Introduction au rendu 3D Temps Réel

Cours Objectif 3D
Frédéric Mizac - 2019

Notes

Ici, on va considérer que DirectX est installé correctement.

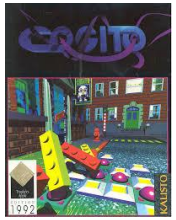
Présentation

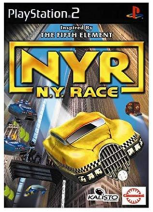
Frédéric Mizac - Développeur de Jeux Vidéos
fred@mizac.net

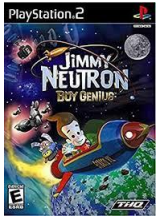
https://www.linkedin.com/in/fredericmizac/

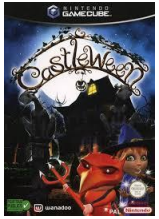
Parcours professionnel:

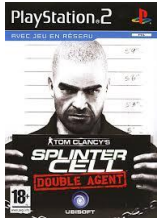
Quelques titres

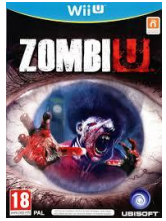


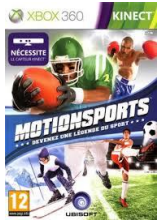












Quelques rappels

- Aucune question n’est mauvaise!
- N’hésitez pas à poser des questions !
- Si je ne suis pas assez clair, dites le :)

Introduction au rendu 3D Temps réel

Nous allons dans un premier temps nous intéresser à l'historique de l'affichage dans le monde PC et voir comment fonctionner l'affichage 2D. Pourquoi, parce qu'il n'y a pas de 3D sans 2D :)

Nous allons ensuite aborder les différentes API et les pipelines graphiques principaux.

Puis nous allons nous intéresser aux diverses techniques et découvrir de nouveaux mots!

Nous terminerons pas un exemple complet qui va afficher… quelque chose de simple pour commencer.

Plan du cours

- Historique de l'affichage sur ordinateur
- Fonctionnement des cartes 2D
- La mémoire
- Les premières cartes 3D
- Les différentes générations de hardware
- Rapports de puissance entre les cartes graphiques
- Parlons Rendu 3D Maintenant
- Le système de coordonnées
- La caméra
- Les Principaux flux de données
- Mémoire Centrale vs Mémoire graphique
- Occlusion et Visibilité
- Les artefacts de Rendu
- Le pipeline jusqu'à l'écran
- Maintenant, côté Programmation
- Les principales API 3D
- Occlusion et Visibilité
- Les principales passes de rendu
- L'envoi à la carte graphique
- L'éclairage
- Les matériaux et les textures
- Les coordonnées de textures
- Vocabulaire et définitions
- Les Shaders
- Enfin du code, un exemple
- Le Debugger
- Conclusion

Historique de l’affichage sur ordinateur

Les ordinateurs ont d’abord été capable d’afficher de la 2D pure, c’est à dire des pixels.

Le premier standard graphique a été le CGA, puis il y a eu EGA, puis le VGA et puis plein d’autres régulièrement, jusqu’à l’arrivée des premières cartes graphiques.

Quelques chiffres :

CGA (Color Graphics Adapter) : (https://fr.wikipedia.org/wiki/Color_Graphics_Adapter)

- Apparu en 1981.
- Affichage en 320x200 en 2 bits par couleurs (4 couleurs)
- Affichage en 640x200 en 1 bit par couleurs (2 couleurs)
- Taille de la ram video : 16 ko (pourquoi ?)
- Ram video à l’adresse B800:0000

Historique de l’affichage sur ordinateur

VGA (Video Graphics Array) :

- Apparu en 1987.
- 256 Ko de ram video !
- Affichage en 320 x 200 en 8 bits par couleurs (256 couleurs parmi 262 144)
- Affichage en 640 x 480 en 4 bits par couleurs (16 couleurs palette)

Historique de l’affichage sur ordinateur

VGA (Enhanced Graphics Adapter) :

- Apparu en 1984.
- Affichage en 320x200 en 4 bits par couleurs (16 couleurs fixes)
- Affichage en 640x350 en 4 bits par couleurs (16 couleurs parmi 64)
- Taille de la ram video : Alors, combien environ d’après vous ?

L’affichage 2D

Comment fonctionne l’affichage 2D d’après vous ?

L’affichage 2D

On écrit directement dans la RAM video, à la bonne adresse mémoire, avec le bon format.
Le format dépend du mode d’affichage choisi. Le mode donne la résolution de l’écran (x par y) et la profondeur des pixels. Profondeur = Nombres de bits par pixel. 4 bits => 16 couleurs.
Sur ces cartes, il n’y avait qu’un mode palette, c’est à dire un index vers une couleur définie dans une palette de couleur. Soit cette palette était fixe (prédéfinie donc), soit cette palette était choisie.
On pouvait donc ‘animer’ par exemple les couleurs d’une image en modifiant la palette utilisée sans toucher aux valeurs des pixels.

Il fallait voir l’écran comme un tableau de x par y pixel, pixel 0 en coin haut / gauche.
Le jeu préparait l’image à afficher dans un buffer (par exemple) et à la fin de la trame, le buffer était récopié dans la ram video. Il fallait faire cette opération à la fin de la trame vidéo, pour ne pas que cela se voit, la fin de la trame correspond à ce que l’on appelle la VBL (Vertical Blank Line), quand le canon à électron du moniteur passait du bas de l’écran à droite au haut de l’écran à gauche. (Historiquement).

L’affichage 2D

Si on écrivait pas assez vite ou au mauvais moment dans le buffer de la carte graphique, alors on pouvait voir cela à l’écran : c’est la fameuse bande que l’on peut toujours voir si on désactive la VBL de certains jeux (ça dépend du jeu, de la graphique et du moniteur actuellement), mais le principe reste le même.
Vous avez compris pourquoi on voit une bande ou ligne ?

On peut aussi noter que l’affichage palette a totalement disparu, de nos jours, on parle maintenant uniquement des couleurs réelles (True Color) par opposition aux palettes.

Des questions ?

La mémoire

Nous avons vu que l'ancienne génération de carte graphique permettait d'écrire directement dans la ram video, à une adresse préétablie par le constructeur et connue de tous.
Cette époque est révolue.

On parle maintenant de mémoire protégée. On alloue de la mémoire, qui est accessible grâce à un pointeur, mais ce pointeur ne correspond pas à l'adresse physique de la mémoire, c'est une indirection. Cette indirection est résolue au niveau du CPU.

De la même façon, la mémoire des cartes graphiques n'est pas directement disponible aux applications. Il faut passer par une API qui va permettre de transférer vos données de la mémoire centrale (RAM) de l'ordinateur à la mémoire vidéo (VRAM).
C'est d'ailleurs indispensable car la carte possède son propre CPU, appelé GPU.

Les premières cartes 3D

Dans les années 1995, plusieurs cartes graphiques ont vu le jour. Dans les plus célèbres, il y a notamment la carte 3DFx Voodoo, avec son API propriétaire Glide.

Il faut savoir que chaque carte avait sa propre API propriétaire, c'était AVANT l'ère DirectX sur PC et c'était le début d'openGL (Glide était d'ailleurs un dérivé simplifié d'openGL).

La société 3dFX a ensuite été rachetée par NVidia en 2000.

