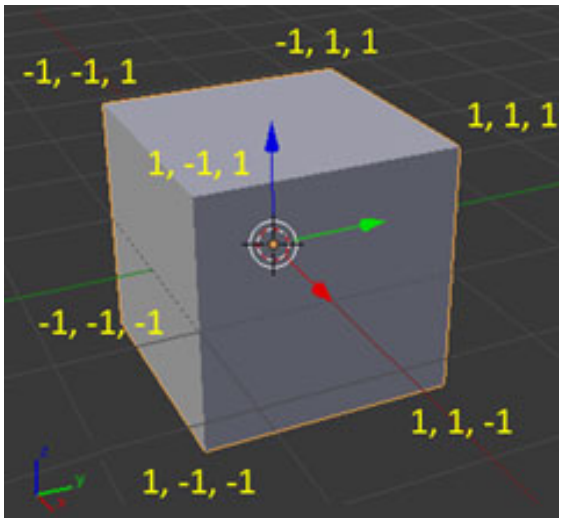
Introduction au jargon de la 3D et à WebGL

CIR3 - 2023/2024

Comment on modélise un objet 3D ?

Dans le jargon des spécialistes, on appelle cela un ***mesh***. Un *mesh*, c'est tout simplement un objet 3D.
Ce mesh est constitué à partir d'un nuage de points. Le point s'appelle un ***vertex***. Quand il y en a plusieurs, on dit ***vertices***. Un vertex est donc un point 3D.



Comment retranscrire la 3D vers la 2D de l'écran ?

On s'imagine comme étant un réalisateur du cinéma : on a une caméra, un environnement où filmer (le monde 3D), des objets (les acteurs, les meubles) et on va placer ces objets dans ce monde.

Comment retranscrire la 3D vers la 2D de l'écran ?

On s'imagine comme étant un réalisateur du cinéma : on a une caméra, un environnement où filmer (le monde 3D), des objets (les acteurs, les meubles) et on va placer ces objets dans ce monde.

1. On décrit le mesh à l'aide d'autant de **points** que nécessaire et on répète ces opérations pour tous les objets qui vont constituer votre univers.

Comment retranscrire la 3D vers la 2D de l'écran ?

On s'imagine comme étant un réalisateur du cinéma : on a une caméra, un environnement où filmer (le monde 3D), des objets (les acteurs, les meubles) et on va placer ces objets dans ce monde.

1. On décrit le mesh à l'aide d'autant de **points** que nécessaire et on répète ces opérations pour tous les objets qui vont constituer votre univers.
2. On positionne ces objets dans l'univers 3D. Les déplacements, rotations, mise à l'échelle sont réalisés grâce à des **calculs matriciels**.

Comment retranscrire la 3D vers la 2D de l'écran ?

On s'imagine comme étant un réalisateur du cinéma : on a une caméra, un environnement où filmer (le monde 3D), des objets (les acteurs, les meubles) et on va placer ces objets dans ce monde.

1. On décrit le mesh à l'aide d'autant de **points** que nécessaire et on répète ces opérations pour tous les objets qui vont constituer votre univers.
2. On positionne ces objets dans l'univers 3D. Les déplacements, rotations, mise à l'échelle sont réalisés grâce à des **calculs matriciels**.
3. On filme avec **une caméra** ce qu'on a créé. On va donc la positionner dans ce monde 3D et viser un objet particulier. La caméra a des propriétés identiques à une « vraie » caméra comme l'ouverture par exemple.

Comment retranscrire la 3D vers la 2D de l'écran ?

On s'imagine comme étant un réalisateur du cinéma : on a une caméra, un environnement où filmer (le monde 3D), des objets (les acteurs, les meubles) et on va placer ces objets dans ce monde.

1. On décrit le mesh à l'aide d'autant de **points** que nécessaire et on répète ces opérations pour tous les objets qui vont constituer votre univers.
2. On positionne ces objets dans l'univers 3D. Les déplacements, rotations, mise à l'échelle sont réalisés grâce à des **calculs matriciels**.
3. On filme avec **une caméra** ce qu'on a créé. On va donc la positionner dans ce monde 3D et viser un objet particulier. La caméra a des propriétés identiques à une « vraie » caméra comme l'ouverture par exemple.
4. On effectue **une projection** de ce que voit la caméra en 3D vers un rendu en 2D pour l'écran.

