

Game Rendering 101

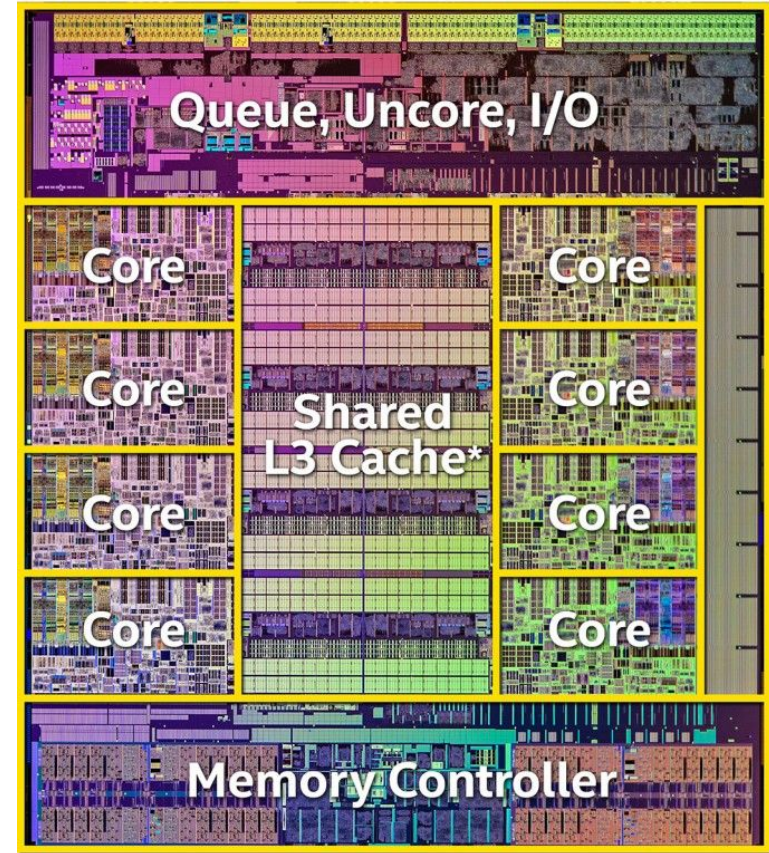
D'un modèle 3D vers une image 2D

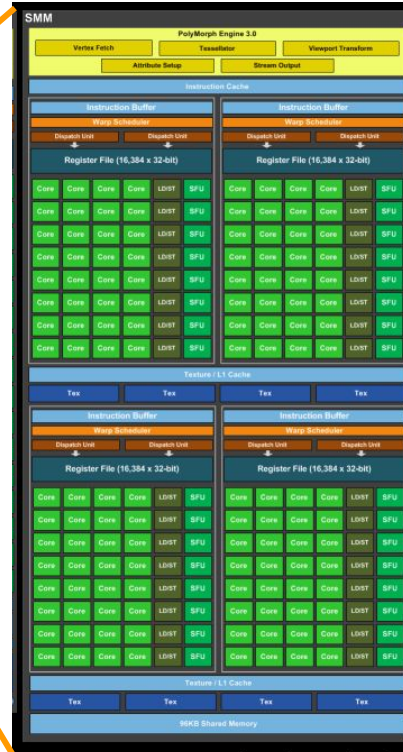
Introduction

GPU qué saco?

Architecture CPU

- Peu de cores (ici 8, 16 logiques avec HT)
- Horloge rapide (ici 3.00 GHz/3.50 GHz)
- Hautement programmable





- Énormément de cores (ici 2048)
- Horloge plus lente (ici 1.1 GHz)
- Partiellement programmable

API graphique

Rends possible la communication
CPU > GPU.

Modèle Client/Serveur, le CPU prépare
des commandes qui seront exécutés dans
le futur par le GPU.

Synchronisation entre les deux limité
et/ou coûteuse.



WebGPU



Microsoft®
DirectX[®]11



OpenGL ES, NVN, AGC, GNM, ...

DirectX 11



Ce cours utilisera **DirectX 11**

Plusieurs raisons:

Meilleurs outils, meilleure compatibilité, API plus simple à utiliser (orienté objet, gestion mémoire simplifiée, etc).

Cours transposable sur OpenGL assez facilement cependant.

Vulkan/DirectX 12 sont en dehors du scope de ce cours car il demande une compréhension plus bas niveau des GPUs et utilise une gestion de la mémoire explicite.

ComPtr

Équivalent Microsoft de `std::shared_ptr`, usage optionnel mais permet de faciliter le management mémoire.

Au lieu d'utiliser un `ID3D11Buffer*` par exemple on peut utiliser un `ComPtr<ID3D11Buffer>` qui va compter les références à la ressource pour nous et va la libérer si elle n'est plus utilisé.

Méthodes les plus utiles:

- `Get ()` => permet d'obtenir le pointeur de la ressource
- `GetAdressOf ()` => permet d'obtenir l'adresse du pointeur de la ressource
- `ReleaseAndGetAdressOf ()` => libère la ressource puis donne l'adresse (utile pour réallouer)
- `Reset ()` => libère la ressource

Template

Template

Pour ce cours je vous invites à cloner/forker ce repo:

https://github.com/Togimaro/TP-ENJMIN_2025

Il contient une base de travail integrant DirectX Tool Kit, une bibliothèque de fonction utilitaire contenant la gestion des inputs, le loading de texture et des classes mathématiques qui nous seront utiles plus tard.

Le projet utilise premake pour la gestion de la solution, après avoir cloné le repo lancez `makeSolution.bat` pour créer la solution Visual Studio. Ouvrez ensuite `Minicraft.sln` et essayez de compiler le projet.



Données

Définition et allocation des ressources nécessaire au GPU

Input Assembler

Assemblage des données fournis par l'utilisateur en primitives (liste de triangle, ruban, ligne, points etc)

Vertex Shader

Transformation des vertices grâce à un programme défini par l'utilisateur (un *shader*)

Données

Qu'est ce qu'on fournit en entrée?

Anatomie d'un modèle 3D

Le terme modèle 3D est assez vaste et peut désigner différentes parties qui forme un “objet 3D”:

- Le maillage
- Son armature et les animations associés
- Les “matériaux” qui lui sont appliqués (textures, interaction avec la lumière, transparence etc)

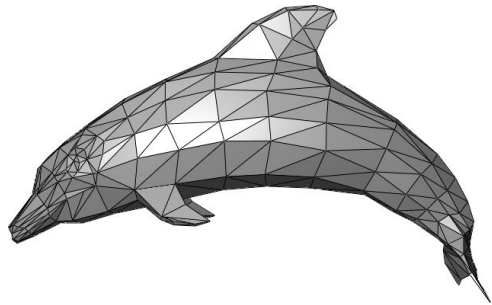
Intéressons nous au maillage

Le maillage (ou *Mesh* en anglais) constituent la partie “tangible” d’un modèle 3D.

Il est composé de sommets (*vertex/vertices*) et de face (généralement des triangles).

Un vertex peut contenir tout un tas d’information nécessaire au rendu:

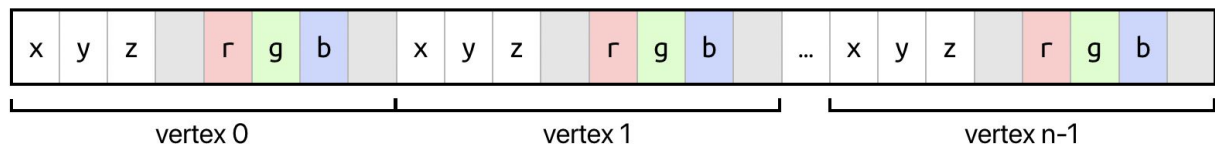
- Une position
- Une ou plusieurs coordonnées de textures
- Une couleur
- Une normal/tangente/binormal
- Des poids et des indices pour les os de l’armature
- À-peu-près tout ce que vous voulez vraiment



Toutes ces données sont facultatives. C’est à l’utilisateur de définir ce dont il a besoin pour un objet donné.