Les différentes générations de hardware

Il y a eu de nombreuses cartes graphiques, toutes plus évoluées les unes que les autres. Les standards de programmation graphiques (sur PC) sont apparus (DirectX et OpenGL).
Cette première génération de cartes graphiques possède ce que l'on appelle un pipeline de rendu fixe.
C'est à dire que chaque carte, selon ces capacités, était capable de faire un certain nombre de choses d'une façon prédéterminée par le constructeur.
C'est l'époque d'avant la programmation par shader. Il fallait utiliser différents trucs et astuces pour réaliser des effets sophistiqués sur ces cartes. Le rendu entre deux cartes différentes pouvait d'ailleurs varier beaucoup.

Actuellement, toutes les cartes graphiques, y compris celles de vos téléphones (dès un iPhone 4S), utilisent un pipeline de rendu programmable, donc des shaders.

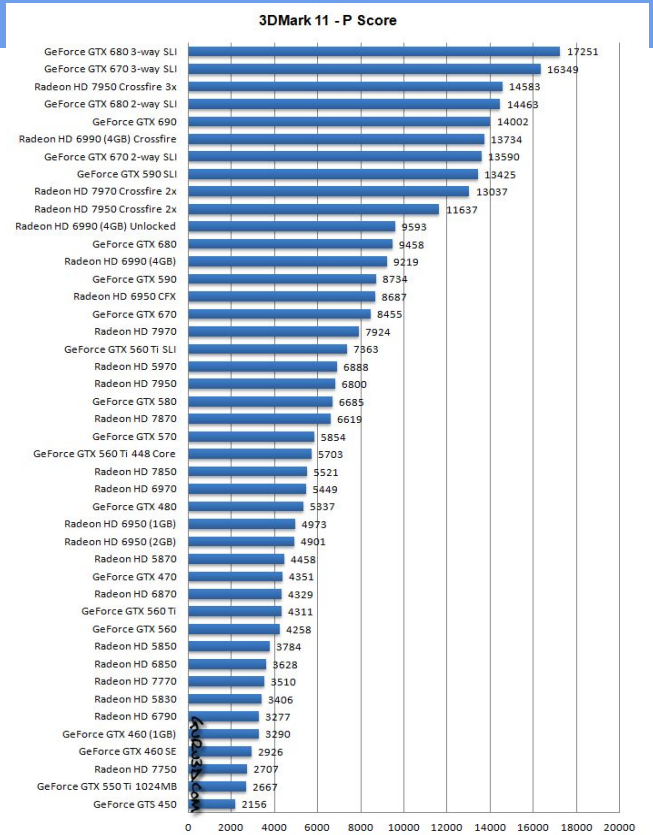
Le rapport de puissance entre les cartes

Comme vous le savez les cartes graphiques peuvent être plus ou moins puissantes, voici un graphique regroupant les principales cartes. On voit qu'il y a un monde entre différentes cartes pourtant pas si éloignées que cela dans le temps.

Cela explique la difficulté que l'on peut avoir à faire un jeu PC qui s'adapte à toutes les cartes avec des performances acceptables.

Notez qu'il peut être compliqué de bencher une carte correctement, certaines mesures sont plus fiables que d'autres :)

Le rapport de puissance entre les cartes



Parlons rendu maintenant

Nous avons déjà vu plein de choses, mais comment cela fonctionne t'il vraiment ?

Comment sont décrit les objets 3D ? Dans quel système de coordonnées ?

Le système de coordonnées

Votre jeu a un système de coordonnées qui lui est propre, mais selon l'api 3D que vous utilisez, vous allez devoir utiliser un système qui conviennent à l'API.

Il y a donc notamment le Système Main-Gauche et le système Main Droite.

DirectX est traditionnellement Main Gauche (left handed), quand OpenGL est main droite (right handed).

Il faut décider duquel vous allez utiliser et faire tous vos maths de la même façon :)

La Caméra

Pour faire un rendu en 3D, contrairement à la 2D, il y a la notion de caméra, c'est à dire un oeil qui va regarder dans une direction. Cet oeil 'ne voit rien' qui soit trop près de lui, c'est le plan de clipping near et ne voit pas plus d'une certaine distance (c'est le plan de clip far).

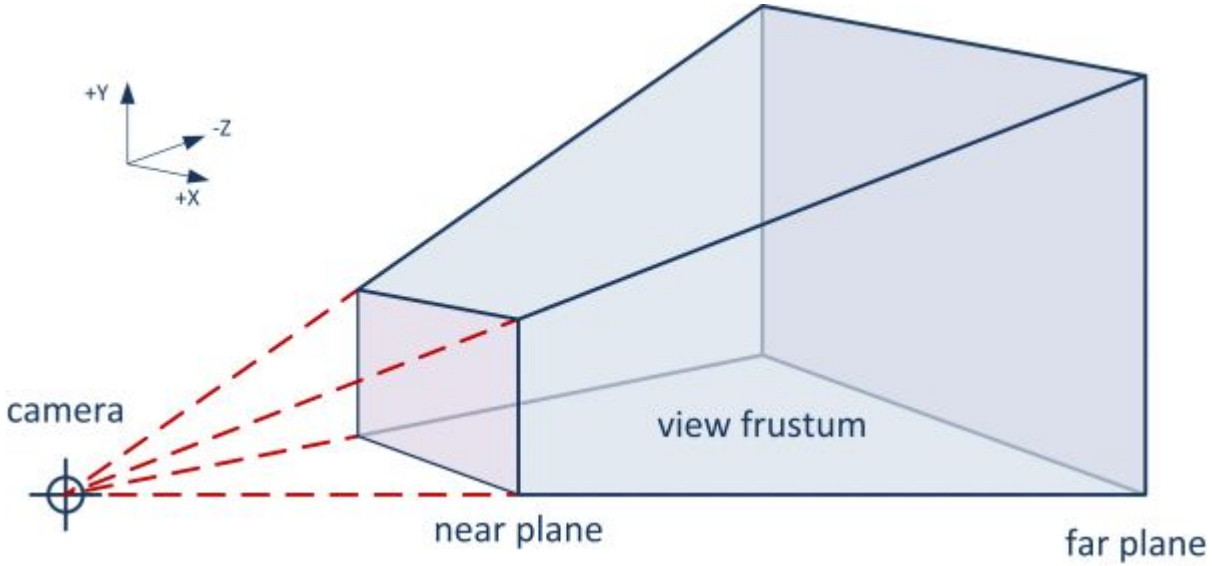
Corollaire : Plus le plan de clip est loin, plus on va donc calculer pour voir ce qui doit être affiché.

Cet oeil a aussi un certain angle d'ouverture horizontal et vertical, c'est la focale (Field of View).

Cet ensemble se nomme la pyramide de vision ou le viewing (ou view) frustum.

Le viewing frustum

Un petit dessin pour mieux comprendre :



La Camera

On voit donc que le rôle de la caméra est bien de déterminer ce que l'on va voir est aussi comment on va le voir. Car il existe plusieurs types de projections de caméra:

- Caméra Orthographique : la caméra orthographique ne corrige pas la position des points sur la distance par rapport à l'oeil. Elle est surtout utilisée dans les applications de type rendu de construction.
- Caméra Perspective : La caméra perspective est la caméra la plus utilisée dans les jeux, elle reprend le principe de l'oeil humain. Plus un point est éloigné de l'oeil, plus il apparaît distant (petit), c'est ce que l'on appelle la perspective.

ref : https://en.wikipedia.org/wiki/3D_projection

Orthographique ou perspective ?

Orthographic Projection

Perspective Projection

Les principaux flux de données lors du rendu

Voyons maintenant une illustration des différents types de pipeline et des flux de données associés.

Mémoire centrale vs Mémoire Graphique

Vous savez tous que votre PC possède une certaine quantité de mémoire et que la carte graphique de votre machine aussi.

Pourquoi 2 types de mémoire ?

Est ce qu’on additionne juste les 2 pour faire plus ?