Comment retranscrire la 3D vers la 2D de l'écran ?

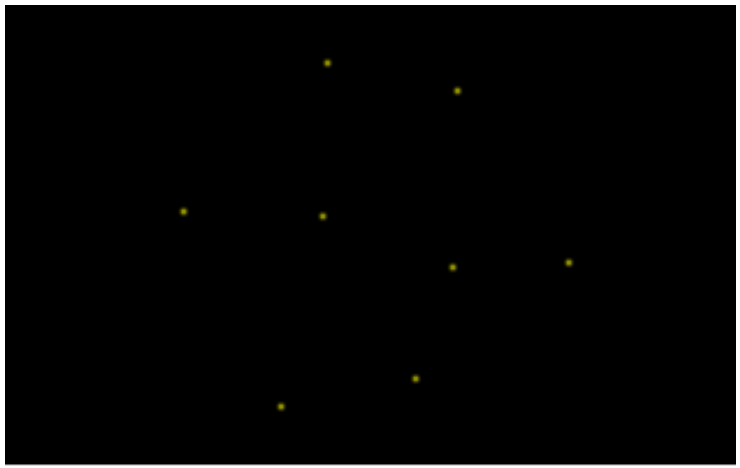
On s'imagine comme étant un réalisateur du cinéma : on a une caméra, un environnement où filmer (le monde 3D), des objets (les acteurs, les meubles) et on va placer ces objets dans ce monde.

1. On décrit le mesh à l'aide d'autant de **points** que nécessaire et on répète ces opérations pour tous les objets qui vont constituer votre univers.
2. On positionne ces objets dans l'univers 3D. Les déplacements, rotations, mise à l'échelle sont réalisés grâce à des **calculs matriciels**.
3. On filme avec **une caméra** ce qu'on a créé. On va donc la positionner dans ce monde 3D et viser un objet particulier. La caméra a des propriétés identiques à une « vraie » caméra comme l'ouverture par exemple.
4. On effectue **une projection** de ce que voit la caméra en 3D vers un rendu en 2D pour l'écran.

Toute la magie est réalisée grâce aux matrices ! C'est la base de la 3D.

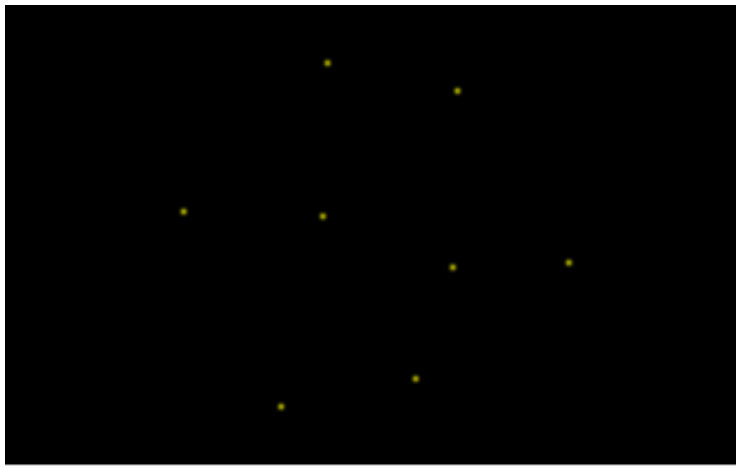
Premier rendu

À ce stade-là, si nous n'affichons que les points du cube, on a :



Premier rendu

À ce stade-là, si nous n'affichons que les points du cube, on a :



Pour aller plus loin, il va falloir dessiner des lignes entre ces points.
Pour être plus précis, on va même dessiner des **triangles**.

Polygônes

Le triangle est la forme 2D la plus simple qui existe en géométrie.

Polygônes

Le triangle est la forme 2D la plus simple qui existe en géométrie.

En résumé, pour pouvoir commencer à envisager une transformation du monde 3D vers le monde 2D de votre écran, il faut commencer par raisonner par groupe de **3 vertices** qui vont nous permettre d'en dessiner des triangles.

Polygônes

Le triangle est la forme 2D la plus simple qui existe en géométrie.

En résumé, pour pouvoir commencer à envisager une transformation du monde 3D vers le monde 2D de votre écran, il faut commencer par raisonner par groupe de **3 vertices** qui vont nous permettre d'en dessiner des triangles.