Stockage de notre mesh

On va mettre toutes nos données à la suite dans un bloc mémoire continu appelé un buffer.



Vous noterez que les données de chaque vertices sont placé à la suite en mémoire sans séparation entre chaque type d'élément ni entre chaque vertices.

(mis à part le padding éventuel afin que tout soit aligné sur 16 bytes)

Stockage de notre mesh avec DX11

On va utiliser `CD3D11_BUFFER_DESC` pour décrire notre stockage.

```
CD3D11_BUFFER_DESC desc(  
    (uint) tailleEnOctet,  
    (uint) bindFlags) // e.g. D3D11_BIND_VERTEX_BUFFER
```

La classe qui va nous représenter notre stockage est `ID3D11Buffer*`.

Comme toutes les créations de ressources c'est `ID3D11Device` qu'il va falloir utiliser.

Pour créer notre buffer on appelle la fonction :

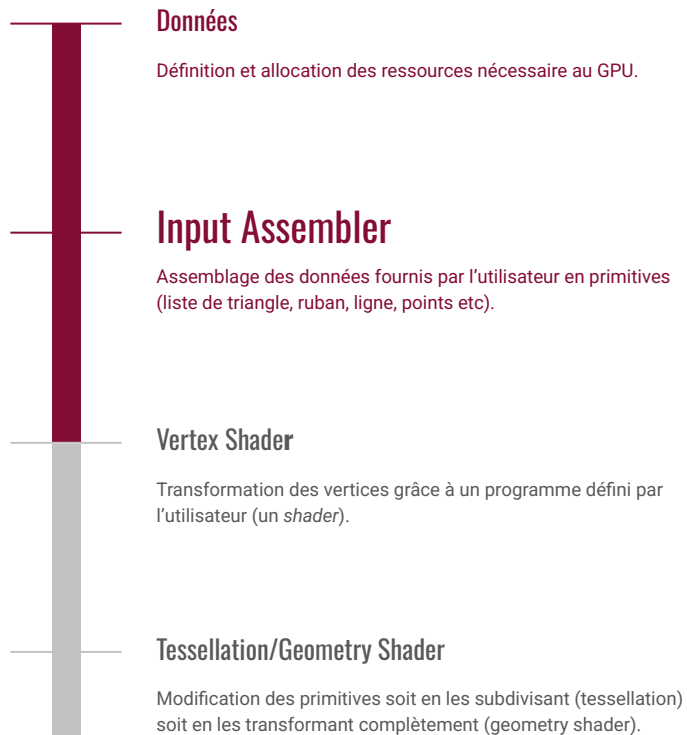
```
ID3D11Device::CreateBuffer(  
    (D3D11_BUFFER_DESC*) desc,  
    (D3D11_SUBRESOURCE_DATA*) dataInitial, // optionnel, peut-être nullptr  
    (ID3D11Buffer**) adresseFuturBuffer); // cf. ReleaseAndGetAddressOf
```

Remplir un buffer en DirectX11

Nous avons trois façon principal de remplir un buffer:

- A l'initialisation en fournissant un `D3D11_SUBRESOURCE_DATA*` à `CreateBuffer`

```
D3D11_SUBRESOURCE_DATA subResData = {};  
subResData.pSysMem = &data; // pointeur vers la data  
device->CreateBuffer(&desc, &subResData,  
vertexBuffer.ReleaseAndGetAddressOf());
```
- Pour les buffers **dynamique**: en utilisant `Map()` pour rendre accessible pendant un temps au CPU la mémoire GPU (attention, cela empêche le GPU d'y accéder jusqu'à l'appel à `Unmap()`). Peut être configuré pour l'écriture, la lecture ou les deux.
- Pour les buffers **statique**: En utilisant la fonction `UpdateSubresource` qui va permettre de permettre de déclencher une copie d'un bloc mémoire accessible au CPU vers un endroit de la mémoire GPU.



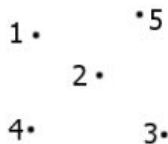
Rôle de l'Input Assembler

On a maintenant un tas de vertex, mais ils ne sont pas encore connectés entre eux.
C'est le rôle de l'Input Assembler.

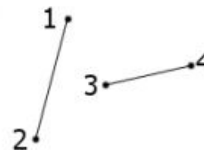
Cette partie du pipeline est configurable.
On va pouvoir y définir la topologie, les buffers à envoyer et le layout de ces buffers.

Ci-contre une liste non-exhaustive des types de topologie les plus communs.
(*TRIANGLELIST* étant de loin celle qu'on utilisera le plus)

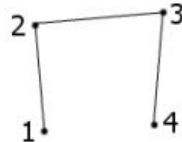
POINTLIST



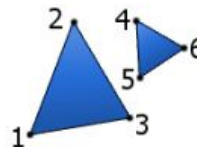
LINELIST



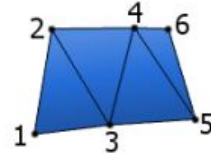
LINESTRIP



TRIANGLELIST



TRIANGLESTRIP



Input Layout - Définition

J'ai dit qu'on avait un tas de vertex, c'est pas exactement vrai!

On a un tas **de nombre** mais le GPU n'a aucun moyen de savoir lesquels sont des positions, lesquels sont des coordonnées de textures, des normals etc.

Pour cela on va définir un Input Layout.

Input Layout - Définition

Pour créer un Input Layout il va nous falloir un tableau de description de chaque élément:

```
D3D11_INPUT_ELEMENT_DESC InputLayoutDescription[] = {
    {
        (CHAR*) semanticName, (UINT) semanticIndex, // eg "POSITION", 0
        (DXGI_FORMAT) format, // eg DXGI_FORMAT_R32G32B32A32_FLOAT
        (UINT) inputSlot, // utile si on passe plusieurs buffer au GPU, sinon 0
        (UINT) alignement, // optionnel, sinon D3D11_APPEND_ALIGNED_ELEMENT
        (UINT) inputSlotClass, // typiquement D3D11_INPUT_PER_VERTEX_DATA
        (UINT) incrementPerInstance // inutile avec PER_VERTEX_DATA
    },
    // répéter pour chaque autre élément de notre vertex
}
```

Input Layout - Création

La classe qui va nous représenter notre layout est `ID3D11InputLayout*`.

Maintenant qu'on a notre tableau de descriptions on va le passer à la fonction `CreateInputLayout` de `ID3D11Device`.

```
ID3D11Device::CreateInputLayout(  
    (D3D11_INPUT_ELEMENT_DESC*) tableauDesDescriptions,  
    (UINT) tailleTableau,  
    (void*) vertexShaderBytecode, (size_t) vsBytecodeLength,  
    (ID3D11InputLayout**) adresseFuturLayout); // cf. ReleaseAndGetAdressOf
```

Comme vous le voyez un vertex shader est nécessaire, cependant un même layout sera utilisable par plusieurs shaders s'ils partagent le même layout.

