Les cartes graphique

**Historique :**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1996 | Génération 1 | 3dfx Voodoo | texture mapping et z-buffer | bus PCI |
| 1998 | Génération 2 | GeForce/Radeon 7500 | GPU effectue Transform&lighting, multi-texting | bus AGP |
| 2001 | Génération 3 | GeForce3/Radeon 8500 | Programmation sur les sommets (vertex shader) |  |
| 2002 | Génération 4 | Radeon 9700/GeForce FX | Premières cartes programmables (fragment shader) |  |
| 2008 | Génération 5 | GeForce9/Radeon HD | Compatibilité OpenGL et DirectX, geometry shader | bus PCIe |

**Composant :**

GPU (Graphical Processing Unit) : Processeur graphique, constituant le cœur de la carte graphique, et qui possède des instructions évoluées de traitement d’image.

Mémoire vidéo (Frame Buffer) : Conserve les images traitées par le GPU avant l’affichage.

RAMDAC (random access memory digital-analog converter) : Convertit les images numériques stockées dans la mémoire video en signaux numériques à envoyer à l’écran.

**Caractéristique :**

Un GPU est une unité de calcul massivement parallèle.

Différence CPU-GPU

**GPU**

GPGPU : parallélise les calculs (SIMD).

Laisse le CPU libre (indépendance).

Framebuffer non accessible, et rien n’est modifiable en dehors des variables locales et des valeurs de sorties.

Programme CPU :

* Traite l’ensemble des données.
* Données accessibles sans contraintes.
* Format de sortie libre.

Programme GPU

* Transforme un flux de données en un autre.
* Format imposé en entrée et en sortie.

Conclusion : Pour que l’utilisation d’un GPU soit utile, il faut que le GPU ait suffisamment de core, et que le nombre de calculs soit élevé et parallélisable.

Les pipelines graphiques

Un pipeline (tuyau) graphique est une succession de tâches réalisées par la carte graphique nécessaires à la création de données géométriques (maillages et textures par exemple) afin de produire une scène 3D.

En entrée, il récupère des coordonnées de points, et en sortie, il renvoi des Texels (TExtured piXEL). Le **texel** est un ou plusieurs pixels, sur les quelles, une texture a été appliquée.

Le pipeline a trois phases :

• Phase 1 : transformations et calculs géométriques sur les sommets.

• Phase 2 : calcul du rendu local et texturage des pixels de la facette.

• Phase 3 : construction et rendu de l'image finale.

# Historique du pipeline - Evolution

1998/1999 : La première génération (rasterisation + texture mapping par le GPU)

1999/2000 : La deuxieme génération (Transform & lightin par le GPU)

2001/2002 : Troisième génération (Vertex shader par le GPU)

2003 : Quatrième génération (Pixel shader par le GPU)

# Deux sortes de pipeline : Fixe (FFP) et dynamique (PFP)

Le pipeline fixe ne contient pas de shader. Il n’est donc pas programmable par les développeurs, mais est assez optimisé pour les calculs de toutes les étapes du pipeline.

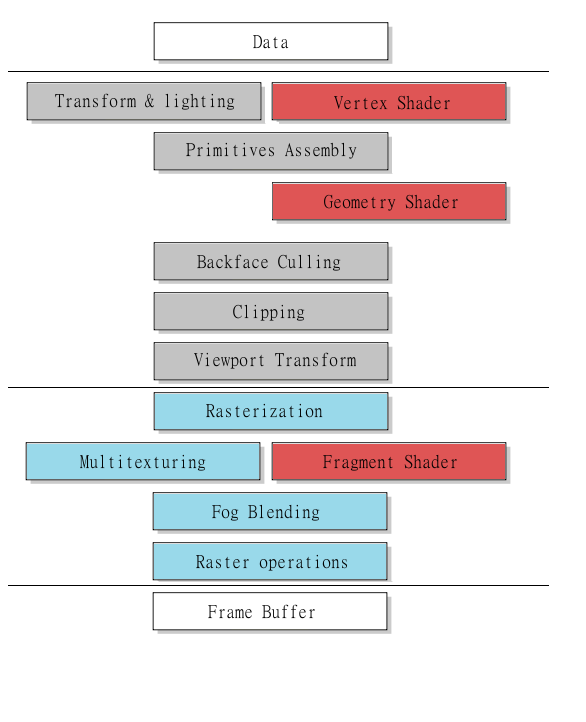
Au contraire, le pipeline dynamique, et ses shaders nous permettent de définir exactement le rendu à l’écran désiré. La création d’algorithmes qui diffèrent de ceux contenus dans le pipeline fixe, permet des changements au niveau de certains paramètres comme par exemple, les contrastes, les ombres, la lumière, les effets de Cell Shading ou de bump mapping…