Mémoire centrale vs Mémoire Graphique

La RAM graphique est dédiée à la carte graphique ! Elle a besoin de celle-ci pour fonctionner. Le rendu et tous les éléments contribuant à l'affichage DOIVENT pas passer par la mémoire graphique.
Le GPU ne PEUT pas accéder à la mémoire centrale du PC (et réciproquement).

On doit faire des transferts d'une mémoire vers l'autre.

On va donc chercher bien sûr à minimiser ces transferts.

Le meilleur moyen d'optimiser à ce stade est évidemment de ne pas envoyer de données inutiles!

Occlusion et Visibilité

Comme nous l'avons vu, tous les vertex, textures, normales, etc… doivent être envoyés à la carte graphique.

C'est là que l'on va chercher à optimiser au travers des différents algorithmes d'occlusion pour n'envoyer à la carte graphique que le minimum.
Par définition, ces calculs ne peuvent être fait que sur le CPU. Le mot anglais est Occlusion Culling.

Quelle que soit la puissance de la carte graphique, on aura TOUJOURS besoin d'optimiser!

Voici les algorithmes les plus classiquement utilisés:
- Frustum Culling
- Octrees
- BSP (Binary Space Partionning) Trees
….

Les artéfacts de rendu

- L ’Aliasing :
  - Qu’est ce que c’est ?
  - Solution ?

- Le moirage :
  - Qu’est ce que c’est ?
  - Solution ?

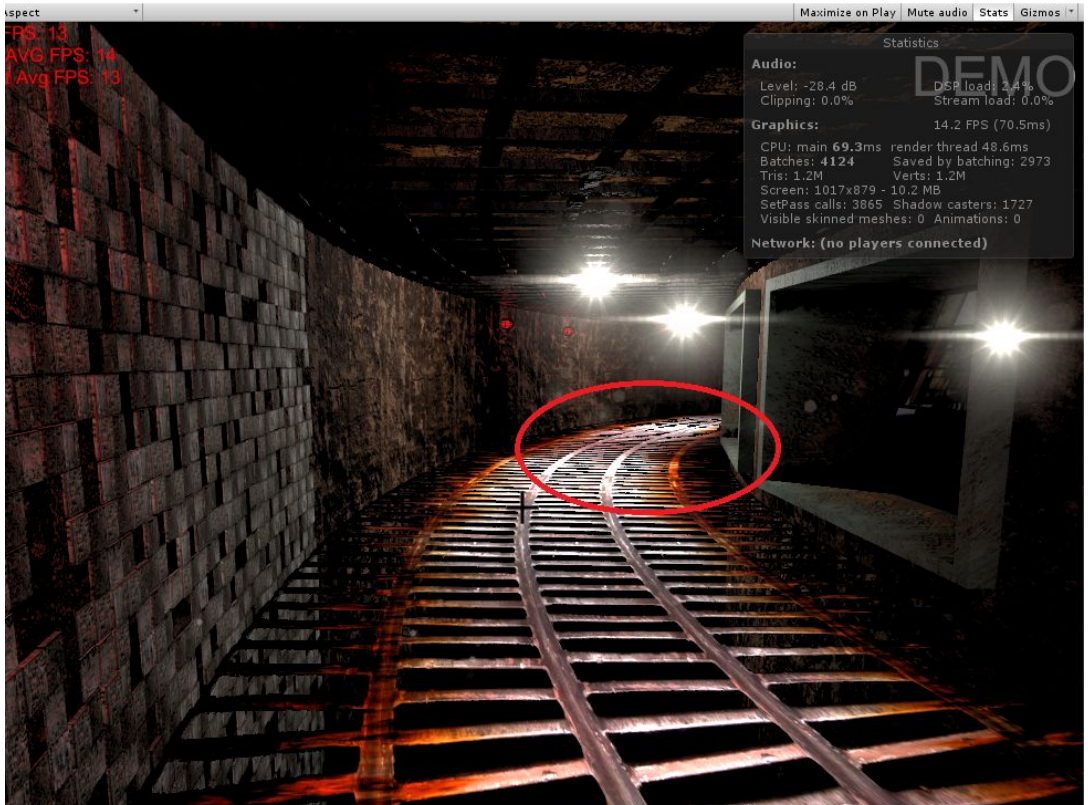
L’Aliasing

On l’appelle aussi effet escalier dans les rendus.
Il est dû au manque de définition et au fait que les pixels sont calculés indépendamment les uns des autres. Point par point.

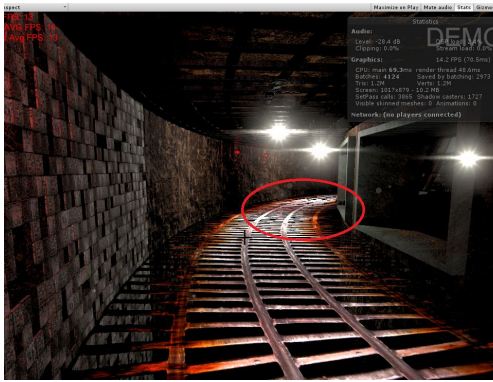

Image courtesy of the Art of Maya

Pour le corriger, on utilise des filtres, qui sont nécessairement full frame, c’est à dire qu’ils traitent l’image entière.
Ces filtres sont efficaces mais assez coûteux en temps réel. Le principe de fonctionnement est de rajouter des points intermédiaires pour casser l’effet escalier.
Les algo utilisés dépendent des types de problèmes.

Le moirage

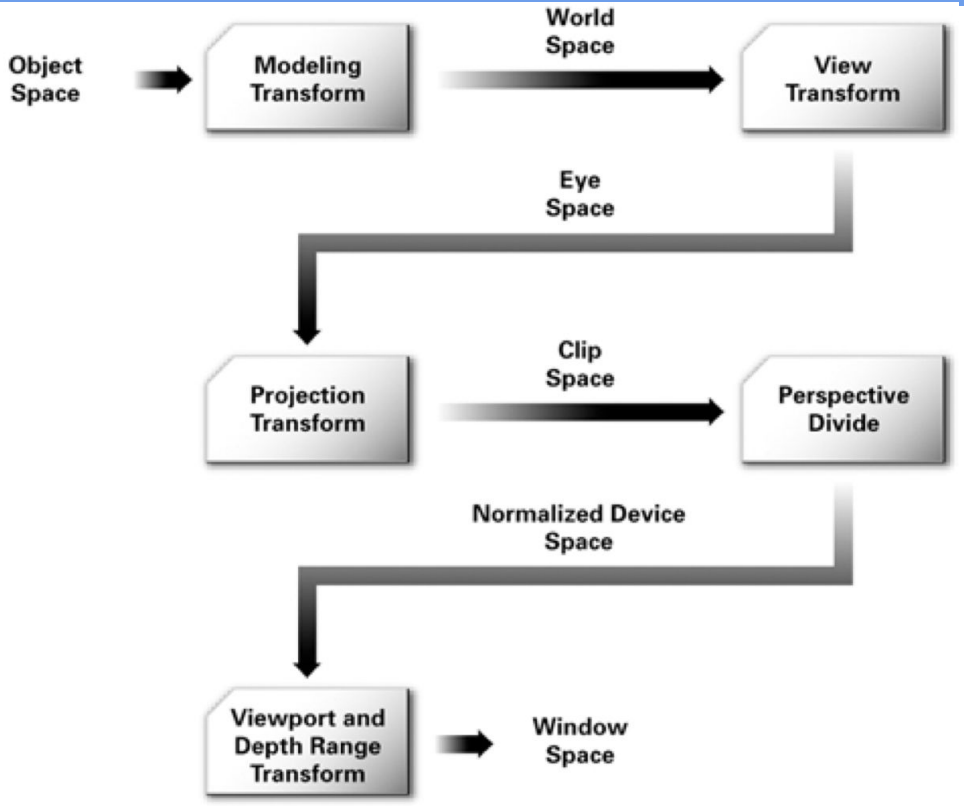

Maximize on Play
Mute audio
Stats
Gizmos
Statistics
Audio:
Level: -28.4 dB
Clipping: 0.0%
Graphics:
14.2 FPS (70.5ms)
CPU: main 69.3ms render thread 48.6ms
Batches: 4124 Saved by batching: 2973
Tris: 1.2M Verts: 1.2M
Screen: 1017x879 - 10.2 MB
SetPass calls: 3865 Shadow casters: 1727
Visible skinned meshes: 0 Animations: 0
Network: (no players connected)
DEMO