Utilisation de l'Input Assembler

Plusieurs fonctions contenu dans `ID3D11DeviceContext`:

- `IASetPrimitiveTopology((D3D11_PRIMITIVE_TOPOLOGY) topology)`
- `IASetInputLayout((ID3D11InputLayout*) layout)`
- `IASetVertexBuffers((UINT) slot, (UINT) nombre, (ID3D11Buffer**) buffers, (UINT*) strides, (UINT*) offsets)`

Exemple:

```
ID3D11Buffer* vbs[] = { vertexBuffer.Get() };
UINT strides[] = { sizeof(Vector3) };
UINT offsets[] = { 0 };
context->IASetVertexBuffers(0, 1, vbs, strides, offsets);
```

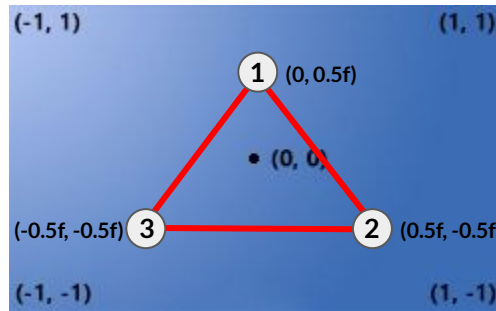
TP 1

Afficher un triangle

Afficher un triangle

A partir de l'application de base fournit:

- Allouer un tableau de float contenant vos coordonnées de triangle (cf image en bas)
- Décrivez un buffer de la bonne taille avec `CD3D11_BUFFER_DESC`
- Utilisez `device->CreateBuffer` pour créer le buffer et initialisez son contenu en passant un `D3D11_SUBRESOURCE_DATA` à la fonction
- Dans la fonction `Game::Render` donnez à l'Input Assembler votre vertex buffer avec `IASetVertexBuffers` puis appelez `context->Draw(3, 0)` pour afficher votre triangle



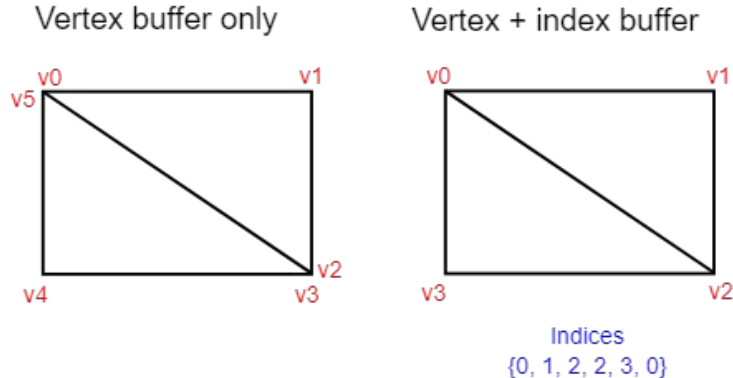
Index Buffer

La topologie est intégralement défini par l'ordre des vertices, cependant il y a de forte chance que vous allez vouloir utiliser un même vertex pour 2 triangles différents.

Les Index Buffers vont nous permettre de définir l'ordre des vertices et de les réutiliser:

```
- IASetIndexBuffer (
    (ID3D11Buffer*) buffer,
    (DXGI_FORMAT) format,
    (uint) offset)
// format: DXGI_FORMAT_R32_UINT
```

Il s'agit juste d'un buffer qui ne contient que des indices qui représente dans l'ordre quelles vertices utiliser.



TP 2

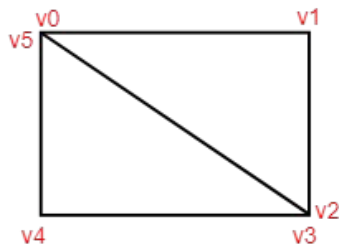
Ajouter un index buffer

Afficher un rectangle

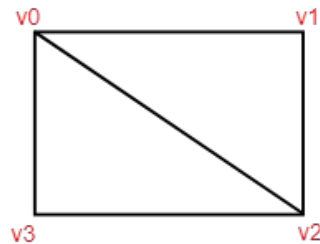
A partir du TP précédent:

- Modifier votre buffer précédent pour avoir les coordonnées d'un rectangle
 - Allouer un buffer contenant des entiers pour vos index
 - Dans la fonction `Game::Render` donnez à l'Input Assembler votre index buffer avec `IASetIndexBuffer` puis appelez `context->DrawIndexed(6, 0, 0)` pour afficher votre triangle
- `DXGI_FORMAT_R32_UINT`

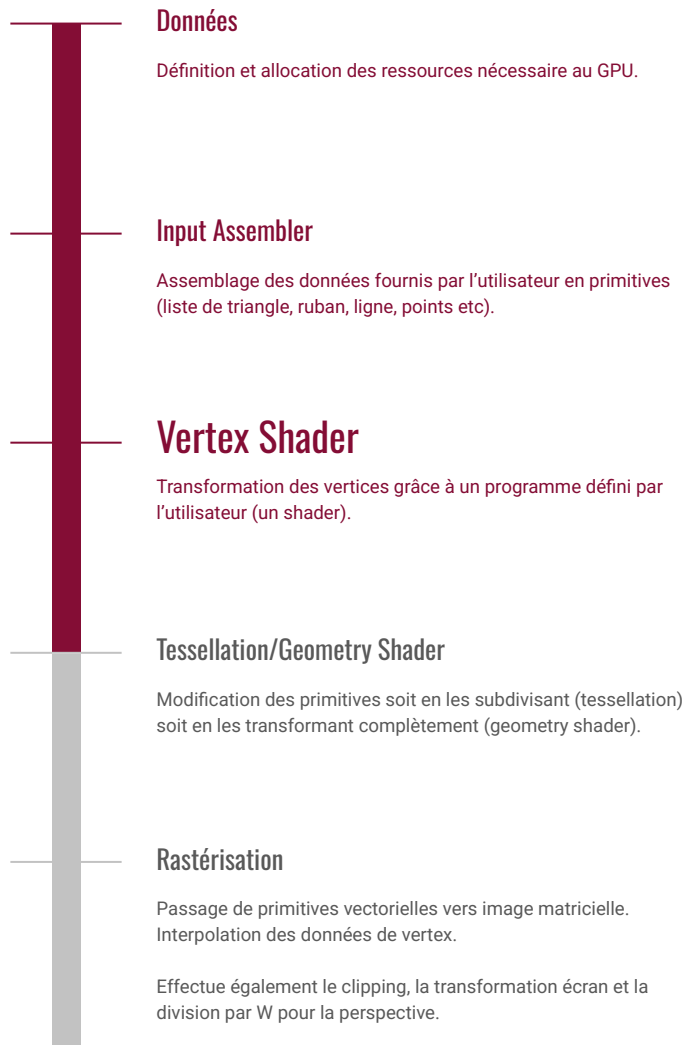
Vertex buffer only



Vertex + index buffer



Indices
{0, 1, 2, 2, 3, 0}



Mais juste avant:
Une petite pause math (désolé)

Qu'est ce qu'un **vecteur** au juste?

Objet mathématique généralisant plusieurs concept:

- **Position** dans l'espace
- **Translation** d'un point
- **Force** physique

Qu'est ce qu'un **vecteur** au juste?

En géométrie il est caractérisé par:

- Une **direction**
- Un **sens**
- Une **norme** (ou **magnitude**)

Opérations sur les vecteurs

Opérations

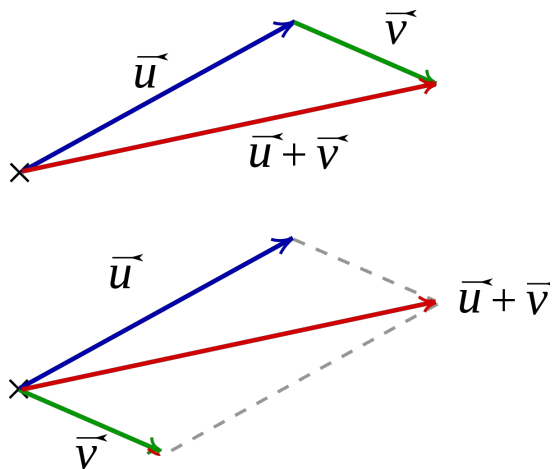
- **Addition/Soustraction** par un autre vecteur
- **Multiplication/Division** par un scalaire
- Produit scalaire (*dot product*)
- Produit vectoriel (*cross product*)
- Obtenir la norme (longueur) du vecteur

Généralement à cela s'ajoute des fonctions de normalisation, de distance

Addition

Pour additionner (ou soustraire) deux vecteurs on ajoute ces composantes une à une

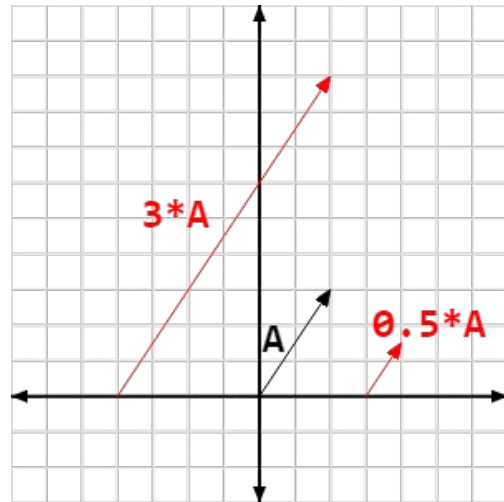
Un vecteur peut être décomposé en somme de vecteur aligné sur les axes du plan, en cela additionner 2 vecteur revient à ajouter les coordonnées



Multiplier

Pour multiplier (ou diviser) un vecteur par un nombre on multiplie chacune des composantes par ce nombre

La division se passe de la même façon.



