webGLStart() {
 let canvas =
 document.getElementById("opengl-canvas");
 initGL(canvas);
 scene = new Scene();
 scene.onSurfaceChanged(640, 480);
 refresh();
}
function refresh() { scene.onDrawFrame(); }
```

- **Corps = une balise canvas**

- ```
<body onload="webGLStart();" >  
    <canvas id="opengl-canvas" />  
</body>
```

Schéma général



Affichage

- Dans `main.html`
 - La fonction `refresh()` appelle la méthode `onDrawFrame()` sur l'instance de la classe `Scene`
- Si on veut animer la scène :
 - Mettre `Utils.repeatRefresh()` qui fait appeler la fonction `refresh()` aussi souvent que possible
 - On peut utiliser le nombre de secondes écoulées depuis le lancement du logiciel, dans `Utils.Time`, pour paramétrer une transformation géométrique (rotation, translation)

Classe Scene

■ Constructeur

```
class Scene {  
    constructor() {  
        this.triangle = new Triangle();  
        gl.clearColor(0.4, 0.4, 0.4, 1.0);  
    }  
}
```

Chaque objet dessiné est l'instance d'une classe

Les prochains effacements de l'écran seront de cette couleur

■ Définition de la taille de la fenêtre OpenGL

```
onSurfaceChanged(width, height) {  
    gl.viewport(0, 0, width, height);  
}
```

Zone de l'écran autorisée pour OpenGL

■ Affichage

```
onDrawFrame() {  
    gl.clear(gl.COLOR_BUFFER_BIT);  
    this.triangle.onDraw();  
}
```

Effacement de l'écran, voir gl.clearColor

Chaque classe d'objet a cette méthode de dessin

Classe Triangle

- **Constructeur**

```
class Triangle {  
    constructor() {  
        ... ça définit le triangle, cf + loin ...  
    }  
}
```

- **Affichage**