Dans le jargon, on parle du nombre de **polygônes** qu'une carte graphique est capable de « cracher ». Et bien, ce sont ces fameux triangles.

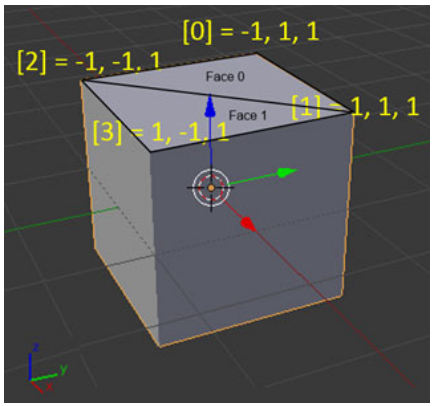
Polygônes

Le triangle est la forme 2D la plus simple qui existe en géométrie.

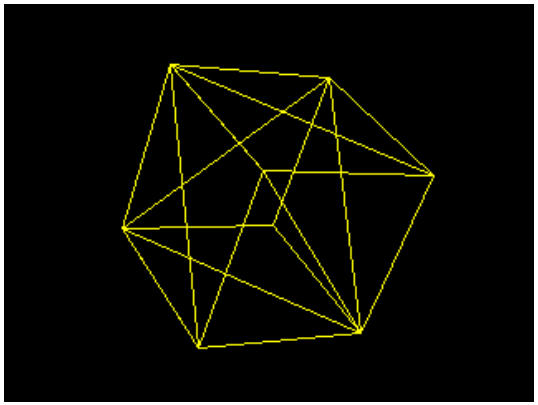
En résumé, pour pouvoir commencer à envisager une transformation du monde 3D vers le monde 2D de votre écran, il faut commencer par raisonner par groupe de **3 vertices** qui vont nous permettre d'en dessiner des triangles.
Dans le jargon, on parle du nombre de **polygônes** qu'une carte graphique est capable de « cracher ». Et bien, ce sont ces fameux triangles.

Il y a ensuite plusieurs façons de dessiner les triangles que l'on appelle également **une face**. Une face est donc constituée d'un groupe de 3 vertices qui vont nous permettre de savoir comment dessiner nos différents triangles.

Polygônes

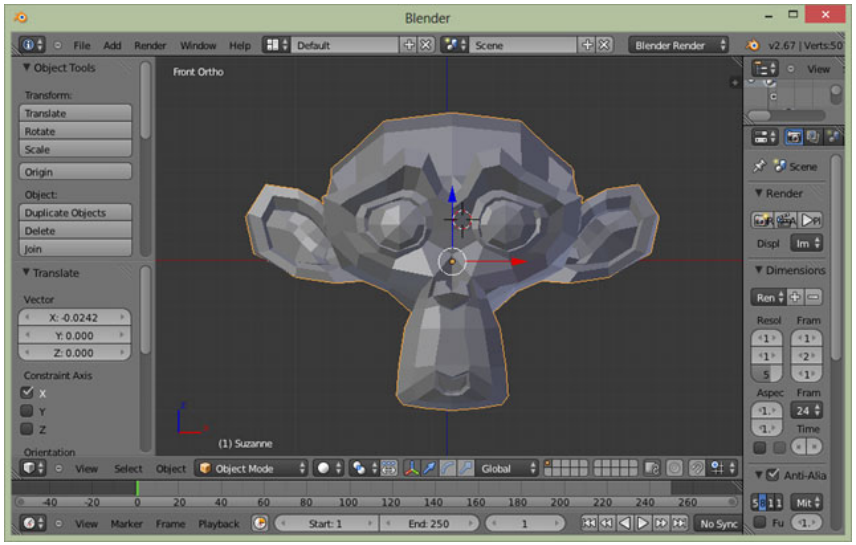



```
var mesh = new Engine.Mesh("Cube", 8, 12);
meshes.push(mesh);
mesh.Vertices[0] = new Vector3(-1, 1, 1);
mesh.Vertices[1] = new Vector3(1, 1, 1);
mesh.Vertices[2] = new Vector3(-1, -1, 1);
mesh.Vertices[3] = new Vector3(1, -1, 1);
mesh.Vertices[4] = new Vector3(-1, 1, -1);
mesh.Vertices[5] = new Vector3(1, 1, -1);
mesh.Vertices[6] = new Vector3(1, -1, -1);
mesh.Vertices[7] = new Vector3(-1, -1, -1);
mesh.Faces[0] = { A:0, B:1, C:2 };
mesh.Faces[1] = { A:1, B:2, C:3 };
mesh.Faces[2] = { A:1, B:3, C:6 };
mesh.Faces[3] = { A:1, B:5, C:6 };
mesh.Faces[4] = { A:0, B:1, C:4 };
mesh.Faces[5] = { A:1, B:4, C:5 };
mesh.Faces[6] = { A:2, B:3, C:7 };
mesh.Faces[7] = { A:3, B:6, C:7 };
mesh.Faces[8] = { A:0, B:2, C:7 };
mesh.Faces[9] = { A:0, B:4, C:7 };
mesh.Faces[10] = { A:4, B:5, C:6 };
mesh.Faces[11] = { A:4, B:6, C:7 };
```