Les deux étapes programmables sont donc **le vertex shader** et **le fragment shader** ou **pixel shader**.

Un vertex ou vertice, est un sommet d’une figure géométrique (un point particulier d’une figure).

# Déroulement du pipeline

## Schéma :



## Les étapes :

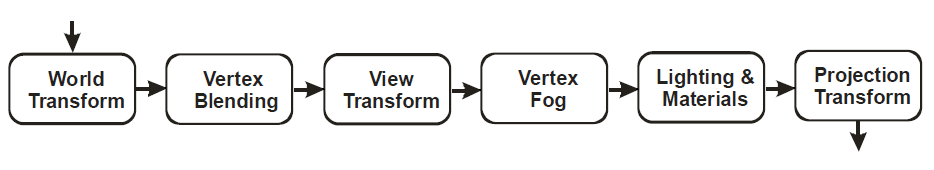
### Data (Données brutes) :

Définition des vertices ainsi que leurs coordonnées. Ces données sont enregistrées dans un tableau de vertex (**Vertex Buffer**) sur le GPU. L’**input assembler** est le circuit qui va charger les vertices dans le pipeline à partir de l’adresse de départ du tableau. A la lecture de chaque vertice, celle-ci est passe dans le **Vertex Cache**. Le mémoire cache étant plus rapide, lors des utilisations ultérieures des vertices, la lecture sera plus rapide.

### Transform & lighting :

Transformation, positionnement et éclairage des sommets en passant du **repère local** au **repère global** puis dans **l’espace projeté**.

Cette tâche peut se décomposée en plusieurs tâches :



**En entrée** : tableaux de coordonnées de sommets dans le repère de l’objet.

**World Transform :**

Le repère local est le repère affecté à chaque objet. Chaque vertices de l’objet est localisé par rapport au centre de l’objet de coordonnées (0, 0, 0).

Le passage au repère global qui est le repère de la caméra, ou de l’observateur, s’exécute en changeant les coordonnées de chaque objet, qui passe donc de (0, 0, 0) à (X, Y, Z). Chaque vertice de chaque objet se met à jour en fonction des coordonnées X, Y et Z.

Il faut que l’objet soit dans la bonne orientation, et qu’il soit au bon endroit. Cela peut nécessiter une translation, une rotation ou une mise à l’échelle. Ces opérations peuvent être effectuées sur chaque vertex, c’est-à-dire sur chaque vecteur (coordonnées X, Y et Z). Les calculs peuvent donc être parallélisés et sont effectués par le GPU. Les calculs de transformation, rotation, et de mise à l’échelle sont des multiplications de ces vecteurs par des matrices prédéfinies. Chaque opération a une matrice de multiplication prédéfinie, et des matrices existent aussi pour effectuer deux opérations en même temps.

**Vertex Blending :**

Combinaison d’un ou plusieurs ensembles de sommets.

**View Transform :**

Passage du repère global au repère de la caméra. Après cette transformation, le point de coordonnées (0, 0, 0) sera la caméra. La direction de la vue de l’observateur sera alignée avec l'axe de la profondeur (l'axe Z). La transformation est opérée par le même principe que le « World Transform ».

**Vertex Fog :**

Calcul de la couleur du brouillard pour chaque sommet.

**Lighting & Materials :**

Chaque vertice fournie des couleurs RGB pour définir comment cette vertice réagie à la lumière (réflexion, teinte, contraste…). On attribut alors à la vertice une couleur RGB correspondant à son éclairage.

**Projection Transform :**

Passage du repère de l'observateur à l'espace projeté.