Le moirage et le mip-mapping

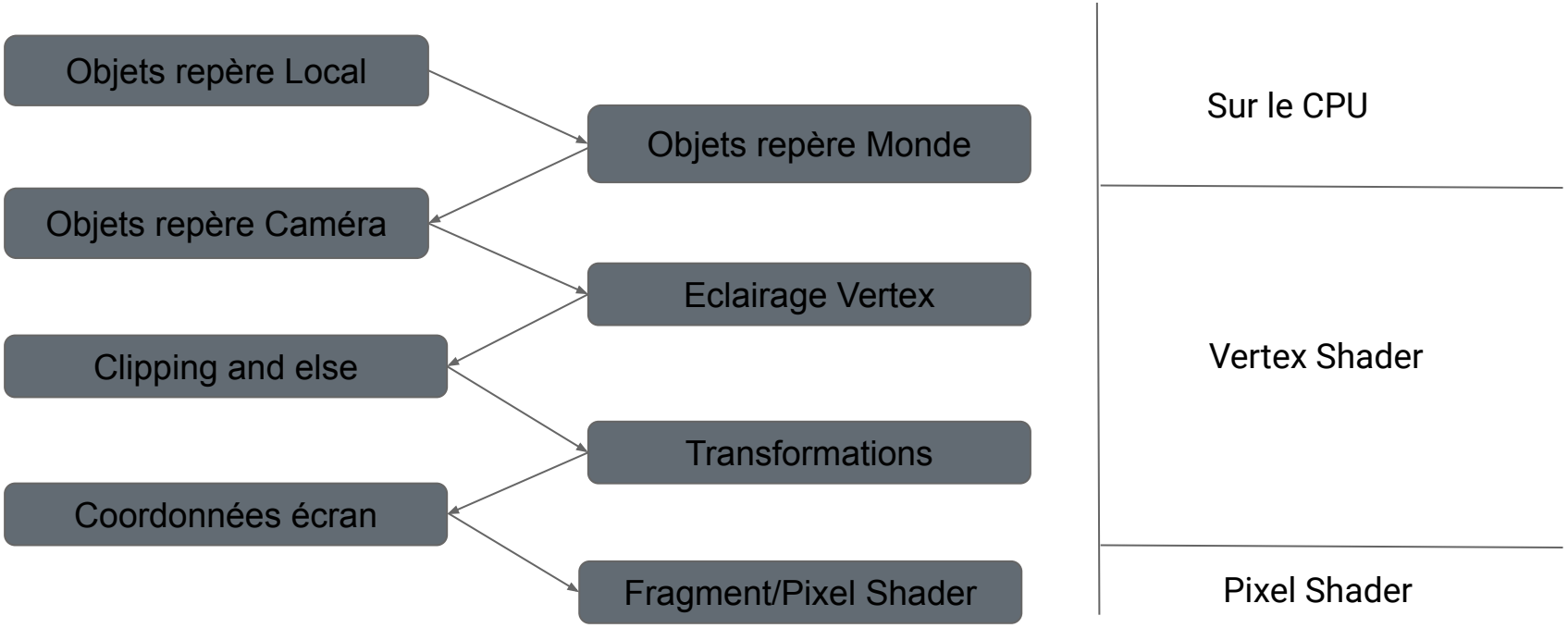


C’est un phénomène d’interférence dû à un excès de précision dans l’image source. Plus l’image contiendra de motifs géométriques (lignes notamment), plus l’effet sera visible.
La solution est le mipmapping. C’est à dire que l’on va pré-calculer des images de résolution inférieure qui seront utilisées en fonction de la taille de la zone à mapper par rapport à l’écran.
L’idée est ici d’avoir une taille de texture adaptée à la taille résultante de la zone mappée à l’écran. Des textures sur-définies NUISENT à la qualité du rendu graphique final.
Certaines cartes/API peuvent précalculer les mipmaps, mais cela sera toujours moins qualitatif que le precalc.
Le mipmapping est aussi une optimisation du rendu, car les niveaux inférieurs de mipmap sont moins gros en mémoire.
A une taille de texture de donnée, le niveau de mipmap suivant sera la taille de la texture divisé par 2, et ainsi de suite jusqu’à atteindre au minimum 2x2 pixels. Les mipmaps sont obligatoires dans le Jeu Video !

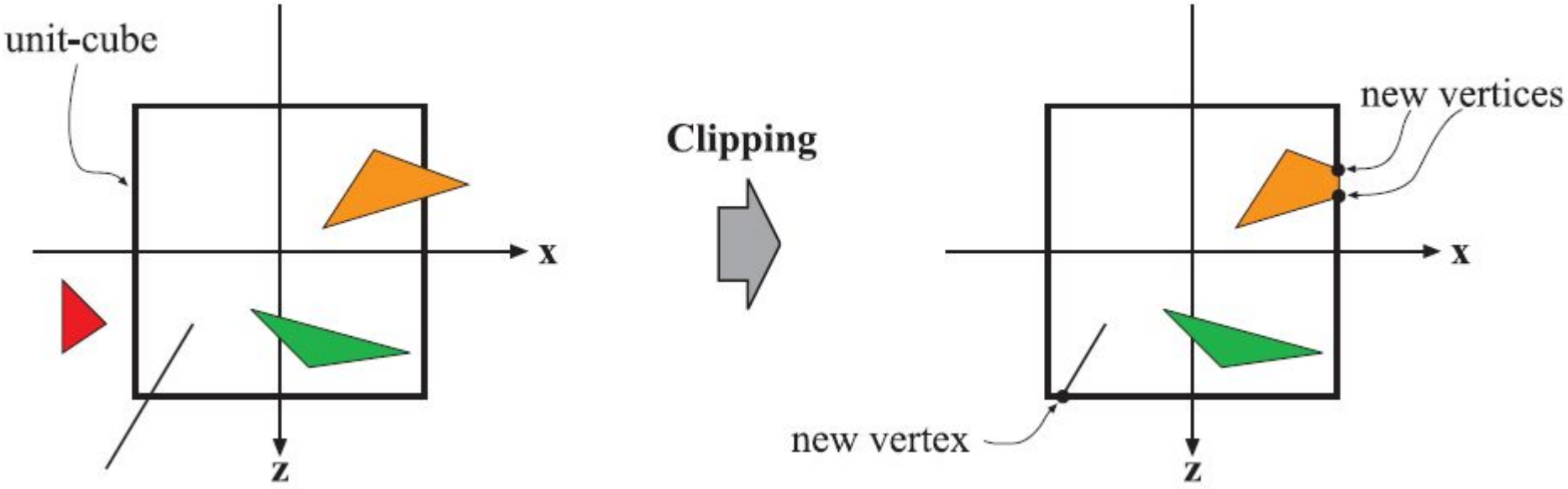
Le pipeline de rendu jusqu'à l'écran :