```
onDraw() {  
    ... ça dessine le triangle, cf + loin ...  
}
```

Jusque là, tout doit sembler compréhensible,
c'est la suite qui est technique...

Classe Triangle en détails

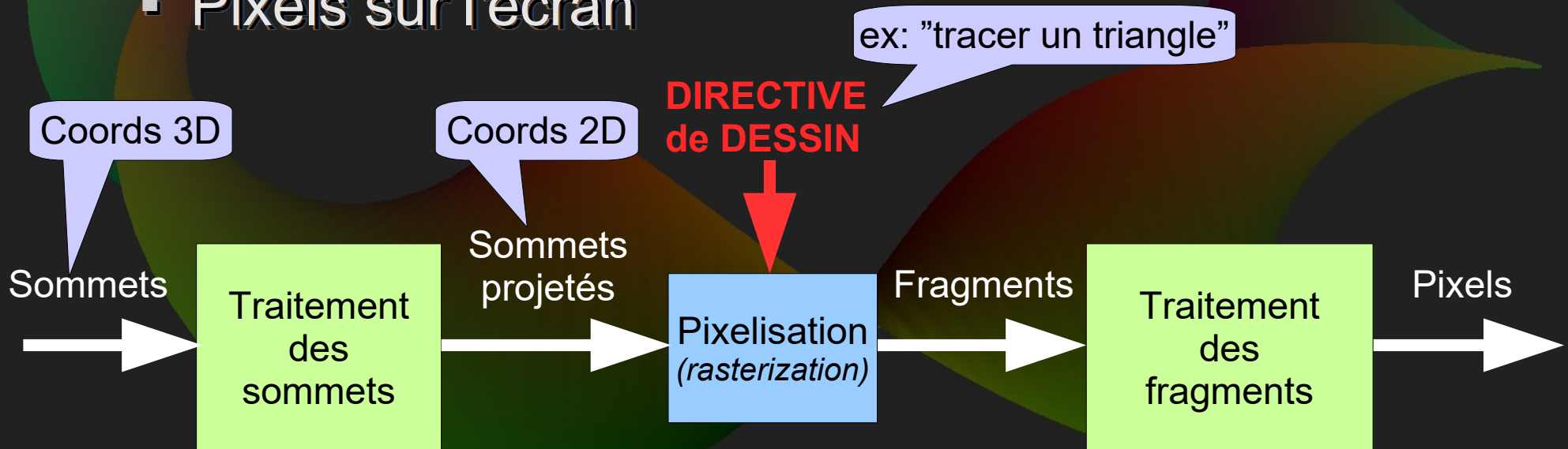
- 1) **Constructeur : prépare des éléments : points, lignes, triangles**
 - a) Remplissage de tableaux en mémoire (coordonnées des sommets et d'autres valeurs)
 - b) Création d'une fonction de calcul de couleur
- 2) **OnDraw : dessine ces éléments**
 - a) Configuration d'OpenGL (mode de travail)
 - b) Activation des tableaux et de la fonction de couleur
 - c) Dessin des éléments
 - d) Désactivation des tableaux et de la fonction

La suite, c'est...

- **Les mécanismes d'OpenGL :**
 - Shaders
 - Vertex Buffer Objects
 - Transformations géométriques
 - Textures
 - Frame Buffer Objects
- que nous verrons au long de ces prochains cours

2.4 - Principales étapes d'OpenGL

- **En entrée :**
 - Tableau des coordonnées des sommets
 - Directive de dessin, ex : TRIANGLES
- **En sortie :**
 - Pixels sur l'écran

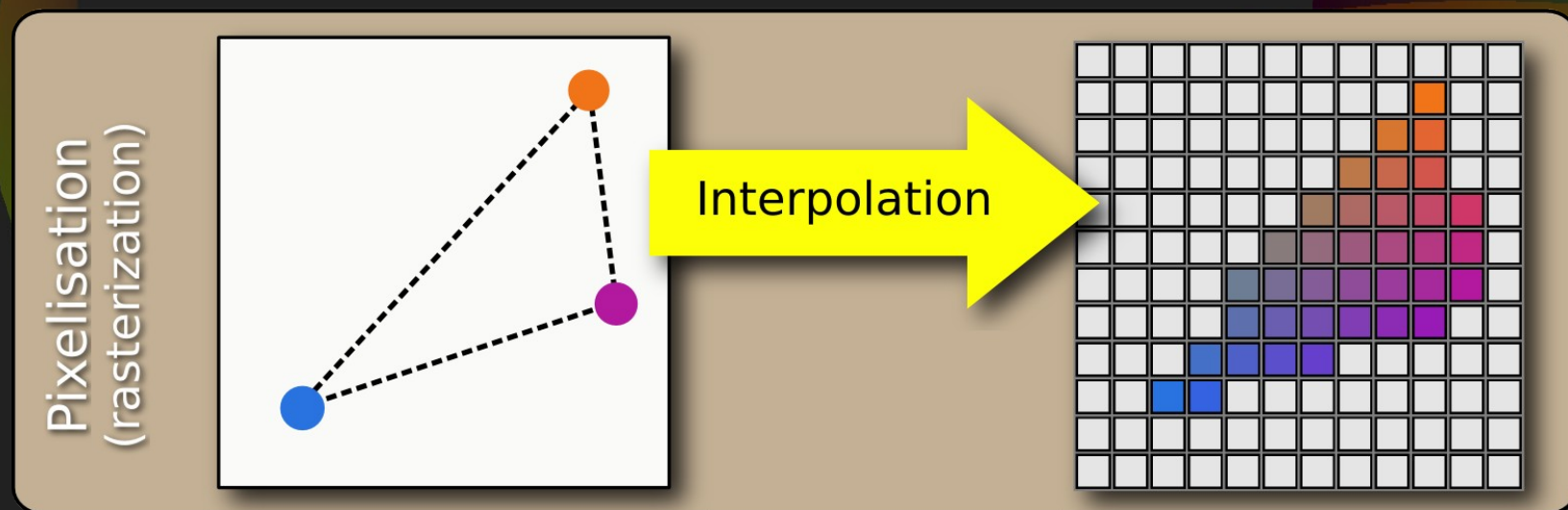


Traitement des sommets

- **Projection sur écran (obligatoire)**
 - Calcul des coordonnées écran à partir des coordonnées 3D et des transformations
 - Projection en perspective
 - Position et orientation relative à la caméra
- **Préparations aux calculs d'éclairement (optionnels, selon besoins)**
 - Calcul de la direction de la lumière
 - Calcul de la normale (sauf si fournie)

Rasterization

- Pixelisation = remplissage des primitives
 - Une ligne = 2 sommets, un triangle = 3 sommets
 - Le hardware détermine tous les pixels **fragments** contenus à l'intérieur et interpole les données présentes sur les sommets : couleur, normale...



Pixels Fragments

- Avec OpenGL, on ne parle pas de pixels, mais de fragments
 - Certains calculs peuvent être faits pour des sous-pixels (anti-crénelage)
 - Les fragments portent davantage d'informations que des pixels : pas seulement une couleur mais aussi diverses coordonnées (texture, normales, tangentes...) et variables de calcul
- Et on dit aussi vertex/vertices au lieu de sommet/sommets

Traitement des fragments

- **Calcul de la teinte finale**
 - Accès aux textures
 - Calcul d'éclairage Phong, Blinn...
 - **Puis la carte graphique termine avec :**
 - Test du depth buffer (voir fin de ce cours)
 - Test du stencil
 - Application du blending
- On verra cela au fur et à mesure

OpenGL 1 et 2 vs OpenGL 3 et 4

- Les anciennes versions d'OpenGL effectuaient elles-mêmes toutes les étapes de calcul :
 - Traitement des sommets
 - Pixelisation
 - Traitement des fragments
- Les versions actuelles demandent qu'on programme soi-même les traitements des sommets et des fragments

Prog. des traitements

- **Les traitements sont programmés en GLSL**
 - C'est un langage de programmation clair, très proche du langage C le plus simple
 - Nombreuses fonctions mathématiques géométriques
- **Votre programme JS fournit un source GLSL qui est compilé par OpenGL, on l'appelle *shader***
 - Le code compilé tourne directement sur le GPU
 - Le compilateur est intégré dans OpenGL. Par contre, les messages d'erreur ne sont pas très détaillés, ça dépend du pilote OpenGL...

GLSL dans OpenGL

- 1) Créer un programme GLSL (éditeur de texte standard ou spécialisé)
 - Deux sources GLSL à fournir : le vertex shader et le fragment shader
 - En C++ : fichiers, en WebGL : chaînes
- 2) Fournir ces sources à OpenGL qui les compile et les lie => programme GLSL
- 3) Activer le programme pour dessiner quelque chose puis désactiver le programme

2.5 - Deux sortes de shaders

On doit définir un :

- **Vertex shader** pour modifier la projection des vertices sur l'écran ou calculer des informations d'éclairage
- **Fragment shader** pour changer les calculs de couleur en chaque pixel afin de reproduire un matériau complexe
- Le VS produit des informations (ex : coordonnées projetées) qui sont transmises au FS après passage dans le pixeliseur

Vertex shader

- Il est exécuté pour chaque sommet
 - Les sommets sont placés dans des tableaux appelés VBO, voir plus loin
- Un sommet ou vertex, c'est :
 - (obligatoirement) des coordonnées 2D, 3D ou 4D selon les besoins ; ces coordonnées sont projetées sur l'écran
 - Les coordonnées projetées doivent aller dans une variable spéciale appelée **gl_Position**
 - NB : dans les premiers exemples des TP, il n'y a pas de projection
 - D'autres informations nécessaires pour le matériau, ex : couleurs, coordonnées de texture, normales, réflectivité...

Fragment shader

- Il est appelé pour chaque fragment généré par le pixeliseur
 - Il doit calculer la couleur finale du fragment et la placer dans une variable en sortie `glFragColor`
 - Elle s'appelait `gl_FragColor` dans GLSL 1.0
 - Il utilise les informations fournies par le vertex shader
- Exemples :
 - Équation de Phong, relief de matériau, application d'ombres portées, reflets simulés, etc...

Paramètres de shader

- Les shaders sont configurables de l'extérieur par le programme client
 - Ex : fournir le temps écoulé au shader
 - Ex : fournir les matrices de transformation
- Ce sont des variables qualifiées par `uniform`