**En sortie** : sommets avec calculs d’illumination dans l’espace projeté.

### Primitive assembly ou Tesselation :

Assemblage des vertex sous formes de triangle. L’assemblage s’effectue en prenant les vertex dans l’ordre spécifié par le tableau de vertex.

Il existe trois méthodes d’assemblage :

* Chaque paquet de trois vertex est un triangle indépendant.
* Les deux derniers vertex de chaque triangle sont les deux premiers vertex d’un autre triangle (bande de triangle ou triangle strip).
* Le premier élément est relié à chaque paire d’élément suivant (ventilateur de triangle ou triangle fan).

Décomposer des formes complexes, en formes géométriques simple

### Backface Culling :

Supprime de l’affichage les triangles qui tournent le dos à la caméra (face arrière d’une face) en effectuant un calcul avec sa normale.

### Clipping :

Découpe et supprime de l’affichage les parties non visibles des objets partiellement visibles.

### Viewport transform :

Supprime les parties qui ne sont pas dans les coordonnées de l’écran.

### Rasterization :

La rastérisation, ou matricialisation, est composée du **triangle setup** et de **l’interpolation des pixels**.

**Triangle setup ou pixellisation des triangles** :

Utilisation de la fonction des contours : Renvoi d’un nombre entier (-1, 0, 1) pour chaque pixel en fonction d’une droite. D’un côté de la droite, -1, de l’autre, 1 et sur la droite 0.

En appliquant cette fonction aux trois segments, nous avons :

* A l'intérieur du triangle, les trois fonctions (une par côté) donneront un résultat positif.
* A l'extérieur, une des trois fonctions donnera un résultat négatif.

Si les 3 résultats des fonctions de contours pour un pixel sont positifs, cela veut dire que le pixel appartient au triangle.

L’optimisation de ce procédé est déterminer le plus petit rectangle qui contient le triangle testé, pour exécuter le test des contours seulement sur un nombre de pixels réduit, et non sur tous les pixels de la forme.

Les tests des pixels sont exécutés parallèlement pour un gain de temps.

**Interpolation** :

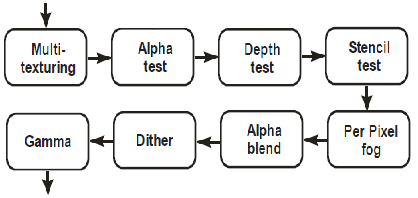
Chaque vertex a reçu divers paramètre comme la couleur, la profondeur. Chaque fragment de chaque triangle, va recevoir en attributs sa couleur, sa profondeur, sa position à l’écran, une valeur de stencil, une transparence, par interpolations des trois vertices définis précédemment.

### Fog Blending :

Il est rare qu’un pixel corresponde à un seul fragment. En général, il y a toujours plusieurs triangles les un sous les autres, qui doivent être mélangés pour donner le pixel final. C’est ce qu’exécute la tâche de « Fog Blending ».

### Raster Operations (ROP):

Cette tâche peut se décomposée en plusieurs tâches :



En entrée : Tableau de fragment

**Multitexturing :**

Composition des textures et des couleurs issues de l'illumination.

**Alpha Test :**

C’est le test de transparence. Certaines textures ou couleurs pouvant être transparentes, il faut pouvoir lire le **canal alpha** d’un pixel. Cette étape supprime tous les pixels qui n’ont pas un alpha acceptable.

**Depth Test :**

C’est le test de visibilité (profondeur). C’est la coordonnée z de chaque fragment qui est comparée avec la coordonnée z des autres fragments qui sont sur le même pixel. La carte graphique utilise un **depth-buffer** qui est un tableau stocké en mémoire. Ce tableau va stocker pour chaque pixel la coordonnée z de l’objet le plus proche de l’écran.

C’est le circuit de gestion de la profondeur qui s’occupe de mettre à jour le depth-buffer. Il va récupérer les coordonnées du fragment reçu à l’écran, puis lire en mémoire la coordonnée z correspondante dans le tableau. Il va comparer celle-ci avec la coordonnée z du fragment reçu, et décider ou non de mettre à jour le depth-buffer et le frame-buffer.

Il faut que ces coordonnées z soient codées sur assez de bits pour avoir une bonne précision, et ne pas se retrouver avec des artefacts visuels.