Quelques explications sur les étapes



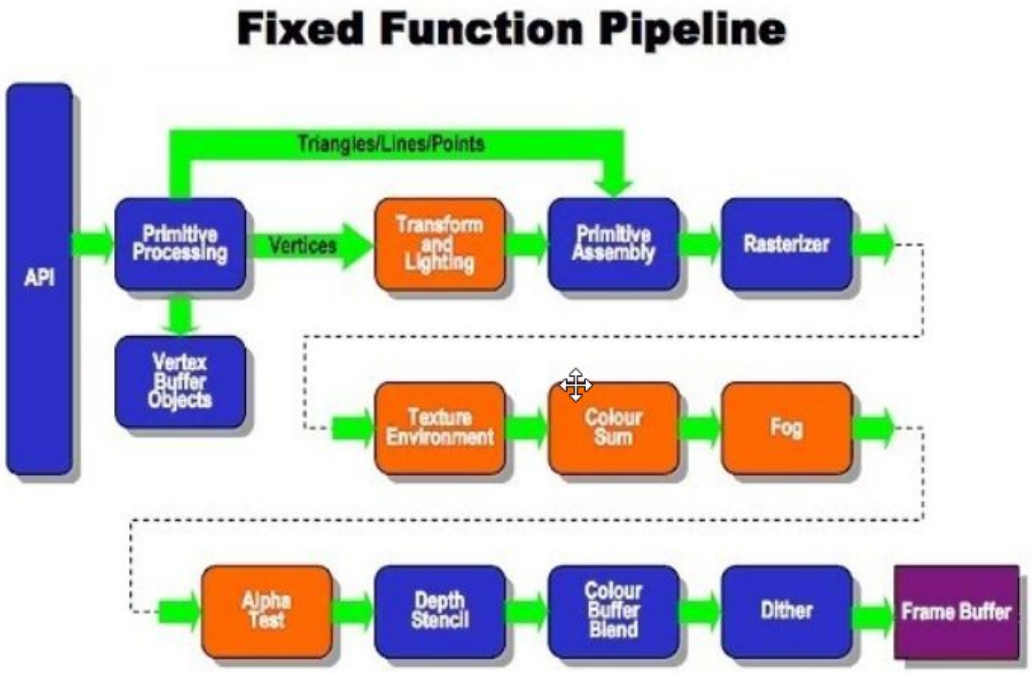
Comment fonctionne le Clipping :



ref : [Akenine-Moeller et al.: Real-time Rendering]

Passons côté programmation maintenant

Le pipeline fixe OpenGL



Quelques explications sur le pipeline fixe:

Vertex Buffer Object (VBO) : une liste de vertex dans un buffer

Primitives : Les géométries que l'on envoie à la carte, sous forme de point (vertex), lignes, triangles.

La passe de Transform & Lightning : les changements de repère des primitives pour calculer l'éclairage

Primitive Assembly : Cette passe découpe toutes les primitives en une séquence de primitives de base qui vont ensuite pouvoir être envoyées au Rasterizer

Rasterizer : C'est la passe qui fait le rendu des primitives dans un buffer intermédiaire

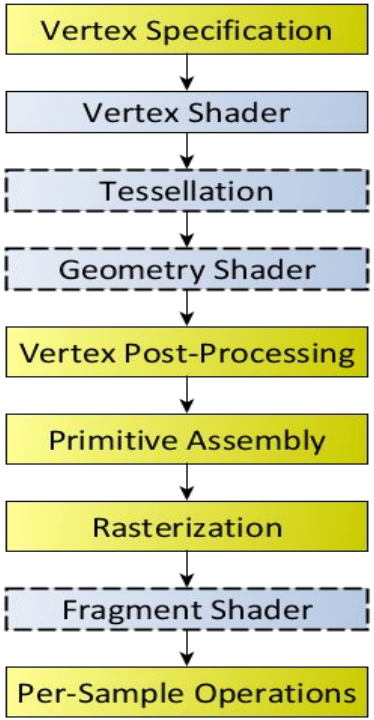
Ensuite, différentes passes prédéterminées arrivent : l'application des textures, l'application des couleurs, la passe de fog, l'alpha test (transparence), le test de profondeur (depth stencil), etc… pour arriver au rendu dans le frame buffer.

le pipeline programmable:

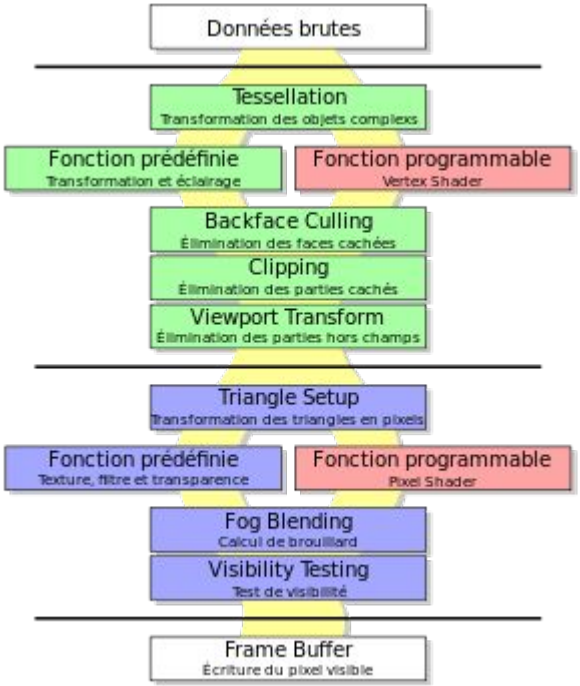
ref : https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

Nous allons voir que le pipeline programmable est par définition beaucoup plus modulable.

Le pipeline programmable Open GL



Différence pipeline fixe vs programmable



le pipeline programmable:

Les données sont envoyées (les vertex notamment).

Chaque vertex est traité par un Vertex Shader (VS) :

- une étape optionnelle de tessellation est possible (génération de vertex)
- une étape de génération de géométrie est possible (génération de primitives)

Ensuite arrive le Vertex Post Processing :

- les transformations sont appliquées
- une première passe de clipping s’applique
- puis la correction de perspective (/w)
- et enfin la transformation en window space

le pipeline programmable:

Puis c'est la passe de génération de primitives (Primitive Assembly)
Ensuite la conversion des primitives en fragments

Les fragment shader (pixel shader en directX) traitent ensuite chaque fragment

Enfin arrivent les opérations dites per-sample:
scissor test
stencil test
depth test
blending

Après toutes ces opérations, on peut effectivement parler de pixel, c'est pour cela qu'openGl parle de fragments.

Explications

Shader : qui pour une définition ?

PixelShader vs Fragment : ?

ZBuffer : qui sait ce que c'est ? peut expliquer ? Algorithme du peintre ?

Depth Buffer

Depth Test

Frame Buffer :

Stencil Buffer :

Explications

Shader : c'est un programme qui va calculer la couleur finale d'un pixel (ou fragment plutôt). Accessoirement, c'est un code qui se parallélise très bien (pourquoi ?) [https://fr.wikipedia.org/wiki/Shader]

PixelShader vs Fragment Shader : Un fragment possède des attributs supplémentaires à la seule position 2D dans l'écran et la couleur, comme par exemple : Normale, UVs, alpha, etc….

Frame Buffer : C'est un buffer qui vont contenir l'image qui sera affichée par la carte graphique

Render Target : C'est la zone mémoire que le rendu va utiliser pour écrire, généralement, la render target 0 est un frame buffer :)

Stencil Buffer : C'est un autre buffer, avec une valeur entière par pixel. Le stencil buffer est souvent utilisé avec le depth buffer pour réaliser des effets. [https://en.wikipedia.org/wiki/Stencil_buffer]

Quand prendre quoi ?