Longueur d'un vecteur

On utilise le théorème de Pythagore pour calculer la norme d'un vecteur.

$$\|\vec{u}\| = \sqrt{x^2 + y^2 + z^2}$$

A noter: l'opération racine carré étant coûteuse, on ajoute souvent une variante `SqrLength()` qui ne la fait pas (pratique pour des simples comparaisons de longueur par exemple)

Produit scalaire

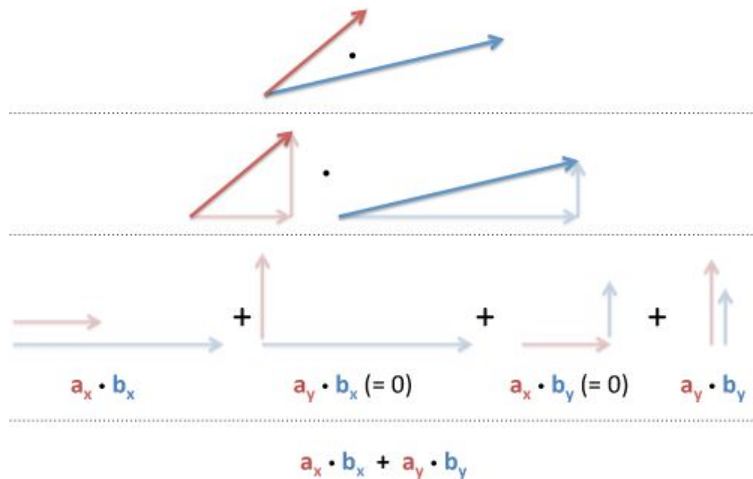
Le produit scalaire se forme comme ceci:

$$\mathbf{u} \cdot \mathbf{v} = u_x \cdot v_x + u_y \cdot v_y + u_z \cdot v_z$$

Il possède des propriétés particulièrement utile dans le domaine du rendu graphique.

Notamment 2 vecteur orthogonaux ont un **produit scalaire égale à 0**. Pratique pour simuler un ombrage (plus sur ça quand on arrivera au pixel shader)

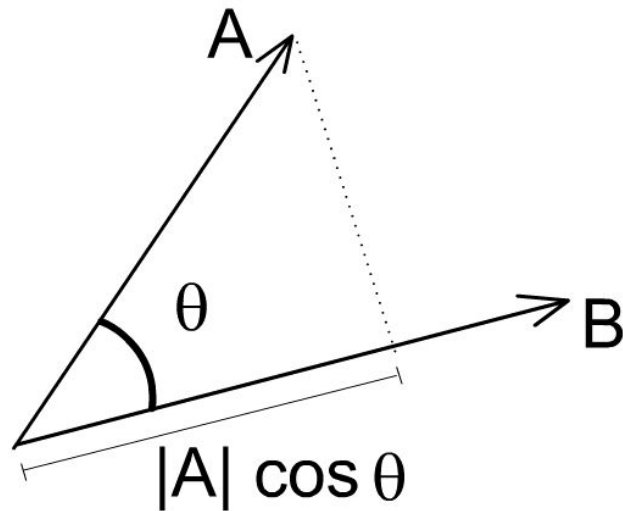
Dot Product: Piece by Piece



Produit scalaire

Autre représentation du produit scalaire:

$$\mathbf{u} \cdot \mathbf{v} = ||\mathbf{u}|| \cdot ||\mathbf{v}|| \cos(\theta)$$

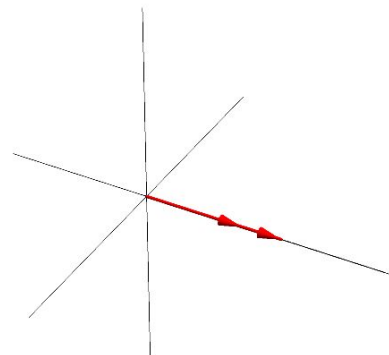


Produit vectoriel

Le produit vectoriel se forme comme ceci:

$$\mathbf{u} \wedge \mathbf{v} = \begin{pmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{pmatrix}$$

(ici 1 2 3 sont là pour représenter x y z)



Il permet (entre autre) de calculer la normal d'un plan défini par 2 vecteurs

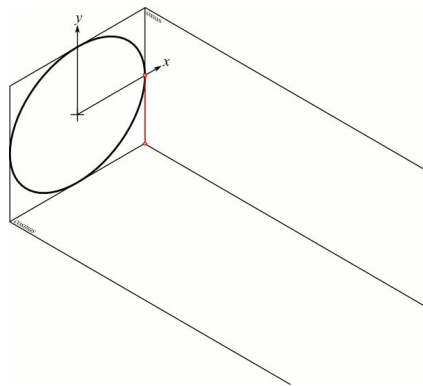
Rotations

Rotation en 2D

On va utiliser les principes de bases de la trigonométrie pour effectuer une rotation en 2D. Il s'agit de la même formule que pour obtenir un point autour d'un cercle, seulement adapté pour faire tourner n'importe quel point

$$x' = x \cos \varphi - y \sin \varphi$$

$$y' = x \sin \varphi + y \cos \varphi$$

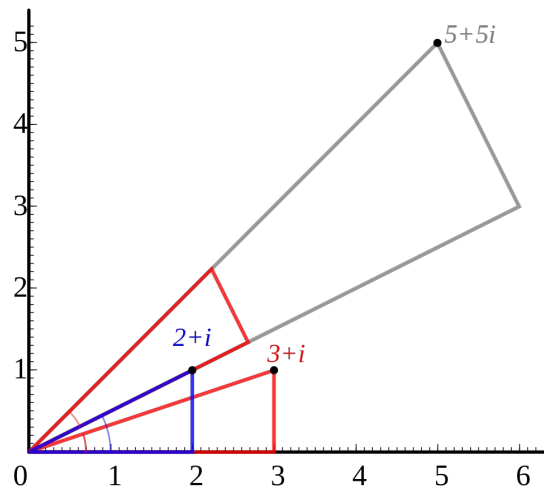


Détour: Nombres complexes

Un nombre complexe peut être utilisé pour **représenter une rotation**. En effet, multiplier 2 nombres complexe entre eux agit comme une rotation du deuxième sur le premier (et d'un agrandissement selon la magnitude du nombre complexe).

Cela se voit particulièrement dans l'écriture sous forme polaire d'un nombre complexe:

$$m(\cos \varphi + i \sin \varphi) = m e^{i\varphi}$$



Pour la 3D: Les Quaternions

En 3D le principe est similaire mais au lieu de 2 composantes il y en a 4 (une partie réel et trois parties imaginaires), c'est ce qu'on appelle des **Quaternions**.

Comme il s'agit d'un nouveau type de nombre il s'accompagne de formules qui leurs sont propres.