```
uniform float Time;  
uniform mat3 matN;
```

 - Leur valeur reste constante pendant le dessin de toute une primitive graphique
 - Le script JS les affecte avant de dessiner, cf + loin

Variables interpolées

- En général, il faut transmettre des informations entre le Vertex Shader et le Fragment Shader
 - Couleur, normale, coordonnées de texture, etc.
- Pour cela
 - Le VS affecte une variable qualifiée par le mot clé **out**, ex : **out** vec3 frgN;
 - Le rasterizer interpole cette variable
 - Le FS définit la même variable qualifiée par le mot clé **in**, elle contiendra successivement les valeurs interpolées, ex : **in** vec3 frgN;
- NB : avant, on les qualifiait par le mot **varying**

Exemple de varying

■ Vertex Shader

```
#version 300 es
uniform mat3 matN;
in vec3 glNormal;
out vec3 frgN;
...
void main() {
    frgN = matN * glNormal;
    ...
}
```

Interpolée d'un frag. à l'autre

■ Fragment Shader

```
#version 300 es
...
in vec3 frgN;
...
void main() {
    vec3 N = normalize(frgN);
    float dotNL = dot(N, L);
    ...
}
```

2.6 - Vertex Buffer Objects

- Les sommets des primitives à dessiner doivent être placés dans des tableaux appelés VBO
 - 1 VBO par information : coordonnées, couleurs...
 - Le vertex shader définit une variable qualifiée par `in` pour recevoir les informations
 - `in vec3 couleur;`
 - NB : dans GLSL1.0, on les qualifiait de *attribute*
- Chaque sommet (1 n-uplet du VBO) est traité séparément par une instance de Vertex Shader
 - Parallélisme massif : de très nombreux sommets sont traités simultanément avec les mêmes calculs

VBO et Vertex shader

- Exemple le plus simple, copie des coordonnées

```
let Triangle_vs = `
    in vec2 glVertex;
    void main() {
        gl_Position = vec4(glVertex, 0.0, 1.0);
    } `;

this.shaderId = Utils.makeShaderProgram(
    Triangle_vs, Triangle_fs, "shader Triangle");
```

- Mot-clé `in` = variable à lier avec un VBO
 - Ici : `vec2` => les nombres seront pris 2 par 2 pour remplir la variable `glVertex`
 - Le nom provient des anciennes versions d'OpenGL

VBO et attribute (in)



- Le VBO contient les coordonnées des sommets, elles sont placées par paires, et injectées à tour de rôle dans la variable glVertex

Création d'un VBO

- Création d'un VBO

```
let vertices = [  
    -0.88, +0.52, // P0  
    +0.63, -0.79, // P1  
    +0.14, +0.95, // P2  
];  
this.vertexBufferId =  
    Utils.makeFloatVBO(vertices,  
        gl.ARRAY_BUFFER, gl.STATIC_DRAW);
```

- La fonction `Utils.makeFloatVBO(tableau, type, usage)` est dans la bibliothèque `utils.js`
 - NB : ici, pas de notion de nombre d'éléments
 - Son cœur est un appel à `glBufferData`

Attention, piège

- Dans un VBO, les données sont à la suite, sans aucune séparation

```
let vertices = [ -0.88, +0.52, +0.63, -0.79,  
+0.14, +0.95 ];
```

- C'est à vous de faire attention au nombre de valeurs devant entrer ensemble dans le Vertex Shader

Ici, c'est **2**

- Il est prudent de définir une constante :

```
const nbfloatpervertex = 2;
```

Emploi d'un VBO

- Dans la méthode draw, on active le VBO :

```
gl.bindBuffer(gl.ARRAY_BUFFER, this.vertexBufferId);  
gl.enableVertexAttribArray(this.glVertexLoc);  
gl.vertexAttribPointer(this.glVertexLoc,  
    nbfloatpervertex, gl.FLOAT, gl.FALSE, 0, 0);
```

- La variable `glVertexLoc` contient « l'emplacement » (sorte d'adresse) de la variable *attribute* `glVertex` dans le shader

```
this.glVertexLoc = gl.getAttributeLocation(  
    this.shaderId, "glVertex");
```

Préparations (constructeur)

- Définir et compiler les shaders

```
let Triangle_vs = `...`;  
let Triangle_fs = `...`;  
this.shaderId = Utils.makeShaderProgram(...);
```

- Récupérer les variables « in » du vertex shader

```
this.glVertexLoc = gl.getAttributeLocation(...);
```

- Créer les VBO

```
let vertices = [...];  
this.vertexBufferId = Utils.makeFloatVBO(...);
```

Dessin (onDraw)

- Activer le programme de shader

```
gl.useProgram(id du shader concerné);
```

- Activer et lier les VBOs aux variables du shader