**Stencil Test :**

Un stencil est en français un pochoir. C’est une sorte de masque que l’on affiche sur l’écran pour définir une vue restreinte comme une vue à travers un hublot, ou une serrure.

**Per-pixel Fog :**

Cette étape permet de rajouter du brouillard sur chaque pixel. Le brouillard est ajouté grâce à une couleur de brouillard, qui est mélangée avec la couleur du pixel (moyenne). Le brouillard dépend de la profondeur du pixel. Un brouillard de base est défini, et s’affaiblie ou s’amplifie selon la distance. Si l’objet est proche, aucun brouillard n’est appliqué, si l’objet est trop loin, c’est seulement la couleur du brouillard qui est affiché.

**Alpha Blend :**

Cette tâche sert à calculer la couleur finale du pixel. Notre GPU contient un color buffer, un tableau qui sert à stocker pour chaque pixel sa couleur finale.

Le calcul est simple, à chaque fragment envoyé :

* Lecture de l’ancienne couleur du pixel,
* Calcul de la couleur finale en fonction du fragment envoyé et de l’ancienne couleur,
* Enregistrement du résultat.

Les opérations « alpha test » et « alpha blend » sont effectuées par un circuit spécialisé : le Color ROP. Il travaille en parallèle des autres tâches.

**Dither :**

Mélange des couleurs des pixels adjacents pour obtenir une couleur plus consistante.

**Gamma :**

Applique la correction gamma définie.

**En sortie** : le pixel prêt à être affiché relativement à l'état courant de traitement du flux de sommets.

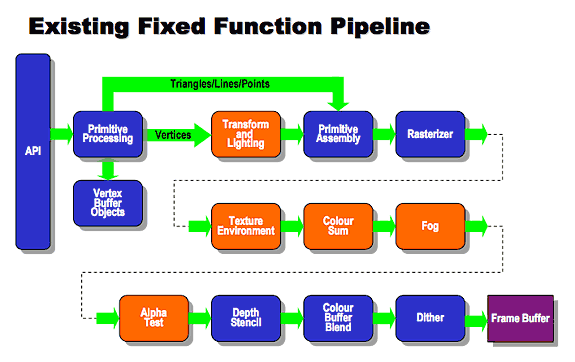
### Framebuffer :

Affichage du résultat à l’écran utilisant le double-buffering, color buffer, Z-buffer.

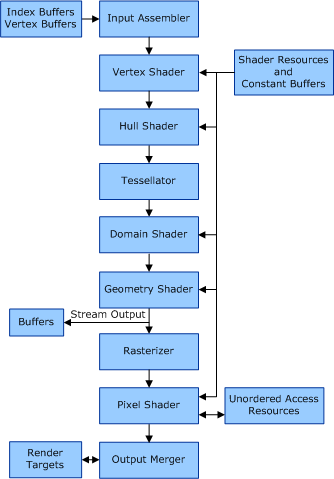
# Différents pipelines selon OpenGL, DirectX

OpenGL :

OpenGL envoi au vertex shader des vertex buffer (tableaux d’attributs par vertex).



Direct3D :



Les Shaders

# Définition :

Les shaders sont des programmes plus ou moins élaborés qui ne sont pas pris en charge par le CPU (processeur), mais par le GPU (carte graphique). Exécutés sur chaque primitive d’une forme, ils permettent un affichage calculé, de pixels en couleur pour un rendu graphique. Les paramètres de ces couleurs sont : la teinte, la luminance, la saturation.

# Historique :

Du point de vue des bibliothèques graphiques, les shaders ont été intégrés à partir d'OpenGL 2.0 et dans DirectX 8.