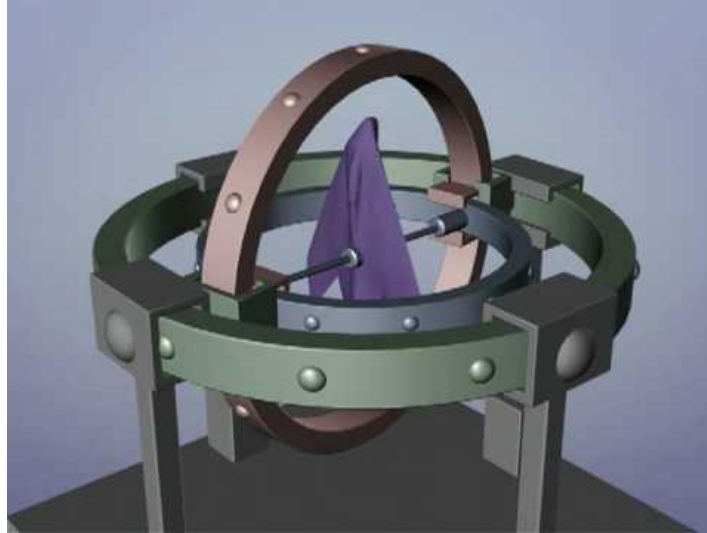
$$i^2 = j^2 = k^2 = ijk = -1$$

<https://eater.net/quaternions>

$$\begin{array}{lll} i^2 = -1, & ij = k, & ji = -k, \\ j^2 = -1, & jk = i, & kj = -i, \\ k^2 = -1, & ki = j, & ik = -j, \end{array}$$

Pourquoi s'embêter avec des Quaternions?

Les angles d'Euler (les angles *classiques* affichés dans l'éditeur) possèdent un souci qui s'appellent le Gimbal Lock:



Application d'une transformation

Vous avez pu voir 3 transformations usuelles:

- La translation (addition de vecteur)
- La mise à l'échelle (multiplication de vecteur par un scalaire)
- La rotation (quaternion)

Cependant si nous voulons appliquer ces transformations il faut actuellement les appliquer une par une, dans un certain ordre, avec des fonctions différentes.

Pour résoudre ce soucis nous allons utiliser des **matrices de transformation**

Matrice de transformation

Pour “stocker” nos transformations nous allons utiliser des matrices.

Une matrices n'est rien d'autre qu'un tableau de nombre, mais avec des opérations qui leurs sont propres, notamment:

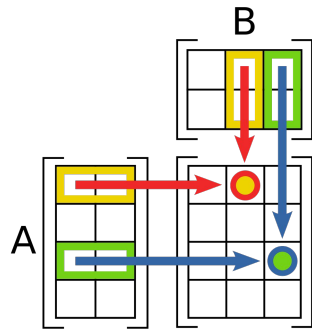
- Produit matriciel
- Transposition
- Inversion

Dans ce contexte, un vecteur n'est rien d'autre qu'une matrice avec une seul ligne.

Produit matriciel

Le produit matriciel est l'opération principal qui va nous permettre d'appliquer nos transformations!

La matrice resultat de l'operation $A \times B$ contient les produits scalaire entre chaque lignes de A par rapport à chaque colonnes de B



⚠ Le nombre de colonnes dans la matrice A **doit être égal** au nombre de ligne dans la matrice B

$$\begin{array}{c} \vec{a}_1 \rightarrow \\ \vec{a}_2 \rightarrow \end{array} \begin{array}{cc} \vec{b}_1 & \vec{b}_2 \\ \downarrow & \downarrow \end{array} \begin{array}{c} A \\ B \end{array} = \begin{array}{cc} C \end{array}$$

$$\begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a}_1 \cdot \vec{b}_1 & \vec{a}_1 \cdot \vec{b}_2 \\ \vec{a}_2 \cdot \vec{b}_1 & \vec{a}_2 \cdot \vec{b}_2 \end{bmatrix}$$

TP 3

Matrice sur papier

Produit matriciel : pratique

Sur papier, calculez la matrice C résultat de l'opération $A \times B$

$$\begin{array}{ccc}
 & \vec{b}_1 & \vec{b}_2 \\
 & \downarrow & \downarrow \\
 \begin{array}{l} \vec{a}_1 \rightarrow \\ \vec{a}_2 \rightarrow \end{array} & \begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} & \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a}_1 \cdot \vec{b}_1 & \vec{a}_1 \cdot \vec{b}_2 \\ \vec{a}_2 \cdot \vec{b}_1 & \vec{a}_2 \cdot \vec{b}_2 \end{bmatrix} \\
 A & B & C
 \end{array}$$

Rappel: $a \cdot b = a_1 \cdot b_1 + a_2 \cdot b_2 + \dots + a_n \cdot b_n$

Attention l'opération de multiplication n'est PAS commutatif : $A \times B \neq B \times A$

Lien avec les transformations

On a : $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$ qui donne donc une fois écrit sans matrices: $x' = ax + by$
 $y' = cx + dy$

Cela vous rappelle-t-il quelque chose?

Lien avec les transformations

On a : $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \\ cx + dy \end{bmatrix}$ qui donne donc une fois écrit sans matrices: $x' = ax + by$
 $y' = cx + dy$

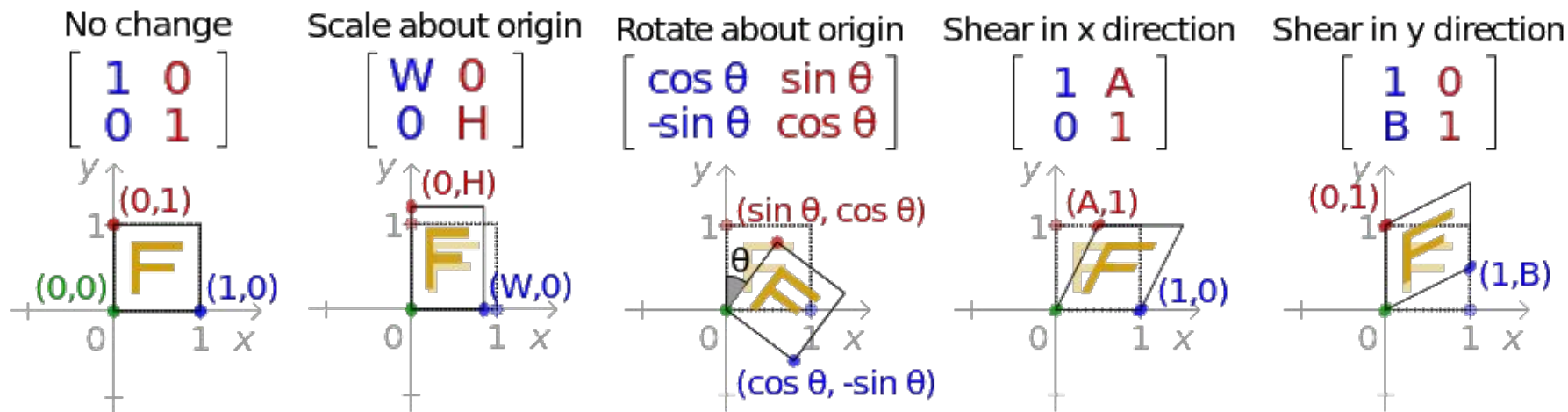
Cela vous rappelle-t-il quelque chose?

$$x' = x \cos \varphi - y \sin \varphi$$

$$y' = x \sin \varphi + y \cos \varphi$$

Lien avec les transformations

Avec cet outil mathématique simple on peut ainsi facilement effectuer tout un tas de transformation affine. Chaque colonne de notre matrice représente une base de notre nouvel espace de coordonnées.



Le problème de la translation

Le fonctionnement actuel fait qu'on ne peut pas ajouter une addition toute seule en fin de calcul, la translation agit comme un décalage et il n'y a aucun lien entre xy et ce décalage. Il est donc impossible d'utiliser une matrice 2×2 pour ça.