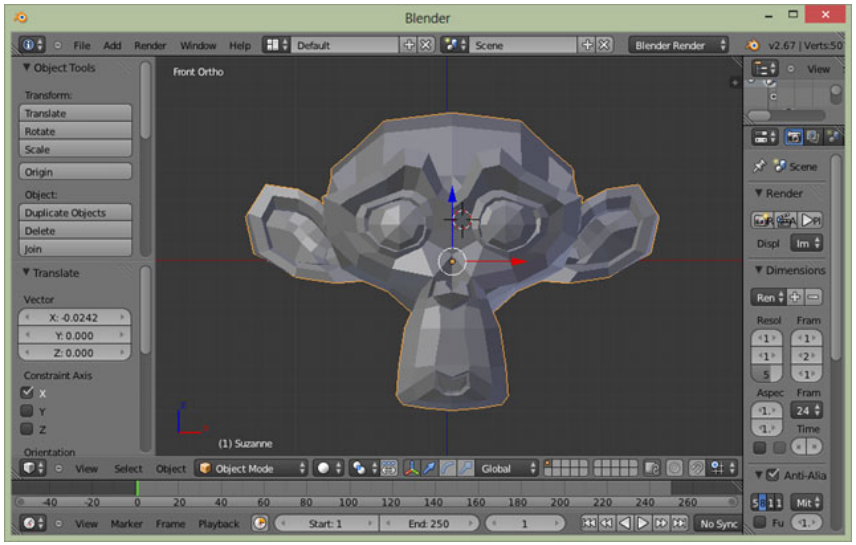
Rasterization et texture

1) Création du modèle avec «Blender»

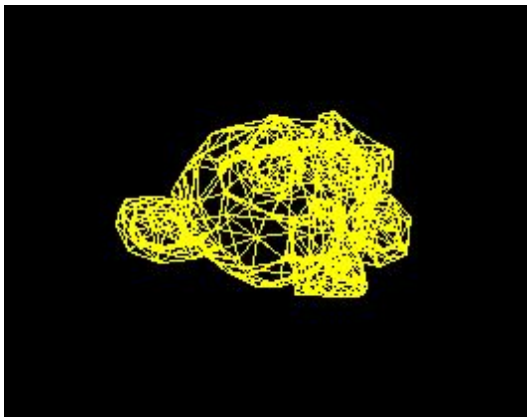


Rasterization et texture

1) Création du modèle avec «Blender»