```
gl.bindBuffer(gl.ARRAY_BUFFER, id du vbo concerné);  
gl.enableVertexAttribArray(loc de la var concernée);  
gl.vertexAttribPointer(...);
```

- Dessiner les primitives graphiques

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

- Tout désactiver

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

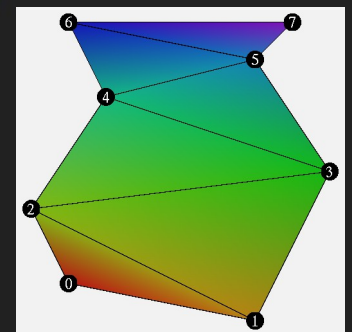
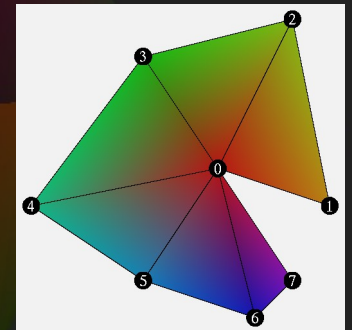
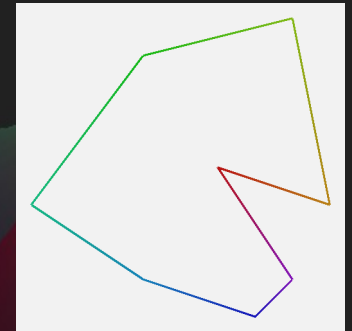
Primitives de dessin

- Pour dessiner, on appelle
`gl.drawArrays(code, 0, nombre);`
 - Code = identifiant de la primitive
 - Nombre = nombre de sommets à utiliser
- OpenGL offre peu de primitives :
 - Points : `gl.POINTS`
 - Lignes : `gl.LINES`
 - Triangles : `gl.TRIANGLES`

Ex : `gl.drawArrays(gl.TRIANGLES, 0, 6)` dessine 2 triangles

Primitives liées

- Il y a également les primitives liées :
 - Ligne fermée : `gl.LINE_LOOP`
Ligne ouverte : `gl.LINE_STRIP`
 - Éventail de triangles : `gl.TRIANGLE_FAN`
 - Ruban de triangles : `gl.TRIANGLE_STRIP`

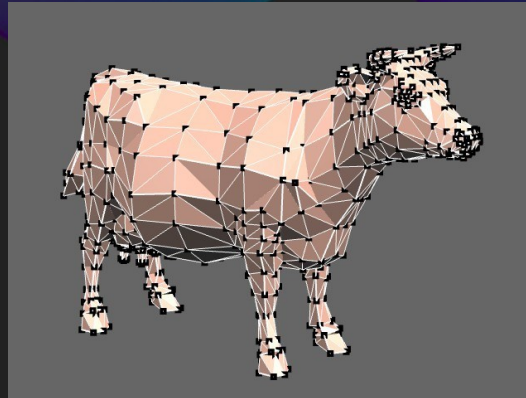


Configuration des primitives

- Il est possible de changer
 - La taille des points : affecter la variable spéciale `gl_PointSize` dans le vertex shader (voir plus loin)
 - Dans OpenGL uniquement, la largeur des lignes : `glLineWidth(largeur)` ;
 - Les navigateurs ne tiennent pas compte de `gl.lineWidth` avec un paramètre autre que 1, voir <https://stackoverflow.com/questions/47444239/how-to-use-gl-linewidth>
- D'autres réglages (hachures, remplissage des polygones) sont possibles avec OpenGL mais pas avec WebGL

2.7 - Primitives indexées

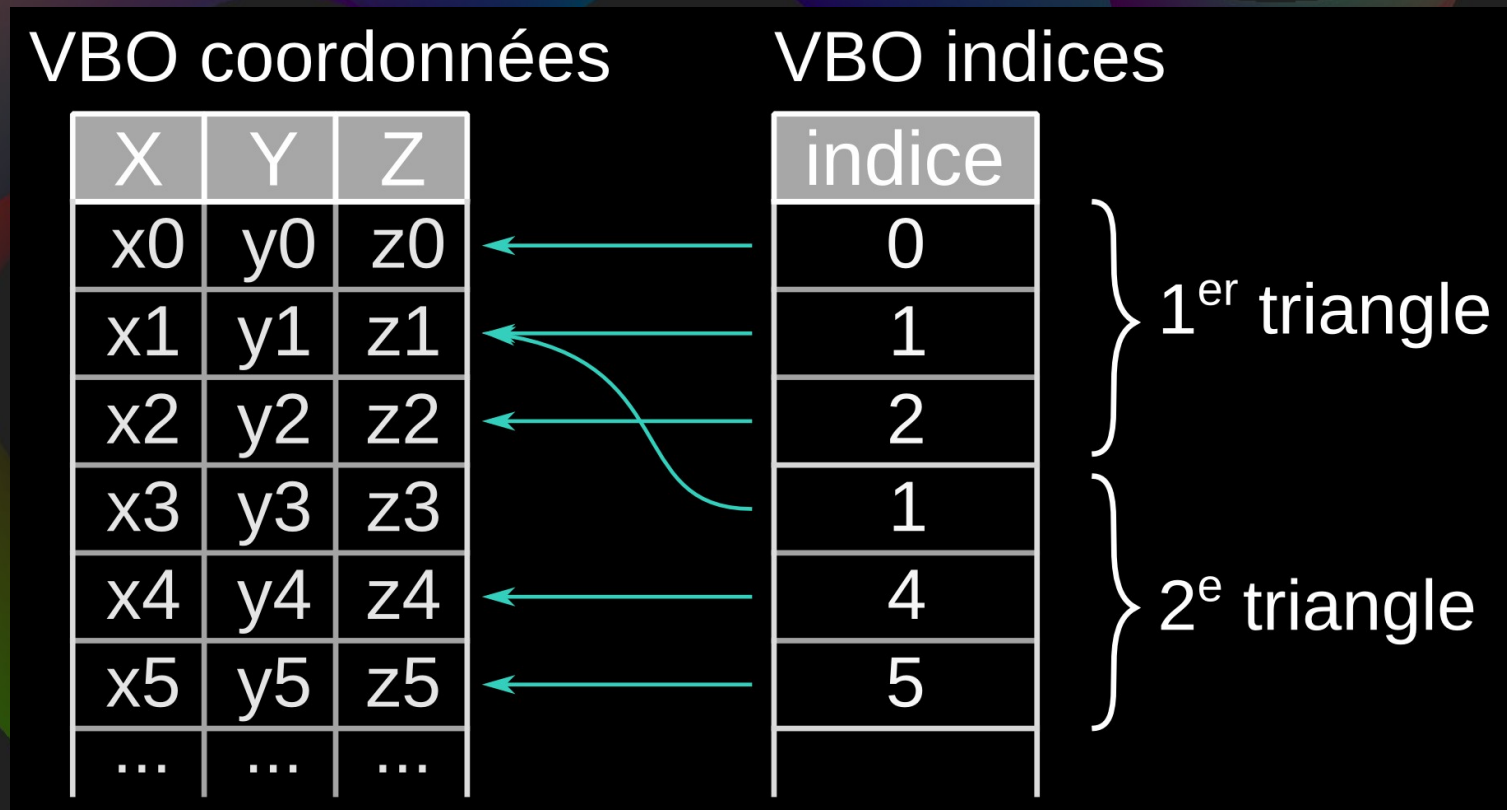
- Dans un objet complexe, les mêmes sommets servent dans plusieurs triangles



- Au lieu de dupliquer les mêmes coordonnées, on ne met qu'un seul exemplaire des coordonnées de chaque sommet, et on dit aux triangles lesquels ils doivent utiliser, à l'aide de numéros

Primitives indexées, suite

- Ça fait deux VBO, l'un pour les coordonnées, et un spécial pour des numéros de sommets :



VBO d'indices

- **Création**