Une caméra persp est surtout utilisée pour un rendu réaliste, donc surtout pour les jeux en 3D.

Une caméra ortho rend les objets plutôt flat. Elle est utilisée pour les jeux en 2D. Les jeux en 3D isométriques utilisent la caméra ortho.

L’envoi à la carte graphique

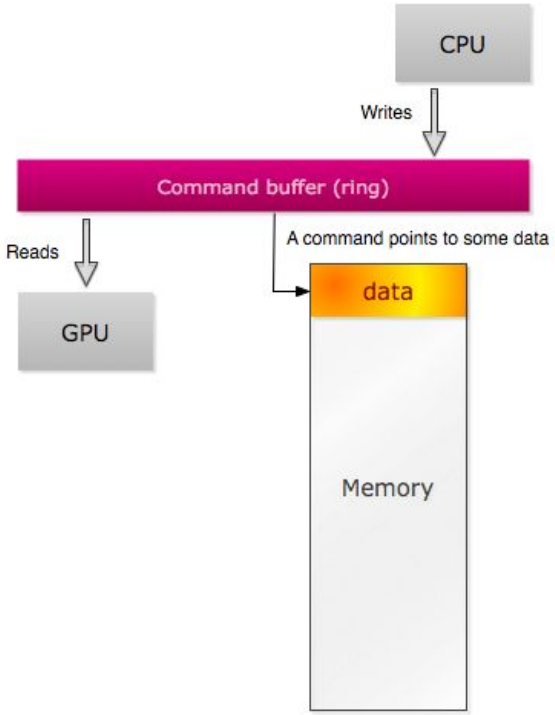
Revenons au jeu.

Très grossièrement, le moteur avec le gameplay du jeu va passer une partie de la trame du jeu à gérer les inputs et en déduire les emplacements logiques (position, orientation, scale) de tous les éléments graphiques.
Ensuite, pendant la passe du rendu, le moteur graphique (c’est à dire la partie de code côté moteur sur le cpu), va devoir envoyer tous les ordres à la carte graphique.

Il va falloir préparer tous les ordres côté CPU et les envoyer le plus efficacement possible sur le gpu. C’est le rôle du command buffer.

Il faut voir le command buffer comme un ring buffer. Il commence à être traité par le gpu alors que l’on va pouvoir continuer à être dedans.

Command Buffer (simplifié)



Command Buffer (simplifié)

Il y a un mécanisme bas niveau, c'est le rôle du driver de la carte graphique, qui va gérer l'envoi des données du command buffer vers la carte graphique.

Le mécanisme est le suivant :

On sette les états nécessaires pour les opérations, on envoit la géométrie voulue, les opérations de tracé, puis on envoie les états suivants, la géométrie suivante, etc…

Il faut noter que les envois de textures sont préparés dans la mesure du possible en amont et qu'on envoie pas les textures dans la ram vidéo à chaque trame, pour des questions évidentes de performances. De la même façon, tous les buffer intermédiaires sont créés lors des phases d'initialisation.

Bien sûr, il faut penser aux problèmes de synchronisations, comme par exemple :
des idées ??

Command Buffer (simplifié)

Bien sûr, il faut penser aux problèmes de synchronisations, comme par exemple :

des idées ??

qu’est ce qui pourrait bien poser problème ?

Command Buffer (simplifié)

- Ne pas effacer les buffer envoyé au rendu tant que ces buffers ne sont pas tous traités (crash)
- Ne pas envoyer au rendu plus qu'il ne peut traiter (low fps)
- Ne pas envoyer trop lentement (low fps)
- Avoir fini d'envoyer toutes les textures dans la ram video
- etc….

Schématiquement, on va envoyer à la carte graphique tous les ordres nécessaires à l'affichage de la trame courante et l'on va ensuite passer au calcul de la trame suivante, pendant que le gpu s'active sur le travail demandé.
Le gpu calcule donc toujours APRÈS la boucle moteur donc fatalement avec une trame de retard.

Quand on est au point d'envoyer un nouveau command buffer au GPU, on doit à ce moment réaliser un point de synchro. Ce point est bloquant si le gpu n'a pas fini de traiter les commandes précédentes.

L’éclairage

L’éclairage est un point très important dans le rendu temps réel. C’est l’éclairage qui va rendre une scène particulièrement réaliste (ou pas du tout).

Il existe de nombreux modèles mathématiques qui vont tenter de reproduire un éclairage le plus réaliste possible. En général, plus l’algorithme va produire un résultat réaliste, plus il sera coûteux, c’est à dire qu’il prendre du temps.

Voyons comment fonctionne un éclairage simpliste de type lumière ambiante matérialisée par un seul point. Cette lumière sera de type sphérique.

Une approche naïve pourrait être de ? qu’en pensez vous ?

L’éclairage

Approche naïve : on va calculer la distance entre la lumière et le vertex est appliqué la lumière à la couleur de ce vertex. Pour cela, on imagine que la lumière à une intensité de 100% sur une certaine distance puis une intensité réduite ensuite jusqu’à ne plus éclairer. On appelle ça un falloff.

Que va-t'il se passer si on réalise cet éclairage par vertex ?

Voyez vous d’autres problèmes à cette approche simpliste ?

Que proposez vous pour améliorer cela ?

L’éclairage

Il existe de nombreux type d’éclairage, les expliquer tous dépasse largement le scope de cette introduction. Cependant, voici les principaux :

Global Illumination : technique très à la mode pour des rendus réalistes

Lumières directionnelles : source lumineuse sans origine particulière

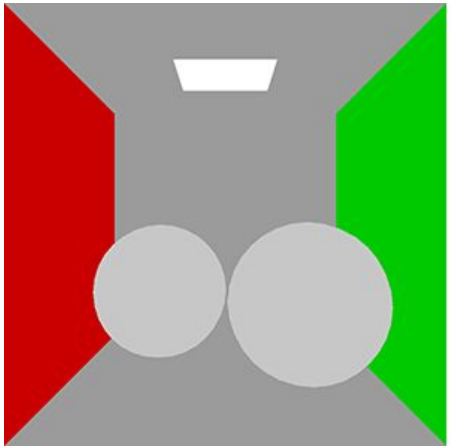
Point Light : source lumineuse de type point

SpotLight : cône qui éclaire depuis un certain point

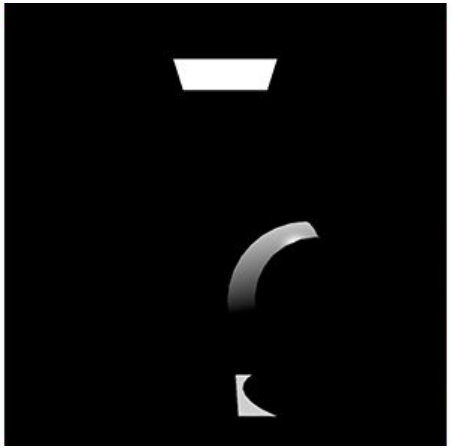
etc, etc...