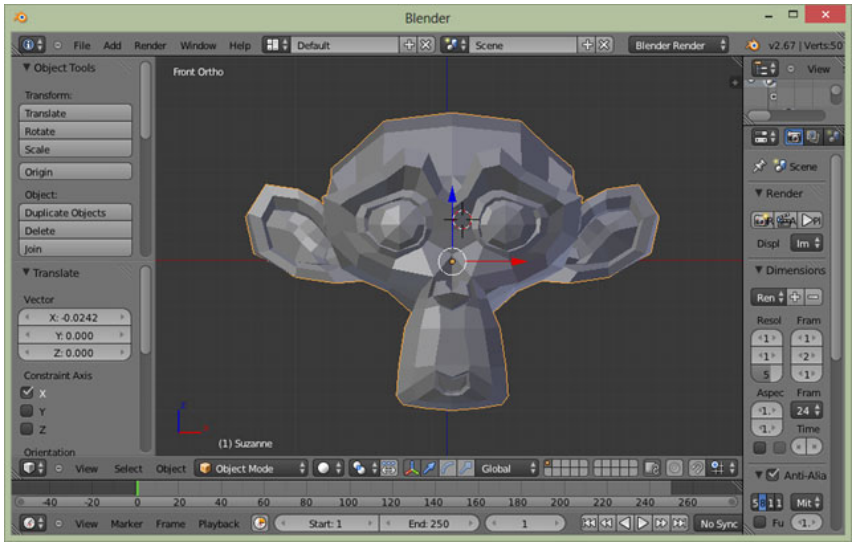


2) Exportation de la géométrie

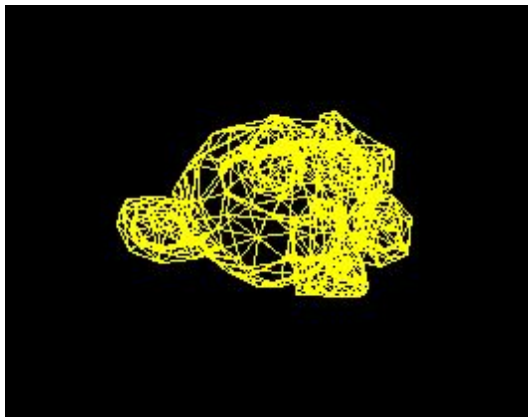


Rasterization et texture

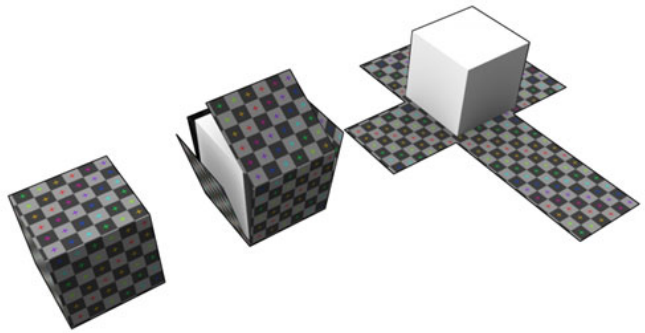
1) Création du modèle avec «Blender»



2) Exportation de la géométrie



3) Ajout de la texture

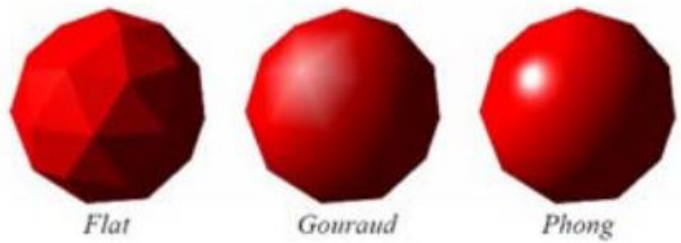


Rasterization et texture

- « ***Rasterization*** » : remplir les triangles.
- ***Éclairage*** : définir la couleur de ces triangles.

  On calcule la normale à la face. Plus l'angle entre cette normale et notre source lumineuse sera grand, moins la face sera éclairée.

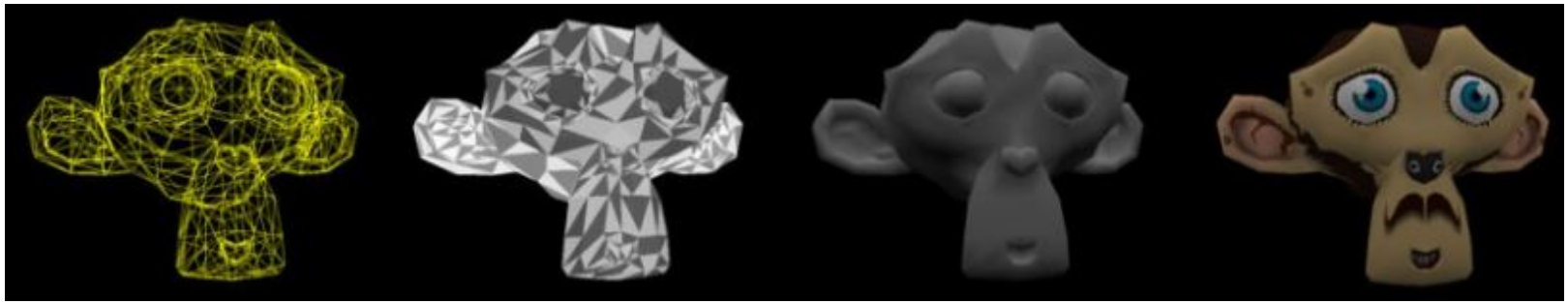
  Il y a différents algorithmes d'éclairage connus depuis des années : « flat » où l'on continue de bien voir les triangles, « gouraud » où l'on ne voit presque plus les triangles avec un dégradé de couleur effectué entre chaque extrémité du triangle et « phong » encore plus gourmand en ressources mais plus précis.