La solution: ajouter une composante et passer sur une matrices 3×3

$$\begin{bmatrix} 1 & 0 & Tx \\ 0 & 1 & Ty \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x + Tx \\ y + Ty \\ 1 \end{bmatrix}$$

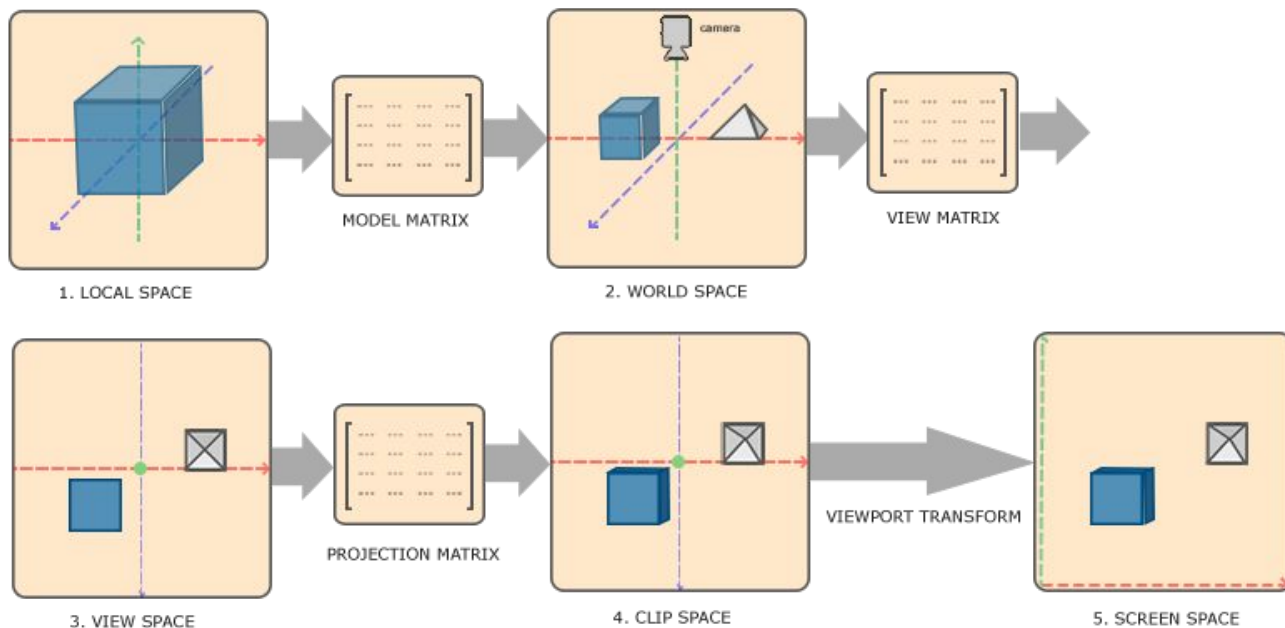
Cette composante ajouté va s'appeler w , on parle alors de coordonnées homogènes.

Talk GDC à ce propos: <https://www.youtube.com/watch?v=o1n02xKP138>

Géométrie des transformations

Affichage d'un objet 3D dans un espace écran 2D

L'affichage en 3 dimensions consiste juste en une suite successive d'application de matrice:

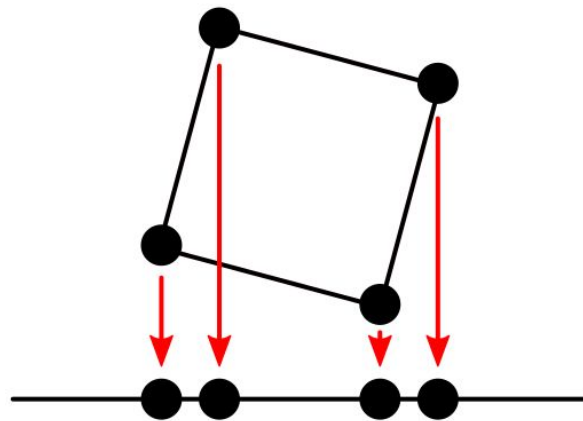


Projection orthographique

En projection orthographique on ignore totalement la composante Z lors de la projection

On obtient une image aplati sans profondeur ni perspective

$$\begin{pmatrix} \frac{1}{r} & 0 & 0 & 0 \\ 0 & \frac{1}{t} & 0 & 0 \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Le problème de la perspective

Pour simuler la perspective nous voudrions divisé chaque composant de notre vecteur par son Z afin qu'un objet plus loin apparaissent plus petit.

Mais malheureusement ce n'est pas possible d'avoir une division de la sorte avec les matrices!

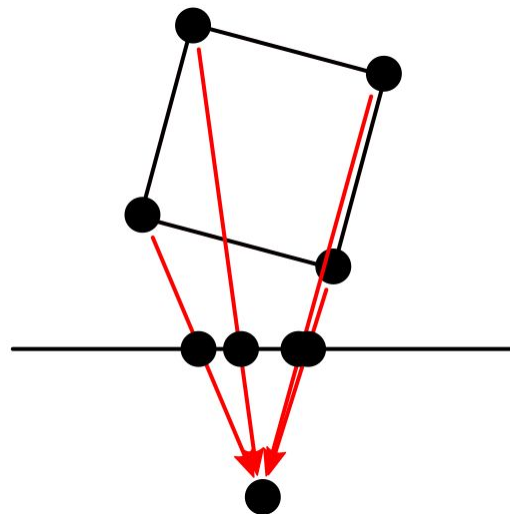
Heureusement pour nous les coordonnées homogènes ne nous sont pas utile que pour la translation! En effet, notre vecteur sera diviser par w automatiquement par le GPU afin de passer de coordonnées homogenes à des coordonnées carthesiennes!

Il nous suffit donc de mettre notre composante z dans w a l'aide d'une matrice

Projection perspective

Et c'est comme ça que l'on arrive à une matrice de projection perspective! Le composant entouré en rouge à pour effet de mettre -Z (*Forward en repère main droite*) dans W. Le reste de la matrice est là pour gérer l'aspect ratio et remapper Z entre near et far afin de le conserver malgré la division!

$$\begin{pmatrix} \frac{1}{\text{aspect} * \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} & -\frac{2 * \text{far} * \text{near}}{\text{far} - \text{near}} \\ 0 & 0 & \boxed{-1} & 0 \end{pmatrix}$$



DirectXMath - SimpleMath.h

Heureusement nous n'allons pas manipuler tout ça nous même. DirectX nous fournit la bibliothèque DirectXMath pour nous faciliter la vie.

Et mieux encore DXTK nous fournit SimpleMath, un wrapper de DirectXMath afin de manipuler en orienté objet les différentes fonctions et type de stockage de DirectXMath.

La classe **Matrix** par exemple va contenir tout un tas de fonction statique qui vont nous faciliter nos transformation: **Matrix::CreatePerspectiveFieldOfView**, **Matrix::CreateWorld**, **Matrix::CreateLookAt**, etc.

DirectXMath - Matrices

Il existe deux type de convention pour les matrices, la pré-multiplication et la post-multiplication, cela influt dans quel sens sont stocké les matrices et dans quel ordre les multiplications successives doivent se faire.

DirectXMath et SimpleMath ont déjà fait ce choix* pour nous:

> Both DirectXMath and SimpleMath use row-major matrices, row vectors, and pre-multiplication.

$$\begin{pmatrix} T_x \\ T_y \\ T_z \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \begin{pmatrix} x & y & z & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix} \begin{pmatrix} T_x \\ T_y \\ T_z \\ 1 \end{pmatrix}$$

Vecteur colonne, matrice colonne, post-multiplication

Vecteur ligne, matrice ligne, pré-multiplication

Où en étions-nous?



Données

Définition et allocation des ressources nécessaire au GPU.

Input Assembler

Assemblage des données fournis par l'utilisateur en primitives (liste de triangle, ruban, ligne, points etc).

Vertex Shader

Transformation des vertices grâce à un programme défini par l'utilisateur (un shader).

Tessellation/Geometry Shader

Modification des primitives soit en les subdivisant (tessellation) soit en les transformant complètement (geometry shader).