Pour chaque version différente de shader, nous parlons de modèles de shader (shader models (SM)). Il est possible de préciser la version que nous voulons utiliser dans notre shader.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Année | Version des shaders | DirectX | OpenGL | ATI | Nvidia | Info |
| 2001 | SM 1.x | DirectX8 | OpenGL 2.0 | Radeon R200 | Geforce séries 3 |  |
| 2003 | SM 2.x | DirectX9.0 |  | Radeon R300 / R420 | Geforce FX |  |
| 2004 | SM 3.0 | DirectX9.0c |  | Radeon R520 | Geforce séries 6 / 7 |  |
| 2006 | SM 4.0 | DirectX10 | OpenGL 3.2 | Radeon R600 | Geforce séries 200 / 300 | Apparition des Geometry Shader |
| 2009 | SM 5.0 | DirectX11 | OpenGL 4.1 | Radeon R800 | Geforce séries 400 | Tesselation |

# Fonctionnement :

## Vertex Shader :

Le vertex shader est exécuté pour chaque vertex (sommet). Il remplace la tâche dite de **Transform & lighting**. Elle doit dans un premier temps calculer la projection de coordonnées des sommets à partir de l'espace 3D dans l'espace écran et dans un second temps, définir l’éclairage pour chaque sommet. La projection des formes est le plus souvent décomposée en triangles, car les triangles permettent en remplissage des surfaces complexes sans pertes.

Le vertex shader reçoit en entrée un sommet, un ensemble de paramètres constants.

Le vertex shader renvoi en sortie le même sommet qu’en entrée, mais transformé, avec des attributs spéciaux comme ses coordonnées dans l’espace écran (obligatoire), et des attributs interpolés comme la couleur (obligatoire), la texture (facultatif)…

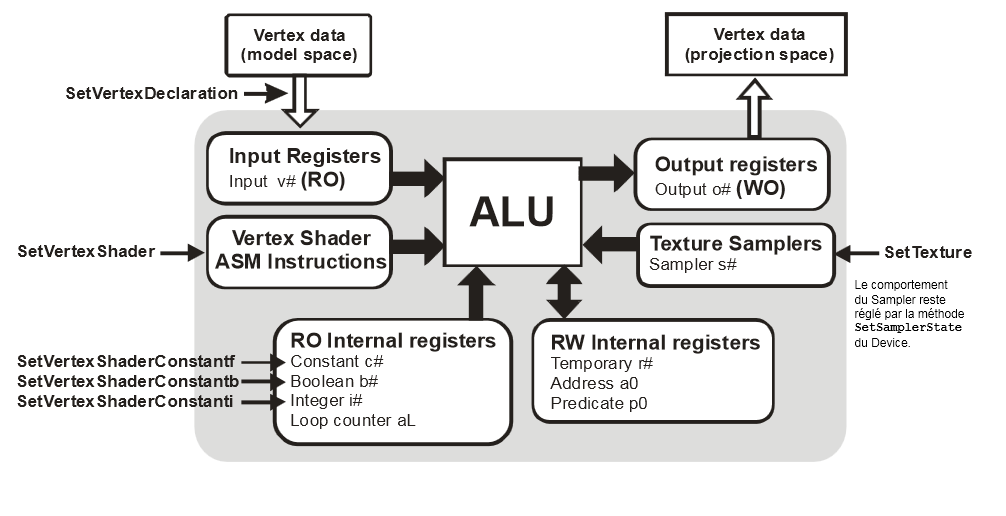
#### Possibilités du vertex shaders :

* Position procédurale
* Mélange de transformations (‘Vertex Blend’) -> arrondis des coins plus propres
* Mélange de maillages (‘Morphing’)
* Déformation du maillage (‘Vertex Deformation’) -> changer quelques points dans le maillage pour plus de réalités.
* Eclairage par sommet (‘Per Vertex Lighting’)
* Tissu, peau, interpolation, displacement maps…
* Coordonnées de texture
* Brouillard particulier
* Taille d’affichage d’un point

**Transform** est la tâche qui permet de convertir des données spatiales depuis un espace virtuel en 3D vers l'écran en 2D.

**Lighting** est la 3e opération qui va permettre d'éclairer la scène : éclairage des objets dans la scène 3D, calcul de l'interaction des composantes de la lumière sur les objets puis envoi de la scène ainsi calculée à l'écran.

**Architecture :**



ALU (Arithmetic logic unit) : C’est l’unité qui effectue les calculs, grâce aux variables contenues dans les registres.

Description des registres de la structure :

## Fragment Shader :

Un fragment est un pixel qui n’a pas encore été traité. Ses attributs sont définis par le vertex shader, et interpolés par la rastérisation.