- ***Texture*** : augmenter le réalisme.

Rasterization et texture

Démo :



Exemple 3D dit « *software* »

CPU et GPU

Utilisation de la **CPU** (Central Processing Unit)

CPU et GPU

Utilisation de la **CPU** (Central Processing Unit) $\Longrightarrow$ difficultés d'affichage, de rendu, de stabilité

CPU et GPU

Utilisation de la **CPU** (Central Processing Unit) $\Longrightarrow$ difficultés d'affichage, de rendu, de stabilité

Solution : utiliser la **GPU** (Graphics Processing Unit)

- Afficher un monde 3D tournant à 60 images par seconde (FPS) de manière stable
- Utiliser une technologie particulière : DirectX, OpenGL et il faut développer en C++

CPU et GPU

Utilisation de la **CPU** (Central Processing Unit) $\Longrightarrow$ difficultés d'affichage, de rendu, de stabilité

Solution : utiliser la **GPU** (Graphics Processing Unit)

- Afficher un monde 3D tournant à 60 images par seconde (FPS) de manière stable
- Utiliser une technologie particulière : DirectX, OpenGL et il faut développer en C++

Aide : **WebGL**

- C'est un sous-ensemble d'OpenGL exposant des API (Application Programming Interface) vers le monde JavaScript. WebGL va s'occuper de discuter avec le GPU et de projeter le monde 3D dans le canvas HTML5.
- Avec WebGL un simple navigateur Internet devient un environnement hors du commun pour créer et animer en temps réel des scénes 3D pouvant être à la fois complexes et réalistes.
- Le but de WebGL est d'utiliser l'infographie comme outil aux multiples facettes pour le développement et la conception innovante de sites Web.

Pourquoi WebGL ?

- Outil de visualisation 3D temps-réel
- Développement pur en Infographie 3D adapté au contexte Internet
- API basée sur la norme HTML5 et le langage JavaScript (JS)
- Jeux vidéo, visualisation d'environnements virtuels, simulation
- Gratuit
- Pas de plugin, ni compilateurs, ni librairies annexes
- Multi-OS
- Navigateurs : Firefox 4, Chrome 9, Safari 5.1, Opera 12, Microsoft Internet Explorer(IE)11.

Pourquoi WebGL ?

- Outil de visualisation 3D temps-réel
- Développement pur en Infographie 3D adapté au contexte Internet
- API basée sur la norme HTML5 et le langage JavaScript (JS)
- Jeux vidéo, visualisation d'environnements virtuels, simulation
- Gratuit
- Pas de plugin, ni compilateurs, ni librairies annexes
- Multi-OS
- Navigateurs : Firefox 4, Chrome 9, Safari 5.1, Opera 12, Microsoft Internet Explorer(IE)11.

Pratique, complet, portable, utile et à la pointe de la technologie

Pourquoi WebGL ?