Rastérisation

Passage de primitives vectorielles vers image matricielle.
Interpolation des données de vertex.

Effectue également le clipping, la transformation écran et la division par W pour la perspective.

Vertex Shader

Shaders - késako?

Un Shader est un programme défini par l'utilisateur. Il y en a plusieurs tout au long du pipeline et ce sont des étapes intégralement programmable. Elles vont nous permettre le plus de customisation en terme de rendu.

Voyez cela comme un fichier exécutable pour le GPU.

Il en existe de toute sorte:

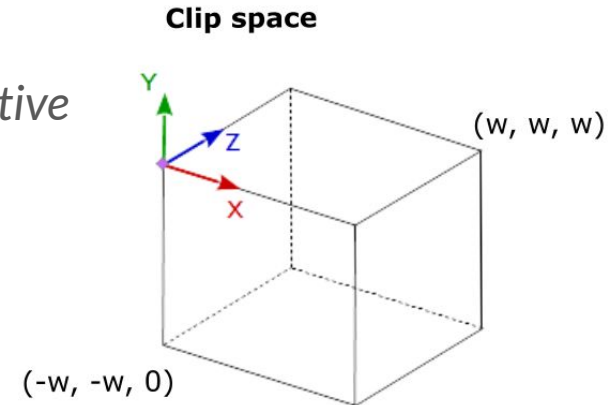
- Vertex Shader
- Pixel Shader
- Compute Shader
- Geometry Shader
- Domain Shader, ...

Vertex Shader - Paramètres

- Prends en entrée:
 - Les vertices de nos primitives générés par l'Input Assembler
 - Des buffers constants envoyés depuis notre moteur
- Renvoie en sortie:
 - Une coordonnée en "Clip Space": `SV_POSITION`
 - N'importe quel données que l'on veut fournir au Pixel Shader plus tard

Vertex Shader - Clip Space

- Espace de coordonnées (*cf image*) que toute les API graphiques attendent en sortie de Vertex Shader. Elle sert au hardware à découper (*clip*) les primitives qui dépasse de la zone visible de l'écran.
- Le GPU appliquera ensuite une division par w (*perspective divide*) qui nous mettra dans ***l'espace de coordonnées d'appareil normée (NDC)***.
 - $[-1; 1]$ en XY et $[0; 1]$ en Z sur DX



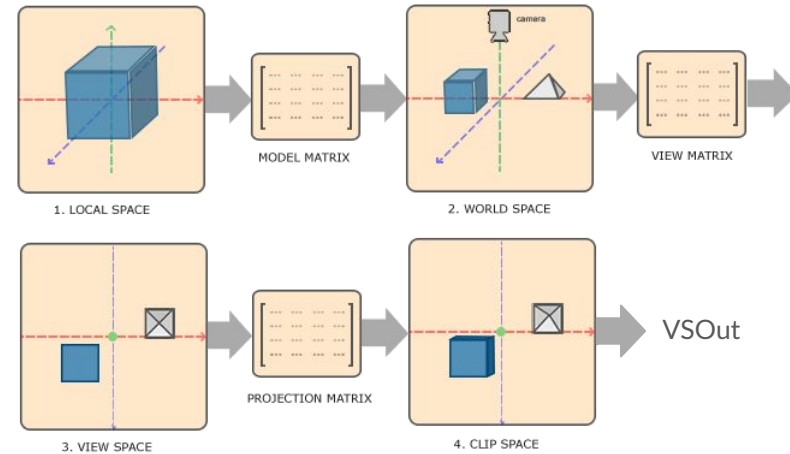
Vertex Shader - Utilité

Le Vertex Shader va nous permettre d'effectuer facilement sur le GPU les transformations de chaque vertex en préparation de la rasterisation.

Notre but va être d'écrire un programme qui passe de coordonnées objets (plus pratique à utiliser pour nous) à des coordonnées Clip Space (requis par le GPU).

C'est là que les matrices de transformation vont être utiles.

On va pouvoir les utiliser pour envoyer de façon compacte nos transformations et de les appliquer en une seule multiplication!



Constant Buffer - Déclaration

Comment allons nous passer nos matrices à notre shader? Avec un autre type de buffer!

Les **Constant Buffers** (CB) vont nous permettre de définir des datas qui seront *constantes* pour chaque invocation de shader (aka chaque vertice/pixel/compute/etc vont recevoir le même buffer).

Pour l'initialiser le plus simple est de passer par une struct contenant les données voulu:

```
struct ModelData {  
    Matrix mModel;  
};
```

Et lors de la description il faudra juste fournir **D3D11_BIND_CONSTANT_BUFFER**:

```
CD3D11_BUFFER_DESC descModel(sizeof(ModelData), D3D11_BIND_CONSTANT_BUFFER)
```

Constant Buffer - Assignment

Comme vu précédemment pour modifier le contenu de votre Constant Buffer avant chaque Draw Call* vous pouvez appeler:

```
ID3D11DeviceContext::UpdateSubresource (  
    (ID3D11Resource*) ptrResource,  
    (UINT) indexSousRessource, // 0 pour un Constant Buffer  
    (D3D11_BOX*) emplacementDestination, // NULL pour un Constant Buffer  
    (void*) data,  
    (UINT) rowPitch, // taille de ligne, 2D/3D uniquement, 0 pour un CB  
    (UINT) depthPitch); // taille de ligne*colonne, res. 3D uniquement
```

Cette fonction peut servir pour n'importe quel type de resource dont les textures

**DirectX 11 fait de la magie dans notre dos en effectuant un "renommage" du buffer, il cree un nouveau buffer en VRAM à chaque appel et stocke temporairement le précédent tant qu'il est utilisé par le GPU. En DirectX 12 ou en Vulkan ce genre de comportement n'existe pas et vous devez vous même fournir un nouveau buffer si vous voulez passer un contenu différent entre deux draw calls, généralement implémentée via un buffer circulaire si ces buffers sont temporaire.*

Constant Buffer - Binding

Une fois le buffer déclaré et rempli il n'y a plus qu'à le bind au pipeline graphique!

De façon similaire à `IASetVertexBuffers/IASetIndexBuffer`, il faut utiliser:

- `VSSetConstantBuffers (`
 `(UINT) slot, (UINT) nombreDeBuffers,`
 `(ID3D11Buffer**) buffersArray)`
- `PSSetConstantBuffers (`
 `(UINT) slot, (UINT) nombreDeBuffers,`
 `(ID3D11Buffer**) buffersArray)`
- `CSSetConstantBuffers, GSSetConstantBuffers,`
 `HSSetConstantBuffers...`

Afin d'envoyer un ou plusieurs buffers au shader voulu.

High-Level Shader Language

Le langage de shader de DirectX est le *High-Level Shader Language* aka HLSL.

Comme beaucoup de langage c'est un cousin du C dans sa syntaxe. Cependant il possède une librairie standard propre extrêmement légère dédiée à la programmation graphique (seulement 140 fonctions, principalement des fonctions mathématiques).

La différence la plus notable par rapport au C sont les types, en plus des scalaires habituels `int`, `uint`, `half`, `float`, etc. Vous avez également des types vectoriels et matriciels.

Ils sont déclarable en rajoutant le nombre de composants voulu (max 4 par dimensions) à la fin de n'importe quel type scalaire. Ex: `float4`, `half3`, `uint3x2`, `bool4x4`

HLSL - Entrée/Sortie

Dans le projet nous avons le shader de base `Basic_vs.hlsl`

Son point d'entrée c'est la fonction `main()` qui prends en paramètre une struct `Input` et renvoie une struct `Output`, ces structs sont intégralement personnalisable.