Le fragment shader ou pixel shader est exécuté pour chaque pixel de la forme délimitée par les vertex. Il remplace la tâche dite de « multitexturing ». Son rôle étant de définir la couleur ou la texture de chaque pixel en fonction des couleurs, de l’éclairage et de la profondeur de chaque vertex de la forme.

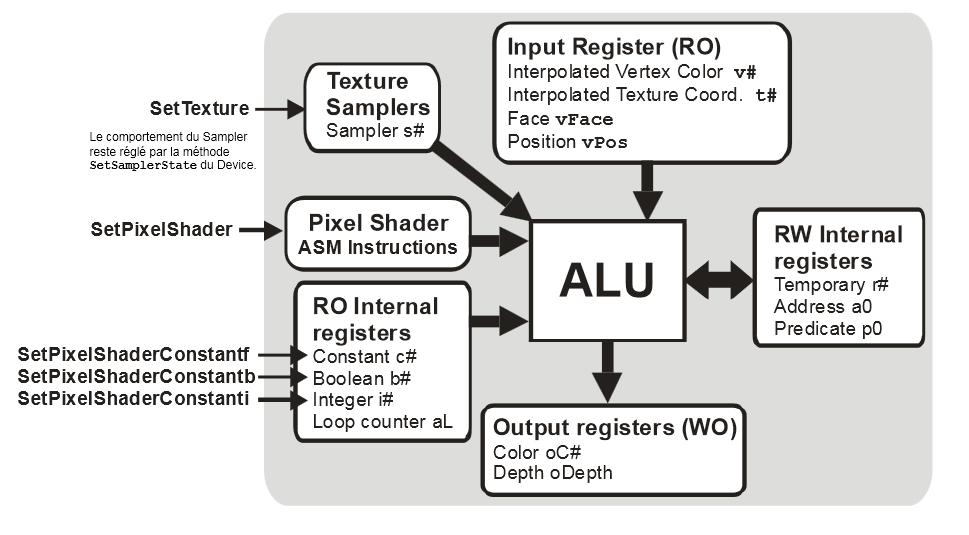
Le fragment shader reçoit en entrée un fragment et ses attributs résultant de l’interpolation des sommets comme sa couleur, ses coordonnées.

Le fragment shader renvoi en sortie le fragment d’entrée transformé en pixel, avec sa profondeur, sa couleur, sa transparence, ou aucun pixel (suppression de pixel).

#### Possibilités du fragment shaders :

* Réflexion par pixel
* Illumination par pixel (phong, BRDF) -> plus de dégradé
* Textures procédurales
* Cartoon shading
* nouveaux modèles d’ombrages (par pixel, géomètrie interpolée).
* textures procédurales ou dynamiques.
* bump-mapping

**Architecture :**



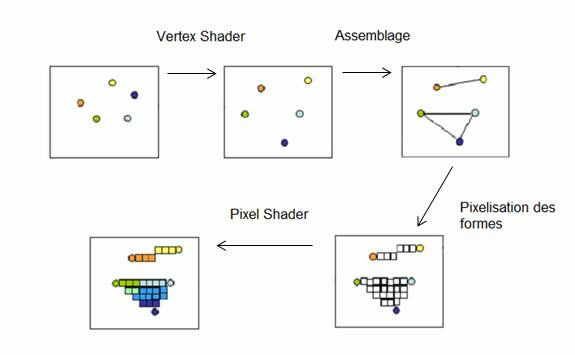
#### G:\Projet\images\pipeline Direct3D.pngG:\Projet\images\gl_pipeline.gifExemple d’algorithmes applicables :