L’importante de l’éclairage : exemple

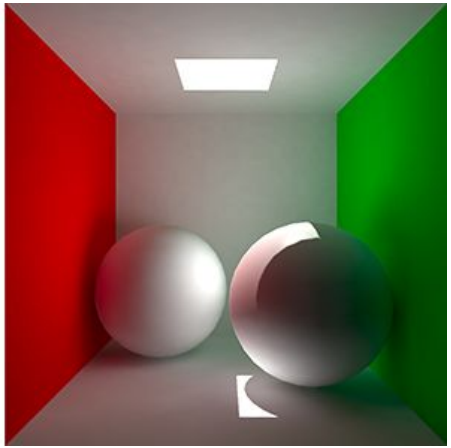
Voici la même scène avec 3 éclairages différents :



Pas d’éclairage



Lumière direct



Global Illumination

L’éclairage

Il existe d’autres types d’éclairage comme l’éclairage statique par lightmap. On va calculer dans une texture l’ éclairage de la scène, pour les sources lumineuses non dynamiques bien sûr. Pendant le rendu, cette texture sera appliqué à la scène comme n’importe quelle autre texture.

La GI est une forme de lightmap. Cependant la technique de GI permet de réaliser un changement d’illumination global de la scène, ce que la lightmap ne permet pas. La GI est compatible avec un changement d’heure (donc d’ensoleillement) d’une scène, par exemple.

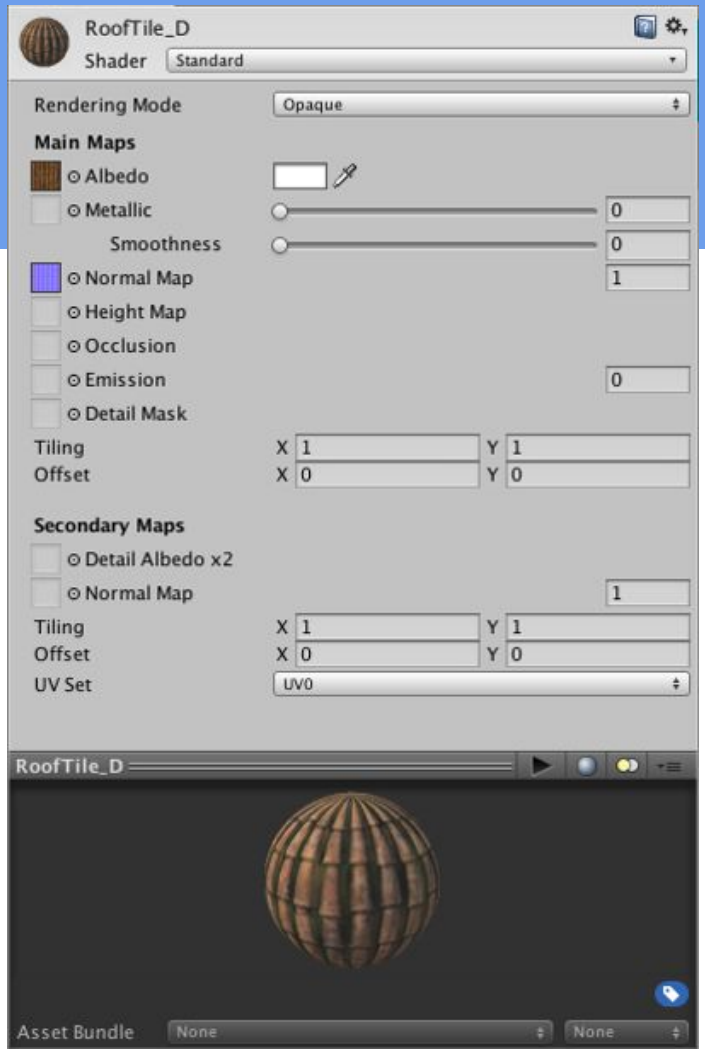
C’est pour cela que l’on fait la différence entre lumières fixes et lumières dynamiques.

Les matériaux

Connaissez vous la différence entre une texture et un material ?

Les matériaux

Et bien la texture peut être un des composants d'un matérial. Celui ci est nettement plus complexe et contient beaucoup d'informations, qui pourront (ou pas) être utilisée directement par les shaders.
Ici, un exemple du material editor de Unity :



Les coordonnées de textures, Le mapping

Pour appliquer une texture à un objet en 3D (un mesh donc), on va utiliser des coordonnées de textures, appelées (u,v) ou uvs. Il s'agit des coordonnées de mapping.

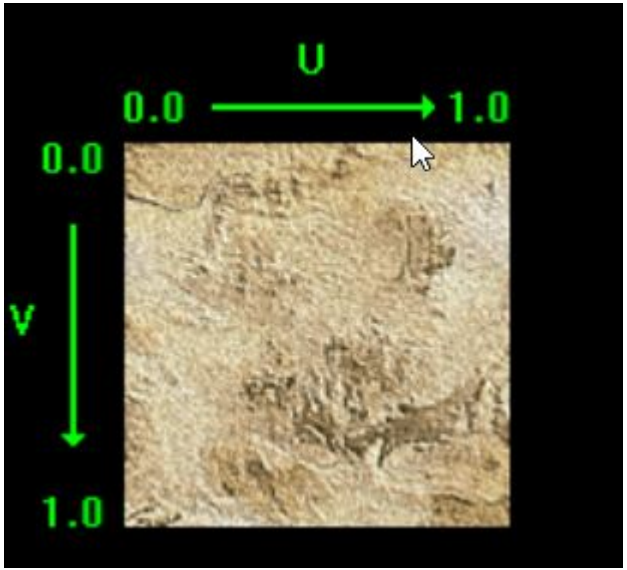
Ces coordonnées s'expriment en flottants, traditionnellement entre 0 et 1.
A chaque vertex de l'objet est associé un uv.

Il faut voir le u comme la position en x dans la texture, 0 étant à gauche et il faut voir le v comme la position en y dans le texture, 0 étant en haut.

Si on veut mapper un quad composé de deux triangles, on va donc avoir des (u,v) de : ??
=> au tableau !

Les coordonnées de textures, Le mapping

(0,0)----------(1,0)
| |
| |
| |
| |
| |
| |
(0,1)----------(1,1)



Vocabulaire et définitions

Tiling : C’est le fait de répéter plusieurs fois une texture. Par exemple, avec des uvs de (2,2), la texture sera dupliquée 2 fois en x et 2 fois en y.

Les shaders

Enfin la partie que vous devez tous attendre… Les Shaders :)

Enfin du code, un exemple

Mais avant de parler de shader, il faut parler de géométrie et comment l'envoyer au GPU.

Pour cela nous avons besoin de Vertex Buffer et d'Index Buffer. VB et IB.

Les Vertex Buffer sont des tableaux des vertex. Ces vertex vont être utilisés dans la géométrie à afficher.
Pour utiliser les vertex, il faut les références, par leur index. C'est pour cela que l'on parle d'Index Buffer. C'est un tableau) une dimension qui contient uniquement des index sur les vertex à utiliser.

Ensuite, les shaders vont entrer en jeu. Un shader s'écrit dans un langage plutôt simple, assez proche du C. Il s'agit du HLSL sous DirectX et du GLSL sous OpenGl. Les langages se ressemblent d'ailleurs suffisamment pour qu'il existe des outils type GLSL vers HLSL, qui vont convertir les shaders d'un format vers l'autre.

Exemple de shader

La première partie d'un shader doit être les variables globales. Ces variables sont modifiées depuis votre code C++ et vont pouvoir être utilisée directement dans votre code de Shader.

Par exemple :

```
cbuffer MatrixBuffer
{
	matrix worldMatrix;
	matrix viewMatrix;
	matrix projectionMatrix;
};
```

Exemple de shader

On va ensuite définir le format de vertex

```
struct VertexInputType
{
    float4 position : POSITION;
    float4 color : COLOR;
};

struct PixelInputType
{
    float4 position : SV_POSITION;
    float4 color : COLOR;
};
```

Exemple de shader

C’est le gpu lui-même qui va appeler le vertex shader quand il doit traiter les données. Le Vertex Shader sera appelé pour CHAQUE vertex de la géométrie. Les données d’entrée du vertex shader doivent être du format du vertex buffer. Le résultat du vertex shader sera ensuite envoyé au Pixel Shader.