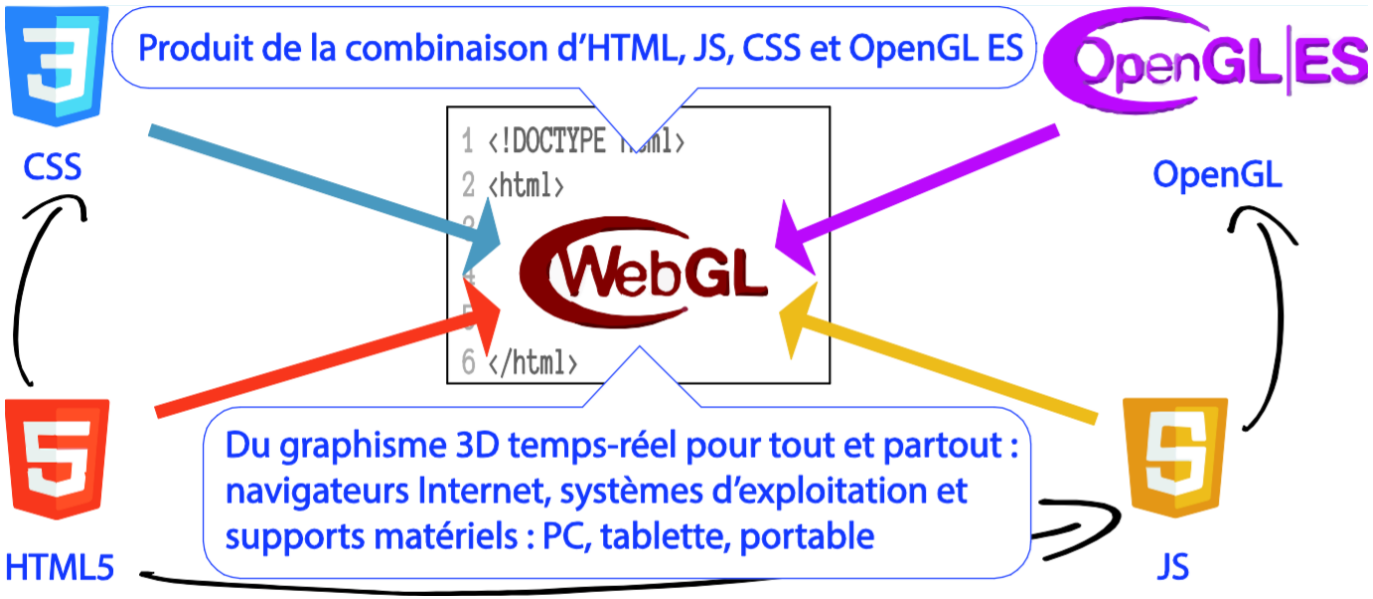
- Outil de visualisation 3D temps-réel
- Développement pur en Infographie 3D adapté au contexte Internet
- API basée sur la norme HTML5 et le langage JavaScript (JS)
- Jeux vidéo, visualisation d'environnements virtuels, simulation
- Gratuit
- Pas de plugin, ni compilateurs, ni librairies annexes
- Multi-OS
- Navigateurs : Firefox 4, Chrome 9, Safari 5.1, Opera 12, Microsoft Internet Explorer(IE)11.

Pratique, complet, portable, utile et à la pointe de la technologie

Contraintes :

- Carte graphique qui permet d'utiliser OpenGL Shading Language (GLSL) $\rightarrow$ Faux problème
- Changement de la façon de programmer (mélange : JS, GLSL, HTML, CSS)
- Certaines incompatibilités d' IE®

D'où vient WebGL ?



Avec WebGL rien n'est précompilé : les parties HTML et JS sont interprétées et le code GLSL est compilé à la volée (*at runtime*) par la carte graphique lors de l'exécution.

WebGL « pur »

WebGL et **langage GLSL** (OpenGL Shading Language) basé sur :

- Le « pixel shader ». C'est un petit morceau de programme qui va dire au GPU comment dessiner chacun des pixels du triangle en fonction de votre propre logique. C'est là-dedans que l'on crée par exemple une équation mathématique simulant l'eau, ses reflets et les vagues.
- Le « vertex shader ». Comme son nom l'indique, il va plutôt travailler sur les extrémités/points du triangle et va plutôt avoir une incidence sur la géométrie.

WebGL « pur »

WebGL et **langage GLSL** (OpenGL Shading Language) basé sur :

- Le « pixel shader ». C'est un petit morceau de programme qui va dire au GPU comment dessiner chacun des pixels du triangle en fonction de votre propre logique. C'est là-dedans que l'on crée par exemple une équation mathématique simulant l'eau, ses reflets et les vagues.
- Le « vertex shader ». Comme son nom l'indique, il va plutôt travailler sur les extrémités/points du triangle et va plutôt avoir une incidence sur la géométrie.

Aide : *moteurs 3D WebGL en JavaScript*,
c'est-à-dire en utilisant de bibliothèques de hauts niveaux permettant un nombre d'abstractions.

ThreeJS

Le moteur 3D **ThreeJS** s'occupe de tout :

- les « shaders » sont déjà pré-écrits et compatibles avec toutes les plateformes ;
- les matrices sont déjà gérées ;
- l'éclairage est déjà calculé ;
- les déplacements à l'aide des contrôles classiques (souris, tactile ou accéléromètre) sont supportés.