```
let indices = [ nos des sommets ];  
id_VBO = Utils.makeShortVBO(indices,  
    gl.ELEMENT_ARRAY_BUFFER, gl.STATIC_DRAW);
```

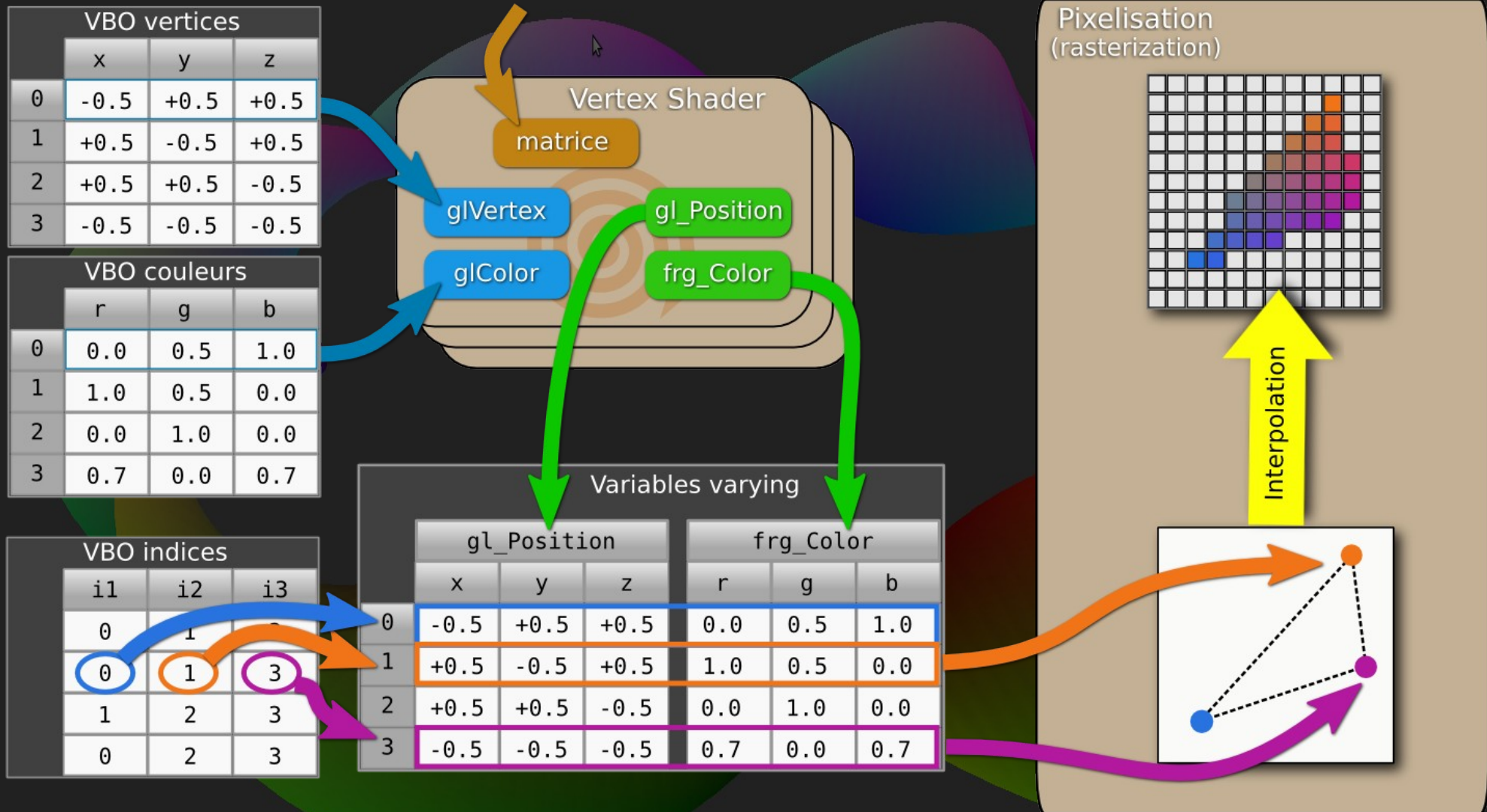
- **Activation**

```
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, id_VBO);
```

- **Dessin**