|  |  |  |
| --- | --- | --- |
| **Anglais** | **Français** | **Utilisation** |
| Cel-Shading | ombrage de celluloïd | éclairage non photoréaliste |
| **Illumination locale** |  |  |
| Bump mapping | placage de relief |  |
| Normal mapping |  | vecteur normale à chaque pixel |
| Displacement mapping |  | texture (height map) |
| Parallax mapping |  | utilise le texel et normal mapping |
| Photon mapping |  | réfraction de la lumière **voir LuxRender** |
| Environnemental mapping |  | texture avec effet mirroir |
| **Illumination globale** |  |  |
| Ray-tracing | lancement de rayon | utilise le parcours inverse de la lumière |
|  | Radiosité | Radiance |
| cone tracing, beam tracing, path tracing |  | Ray-tracing |
| **Illumination locale** |  |  |
| Flat shading | Ombrage plat | Eclairage proportionnel à l’angle entre la normal à la face et la direction de la lumière |
| Gourand shading | Ombrage de Gouraud | interpolation luminosité entre 3 sommets d’un polygone |
| Phong shading | Ombrage Phong | modèle d’illumination Phong, interpolation Phong, illumination local (la lumière ambiante, la lumière diffuse (modèle lambertien) et la lumière spéculaire) |
|  |  |  |
| Alpha blending | simulation de transparance | couche alpha |

## Geometry shader :

Créer de nouvelles formes géométriques.

## Schéma simple du fonctionnement :



# Utilisation et langages:

Les GPUs peuvent être programmés grâce au langage dit « Shading language ». C’est un langage de description des interactions simples ou complexes, entre la lumière et la matière.

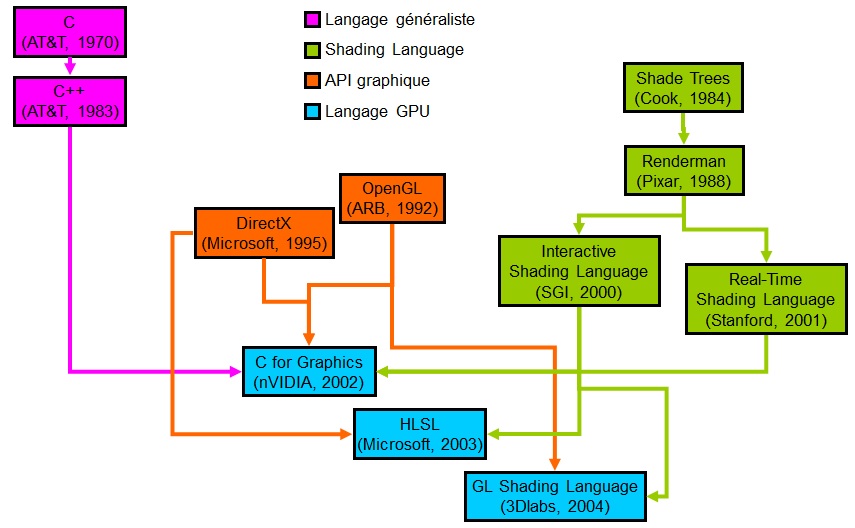
GLSL (OpenGL Shading Language) :

* Intégré à OpenGL.
* Plus simple que Cg.
* OpenGL se charge d’inclure les shaders.
* Multi-platforme (IOS, Windows, Android…).
* Exclusif à OpenGL.
* Peu d’outils additionnels.
* Support moindre que HLSL ou Cg.

HSLS : se charge d’inclure les shaders.

Cg (C for graphics) :

* Indépendant de l’API 3d (Direct3D ou OpenGL).
* Outil débogueur
* Le plus utilisé
* Support important
* Windows uniquement



# Application à notre projet :

Glossaire

Texture :

Une **texture** est une image en deux dimensions (2D) que l'on va appliquer sur une surface (2D) ou un volume en trois dimensions (3D) de manière à habiller cette surface ou ce volume.

Maillage :

Une **maille**, ou **mesh**, est un objet tridimensionnel constitué de sommets, d'arêtes et de faces organisés en polygones sous forme de fil de fer dans une infographie tridimensionnelle. Les faces se composent généralement de triangles, de quadrilatères ou d'autres polygones convexes simples, car cela simplifie le rendu.

Tampon :

Une **mémoire tampon**, ou **buffer**, est une zone de mémoire vive ou de disque utilisée pour entreposer temporairement des données en attendant leur traitement.

<http://progdupeu.pl/tutoriels/29/les-cartes-graphiques/pipeline-non-programmable/unites-de-texture/>

<http://oivdoc90.vsg3d.com/content/201-general-concepts>

<http://fr.wikipedia.org/wiki/Z-buffer>

<http://www-evasion.imag.fr/Membres/Antoine.Bouthors/teaching/opengl/>

http://gl.ict.usc.edu/Research/SpecScanning/