Ici le vertex shader se contente de transformer le vertex pour le mettre dans l’espace de la caméra. On va donc multiplier la position du vertex par la matrice de world, puis par la matrice de view et au final pour la matrice de projection.

Ensuite, on va définir la couleur de ce vertex.

Voici le code du Vertex Shader:

Exemple de shader

```
PixelInputType ColorVertexShader(VertexInputType input)
{
	PixelInputType output;

	// Change the position vector to be 4 units for proper matrix calculations.
	input.position.w = 1.0f;

	// Calculate the position of the vertex against the world, view, and projection matrices.
	output.position = mul(input.position, worldMatrix);
	output.position = mul(output.position, viewMatrix);
	output.position = mul(output.position, projectionMatrix);

	// Store the input color for the pixel shader to use.
	output.color = input.color;

	return output;
}
```

Exemple de shader

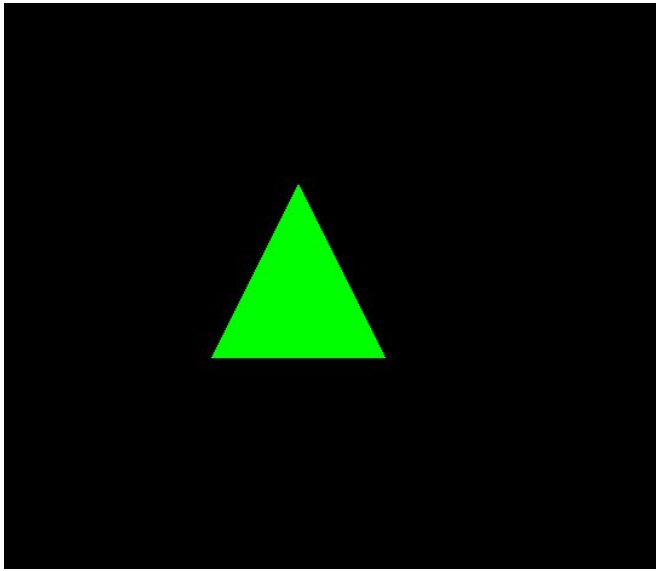
Le pixel shader est responsable d'afficher chaque pixel, il va au final donner sa couleur au pixel, tout simplement. Ici, on se contente de recopier la couleur donnée en entrée.

Voici le code du Vertex Shader:

```
float4 ColorPixelShader(PixelInputType input) : SV_TARGET
{
    return input.color;
}
```

Exemple de shader

Le résultat de votre oeuvre :)



Explications dans le code

Chargement des shaders :

1. Il faut d'abord charger les fichiers dans un buffer intermédiaire (le source donc)
2. Puis, on va compiler le vertex et le pixel shader dans un autre buffer
3. Dernière étape, on va créer le VS et le PS à partir du résultat de la compilation

Explications dans le code

Etude du code, pas à pas

Exercice

Qu’est ce qu’il faudrait modifier que la couleur du triangle soit 2 fois moins lumineuse ?

Qu’est ce qu’il faudrait modifier que la couleur du triangle soit d’une autre couleur ?

Et maintenant, avec une texture ?

On va devoir charger une texture sur le GPU. Puis, référencer cette texture.
Le modèle va bien sûr devoir changer, ainsi que les shaders. Nos données d'entrées pour le shader deviennent donc :

```
struct VertexInputType
{
    float4 position : POSITION;
    float2 tex : TEXCOORD0;
};

struct PixelInputType
{
    float4 position : SV_POSITION;
    float2 tex : TEXCOORD0;
};
```

Et maintenant, avec une texture ?

Ensuite; on va se contenter de recopier les infos d'entrée, cette fois juste pour la texture :

```
// Store the texture coordinates for the pixel shader.
output.tex = input.tex;
```

Et maintenant, avec une texture ?

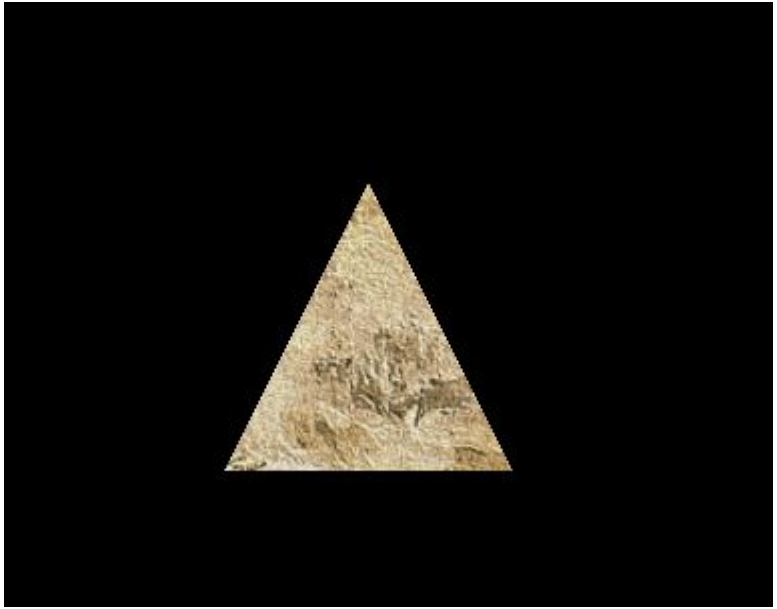
Le code du pixel shader est à peine plus complexe qu'avant, on va 'sampler' la texture en fonction du pixel courant (de sa position donc)

```
float4 TexturePixelShader(PixelInputType input) : SV_TARGET
{
	float4 textureColor;

	// Sample the pixel color from the texture using the sampler at this texture coordinate location.
	textureColor = shaderTexture.Sample(SampleType, input.tex);

	return textureColor;
}
```

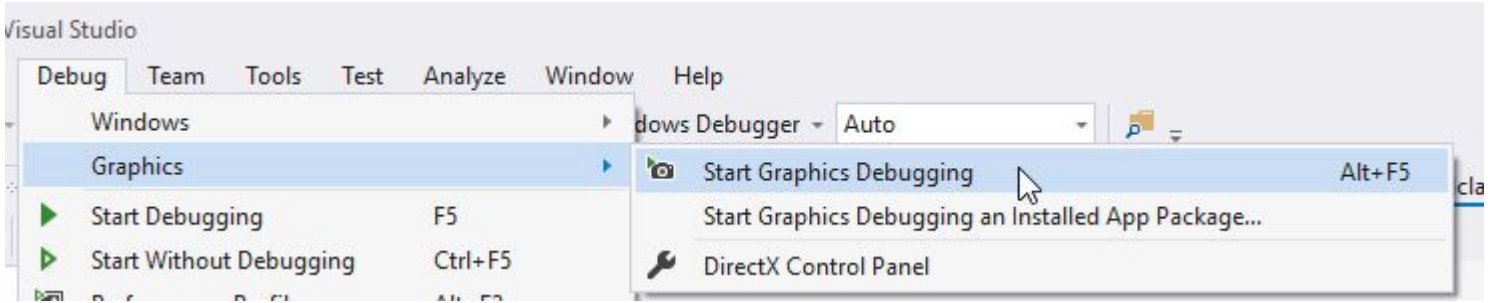
Le résultat



Et le debuggeur dans tout ça ?

Heureusement, il existe maintenant des debuggers graphiques complets pour la programmation graphique et donc le debug des shaders.

Dans visual studio, il est ici :



Conclusion

Nous n’avons fait qu’effleurer le monde du rendu temps réel…