```
gl.drawElements(gl.TRIANGLES, nb_indices,  
    gl.UNSIGNED_SHORT, 0);
```

VBO indexés et shaders

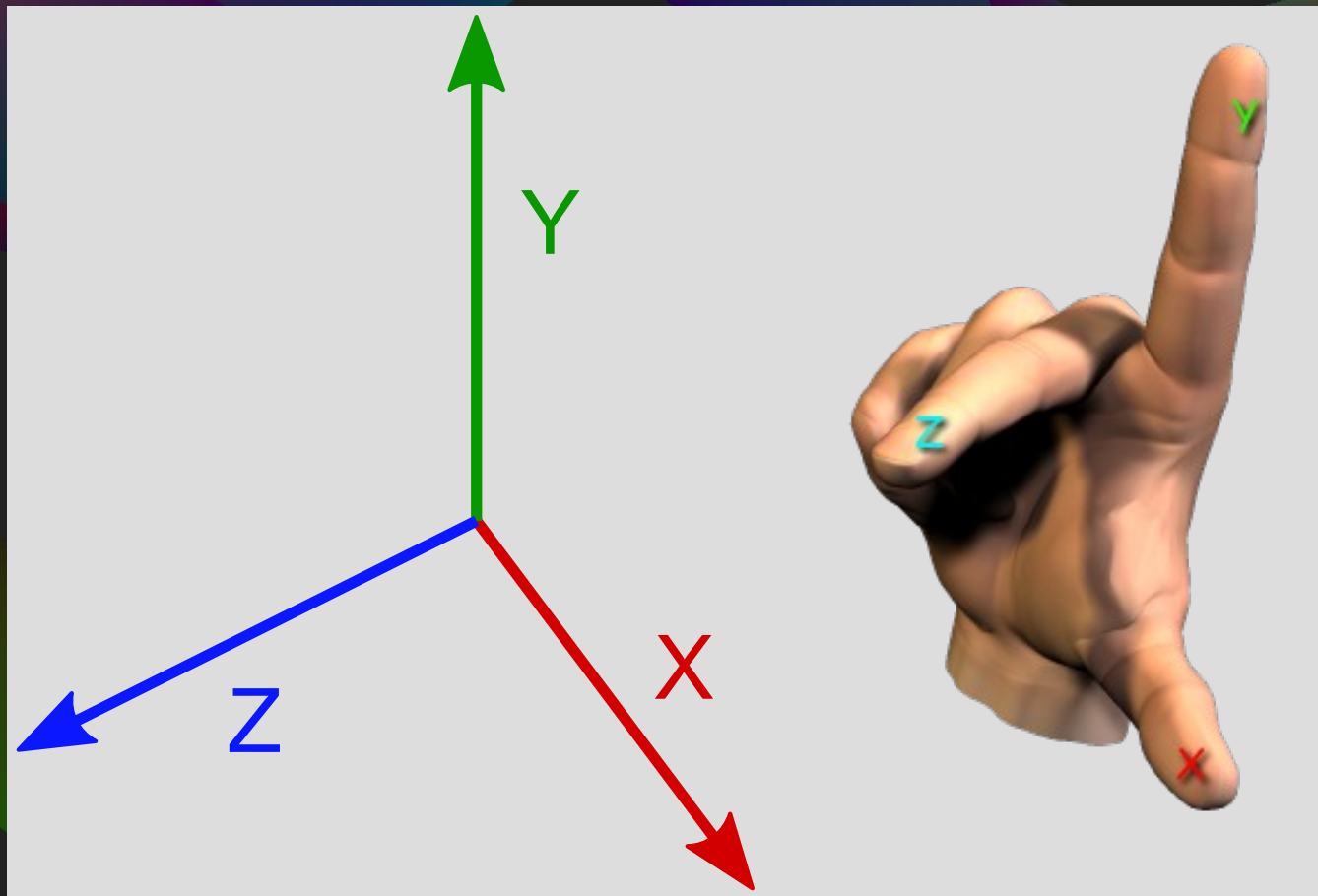


2.8 - Transformations

- Les transformations géométriques permettent de modifier les objets dessinés sans devoir re-remplir les VBO
 - Translations
 - Rotations
 - Homothéties
 - Projections
 - Autres affines (cisaillements, ...)
- Elles sont écrites sous formes de matrices

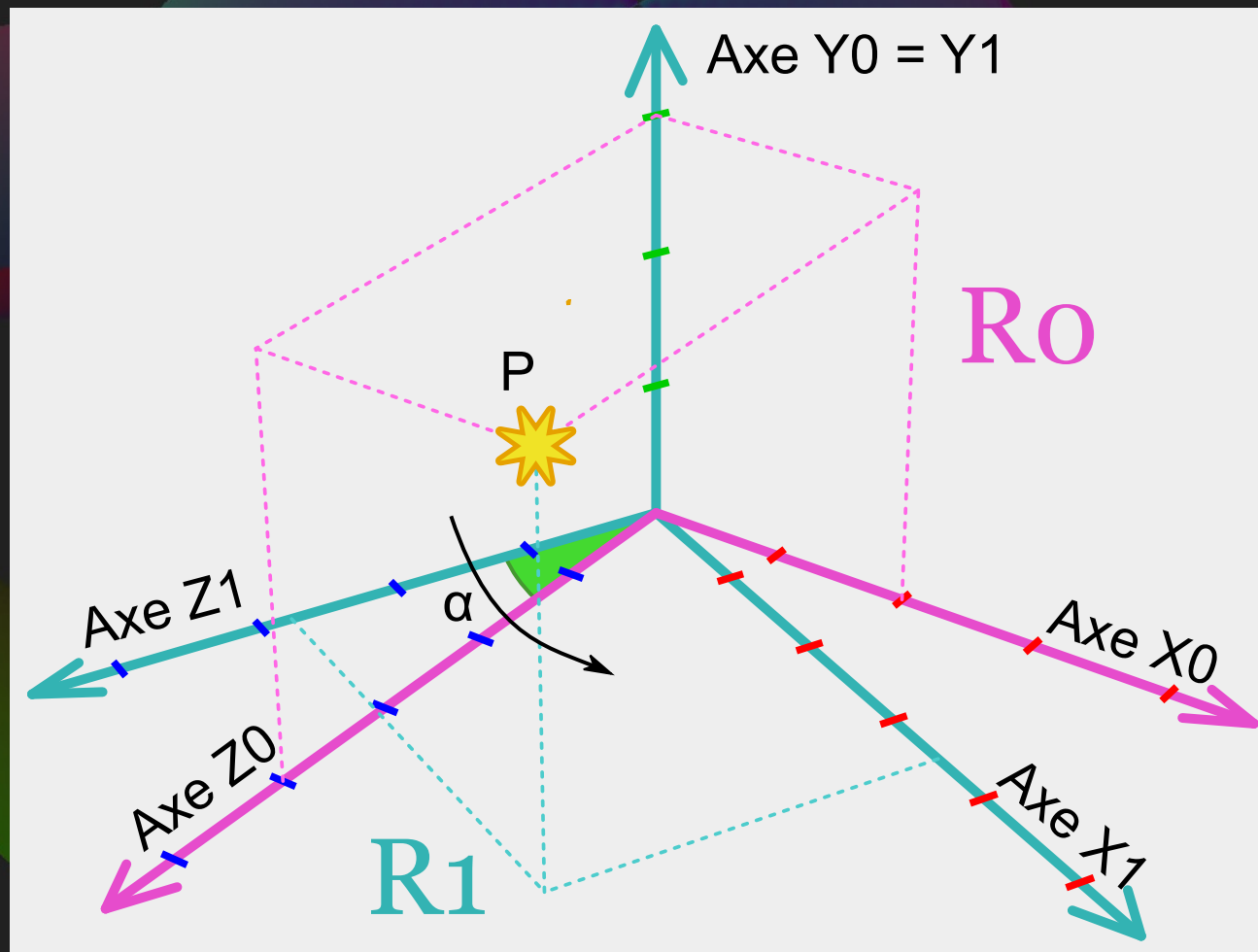
Coordonnées 3D

- Avec OpenGL, on a pris pour habitude de travailler dans un repère « main droite »



Exemple de transformation

- Rotation d'axe Y et d'angle α



Matrices

$$P1 = M * P0$$

Effectue une transformation géométrique sur P0

C'est un changement de repère

- $P0$ = coordonnées de P dans le repère $R0$
- $P1$ = coordonnées dans le repère d'arrivée $R1$
- M = représentation de $R0$ par rapport à $R1$

NB : coordonnées écrites en colonnes

- Tous les détails mathématiques au prochain cours, seulement des infos pratiques ici

Matrices 2x2

- Pour des transformations planes
 - Ex : rotation d'angle α dans le plan xz

$$\begin{pmatrix} z1 \\ x1 \end{pmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix} * \begin{pmatrix} z0 \\ x0 \end{pmatrix}$$

- NB : les coefficients sont donnés pour raviver la mémoire, mais la bibliothèque mathématique qu'on va utiliser masque cela et permet d'éviter les erreurs

Matrices 3x3

- Pour transformer des vecteurs 3D
 - Ex : rotation d'angle α dans le plan xz

$$\begin{pmatrix} x1 \\ y1 \\ z1 \end{pmatrix} = \begin{bmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix} * \begin{pmatrix} x0 \\ y0 \\ z0 \end{pmatrix}$$

- Ex : homothétie

$$\begin{pmatrix} x1 \\ y1 \\ z1 \end{pmatrix} = \begin{bmatrix} sX & 0 & 0 \\ 0 & sY & 0 \\ 0 & 0 & sZ \end{bmatrix} * \begin{pmatrix} x0 \\ y0 \\ z0 \end{pmatrix}$$

Matrices 4x4

- Pour transformer des points écrits en coordonnées homogènes $(x \ y \ z \ w)^T$
 - Classe d'équivalence $(x.w \ y.w \ z.w \ w)^T = (x \ y \ z \ 1)^T$
 - $(x \ y \ z \ w)^T$ est un point si $w \neq 0$ et un vecteur si $w = 0$
 - Intérêt : représenter les translations et les projections perspectives

$$\begin{pmatrix} x1 \\ y1 \\ z1 \\ 1 \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{pmatrix} x0 \\ y0 \\ z0 \\ 1 \end{pmatrix}$$

Composition de transformations

- On peut enchaîner des transformations
 - Soit $P1 = M1 * P0$ puis $P2 = M2 * P1$
 - On a donc $P2 = M2 * (M1 * P0)$
 - On peut écrire $P2 = (M2 * M1) * P0$
représente l'application de M2 sur P1, lui-même étant le transformé de P0 par M1
Le produit des matrices $M2 * M1$ représente le regroupement des transformations M2 et M1 en une seule
- Attention, en général $M2 * M1 \neq M1 * M2$

Emploi dans OpenGL

- Le vertex shader transforme les sommets à l'aide d'une matrice :

```
... mat4 matrix;  
in vec3 glVertex;  
gl_Position = matrix * vec4(glVertex, 1.0);
```

- Deux possibilités pour définir cette matrice :
 - La créer dans le shader
 - La créer dans le programme client (JavaScript)

Matrice dans le shader

- Le shader définit et initialise une variable de type `mat2`, `mat3` ou `mat4`
 - `vec4 col0 = vec4(1, 0, 0, 0);`
`vec4 col1 = vec4(0, 1, 0, 0);`
`vec4 col2 = vec4(0, 0, 1, 0);`
`vec4 col3 = vec4(tx, ty, tz, 1);`
 - `mat4 matrix = mat4(col0, col1, col2, col3);`
- Les matrices sont définies par colonne, cela correspond à la notion de changement de repère, voir prochain cours.

Matrice ds le programme client

- Le shader définit la matrice en tant que variable *uniform*, c'est à dire extérieure au shader :

```
uniform mat4 matrix;
```

- Le programme JavaScript initialise une matrice et la fournit au shader :

```
let rotation = mat4.create();  
mat4.rotateY(rotation, rotation, angle);  
let matrixLoc = gl.getUniformLocation(  
    this.shaderId, "matrix");  
mat4.glUniformMatrix(matrixLoc, rotation);
```

Création de matrices en JS

- La librairie gl-matrix fournit des fonctions pour gérer les vecteurs et matrices

```
let M1 = mat4.create();  
mat4.rotateY(M1, M1, Utils.radians(25.0));
```

- Le principe général des fonctions :
 - classe.fonction(sortie, entrée, paramètres...);

```
var V = vec3.create();  
vec3.scale(V, vec3.fromValues(1,2,3), 4);  
vec3.transformMat4(V, V, rotation);
```
 - Gare aux partages ! (utiliser la fonction clone)

Utilisation des fonctions

- Les fonctions `translate`, `rotate`, `scale` effectuent un produit à droite :

```
let M1 = mat4.create();
```

```
let M2 = mat4.create();
```

```
mat4.rotateY(M2, M1, angle);
```

=> $M2 = M1 * \text{rotation d'axe Y}$

et non pas $M2 = \text{rotation d'axe Y} * M1$

- Cela influe sur la composition des transformations

Composition de transformations

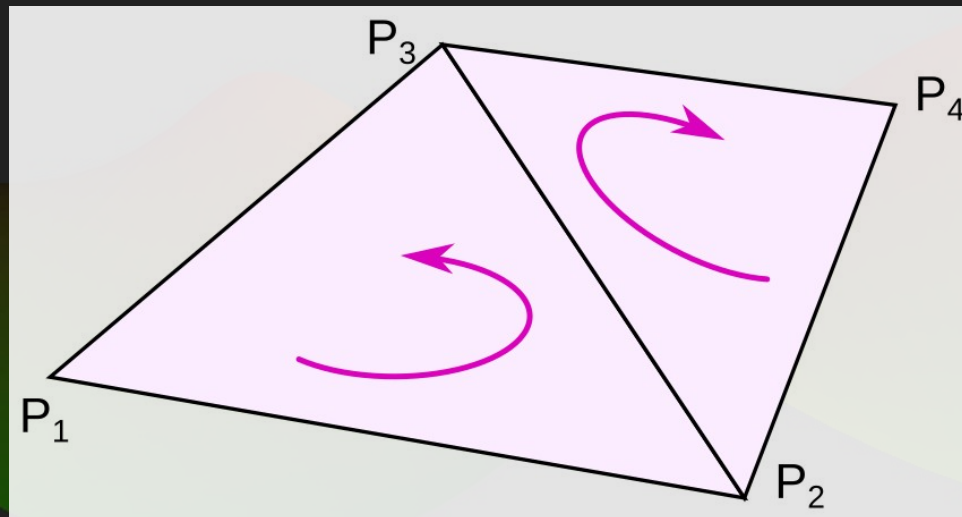
- Avec glmMatrix, la composition de transformations est un peu paradoxale : elles semblent appliquées à l'envers

```
let M = mat4.create();  
mat4.rotateY(M, M, angle);  
mat4.translate(M, M, ...);
```

- $\Rightarrow M = \text{rotationY} * \text{translation}$
et non pas $M = \text{translation} * \text{rotationY}$
- La translation est appliquée en premier sur les sommets quand on fait $P1 = M * P0$

2.9 – Élimination des faces cachées

- Il est intéressant de ne pas dessiner les facettes vues de dos d'un objet opaque
 - circulation sens trigo, en repère main droite
 - Le triangle P_1, P_2, P_3 est vu de face
 - Le triangle P_2, P_3, P_4 est vu de dos



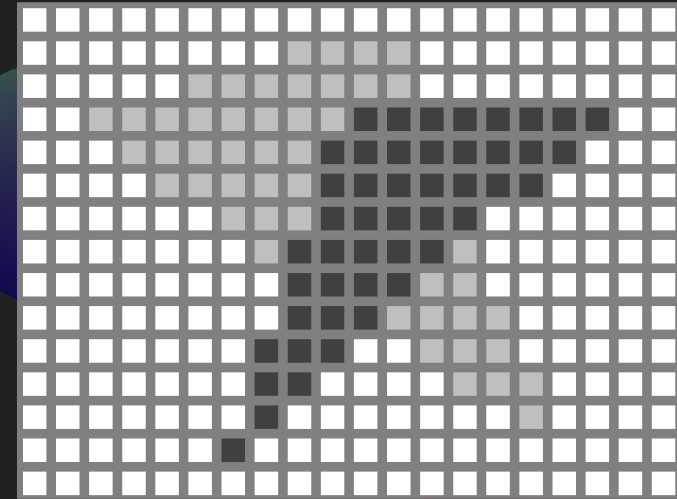
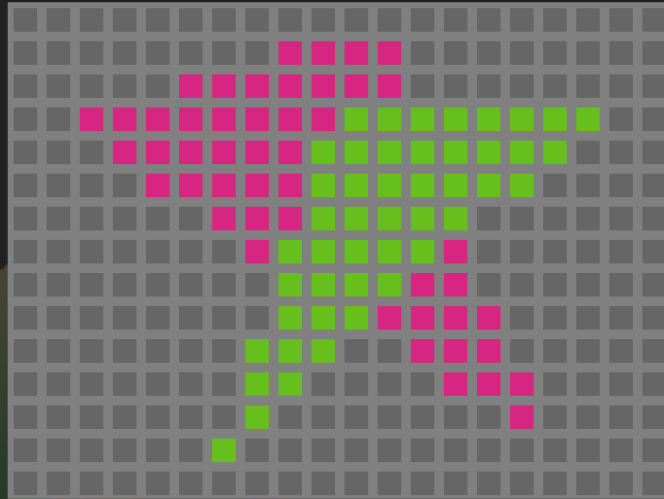
Mise en œuvre

- Il faut donc faire très attention à l'ordre des sommets (VBO d'index ou VBO de données)
- Configuration
 - `gl.enable(gl.CULL_FACE);`
 - `gl.cullFace(gl.BACK);`
- Ensuite, c'est automatique : à la sortie du vertex shader, un triangle mal orienté ne sera pas dessiné

2.10 – Ordre de dessin

- Les dessins ultérieurs recouvrent les dessins précédents, même quand on gère de la 3D (présence de la coordonnée z)
- Deux méthodes :
 - Classer les triangles pour dessiner de l'arrière vers l'avant : « algorithme du peintre »
 - Utiliser un **depth buffer** :
 - Tableau 1 réel par pixel écran donnant la distance de chaque pixel
 - Test avant de dessiner chaque pixel : sa profondeur doit être inférieure à celle présente dans le buffer

Depth buffer



- Depth buffer : clair = loin, foncé = proche
- Color buffer : pixels visibles

Mode d'emploi

■ Configuration

- Il faut que la fenêtre ait un depth buffer (vérifier ce qui est configuré pour sa création)
- `gl.enable(gl.DEPTH_TEST);`
- `gl.depthFunc(gl.LESS);`
- `gl.clearDepth(1.0);`

■ Utilisation

- `gl.clear(... | gl.DEPTH_BUFFER_BIT);`
- Dessiner comme avant, le depth buffer est automatiquement pris en compte