Nous avons pour l'instant comme input:

```
struct Input {  
    float3 pos : POSITION0; // c'est le "POSITION", 0 de l'Input Layout!  
};
```

Et comme output:

```
struct Output {  
    float4 pos : SV_POSITION; // coord clip space qu'il faut calculer  
};
```

HLSL - Main

La fonction `main()` de `Basic_vs.hlsl` est pour l'instant extrêmement basique, elle initialise à 0 tout les membres de `output` et mets directement `input.pos` dans `output.pos` (en castant vers un `float4` et en mettant w à 1).

C'est ici que l'on va pouvoir rajouter nos transformations avec la fonction de multiplication matriciel:

- `float4 res = mul((float4) vec, (float4x4) mat)`

```
Output main(Input input) {  
    Output output = (Output)0;  
    output.pos = float4(input.pos, 1);  
    // insérer ici multiplication matricielle  
    return output;  
}
```


HLSL - Constant Buffer

Pour passer en entrée un CB on ajoute un **cbuffer** en précisant un **register** de type **b**:

```
cbuffer ModelData: register(b0) { // b0 = VSSetConstantBuffers(0,...)
    float4x4 Model;
};
cbuffer CameraData: register(b1) { // b1 = VSSetConstantBuffers(1,...)
    float4x4 View;
    float4x4 Projection;
};
```

Ici par exemple on déclare que le buffer en slot 0 contiendra notre matrice modèle et celui en slot 1 les matrices vue et projection.

C'est une bonne pratique de mettre les données qui change peu entre deux draw call (comme par exemple ici les matrices caméra) dans le même CB. Il pourra ainsi rester bindé sur le GPU.

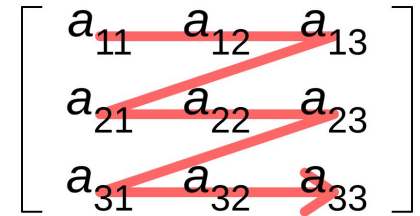
HLSL - Stockage de matrices

En HLSL nous utilisons le type `float4x4` pour utiliser nos matrices. Et en C++ nous allons utiliser `SimpleMath::Matrix` pour nous faciliter leurs créations comme vu précédemment.

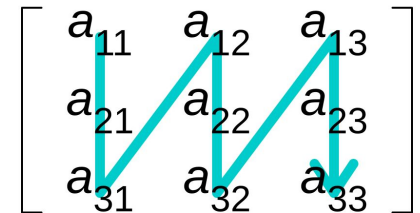
Cependant il y a une différence dans le stockage! HLSL s'attend par défaut à recevoir des matrices **stockées en colonne** pour optimiser les calculs mais SimpleMath les **stocke en ligne**.

Nous avons plusieurs solutions pour régler ce soucis, nous pouvons demander à HLSL de stocker les matrices dans l'autre sens avec le keyword `row_major`, appeler `Matrix::Transpose()` avant l'upload ou alors faire nos maths comme si c'était des matrices colonne en HLSL et de bénéficier d'une transposition gratuite!

Row-major order



Column-major order



TP 4

Matrice de transformation

HLSL - Matrices de transformation

A partir du TP précédent:

- Définissez en C++ et en HLSL les structs `ModelData` et `CameraData`
- Allouez les deux Constant Buffer
- Créez les matrices Modèle, Vue et Projection à l'aide de `SimpleMath::Matrix`
- Dans `Render()` appelez `ID3D11DeviceContext::UpdateSubresource` pour uploader vos matrices.
 - `/!\` Appeler `Transpose()` avant l'upload pour avoir le bon stockage!
- Modifiez `basic_vs.hlsl` pour y ajouter la transformation local vers écran
 - Fonction `float4 output = mul((float4) vec, (float4x4) mat)` en HLSL.
Rappel: ce n'est pas commutatif! Comme nos matrices viennent de SimpleMath on utilise la pré-multiplication (on mets le vecteur que l'on souhaite modifier avant la matrice)!
- Modifiez la matrice projection dans `OnWindowSizeChanged()`, et la matrice vue dans `Update()` en réagissant au clavier/souris

TP 5

Cleanup!

Cleanup - VertexLayout.h

Je vous fourni le fichier Engine/VertexLayout.h qui contient des layouts commun

- Enlever de Game.cpp toute les références à `ID3D11InputLayout`
 - Changez votre `std::vector<float>` en `std::vector<VertexLayout_Position>`
 - Utilisez `GenerateInputLayout` & `ApplyInputLayout` fourni par VertexLayout.h pour la création et le bind du layout
- Nous allons ensuite rajouter un fichier Engine/Buffer.h et implémenter les classes `VertexBuffer<TVertex>`, `IndexBuffer` et `ConstantBuffer<TData>` afin de nous faciliter la création et la modification de ressource

/!\ Le projet utilise premake, n'oubliez pas de lancer `makeSolution.bat` quand vous rajoutez des fichiers.

Cleanup - Buffer.h

- Une classe templaté VertexBuffer avec:
 - `uint32_t PushVertex(TVertex v)`
 - `void Create(DeviceResources* deviceRes)`
 - `void Apply(DeviceResources* deviceRes, int slot = 0)`
- Une classe IndexBuffer avec:
 - `void PushTriangle(uint32_t a, uint32_t b, uint32_t c)`
 - `uint32_t Size()`
 - `void Create(DeviceResources* deviceRes)`
 - `void Apply(DeviceResources* deviceRes)`
- Une classe templaté ConstantBuffer avec:
 - Un membre : `TData data`
 - `void Create(DeviceResources* deviceRes)`
 - `void UpdateBuffer(DeviceResources* deviceRes)` : upload le membre data
 - `void ApplyToVS(DeviceResources* deviceRes, int slot = 0)`
- Intégrez ces nouvelles classes à Game.cpp

Input Assembler

Assemblage des données fournis par l'utilisateur en primitives (liste de triangle, ruban, ligne, points etc).

Vertex Shader

Transformation des vertices grâce à un programme défini par l'utilisateur (un shader).

Tessellation/Geometry Shader

Modification des primitives soit en les subdivisant (tessellation) soit en les transformant complètement (geometry shader).

Rastérisation

Passage de primitives vectorielles vers image matricielle.
Interpolation des données de vertex.

Effectue également le clipping, la transformation écran et la division par W pour la perspective.

Pixel Shader (aka Fragment Shader)

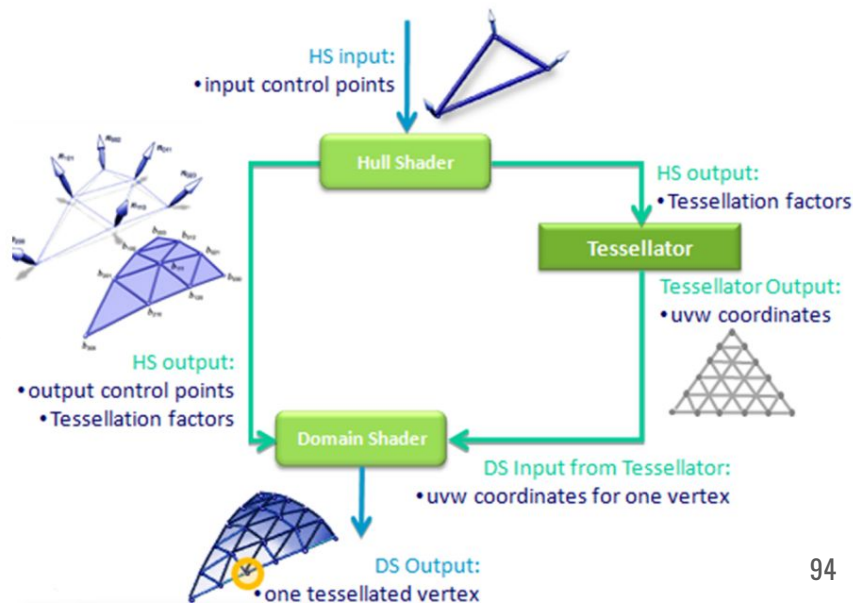
Calcul de la valeur de chaque pixel obtenu à l'étape précédente grâce à un programme défini par l'utilisateur (un shader).

Tessellation

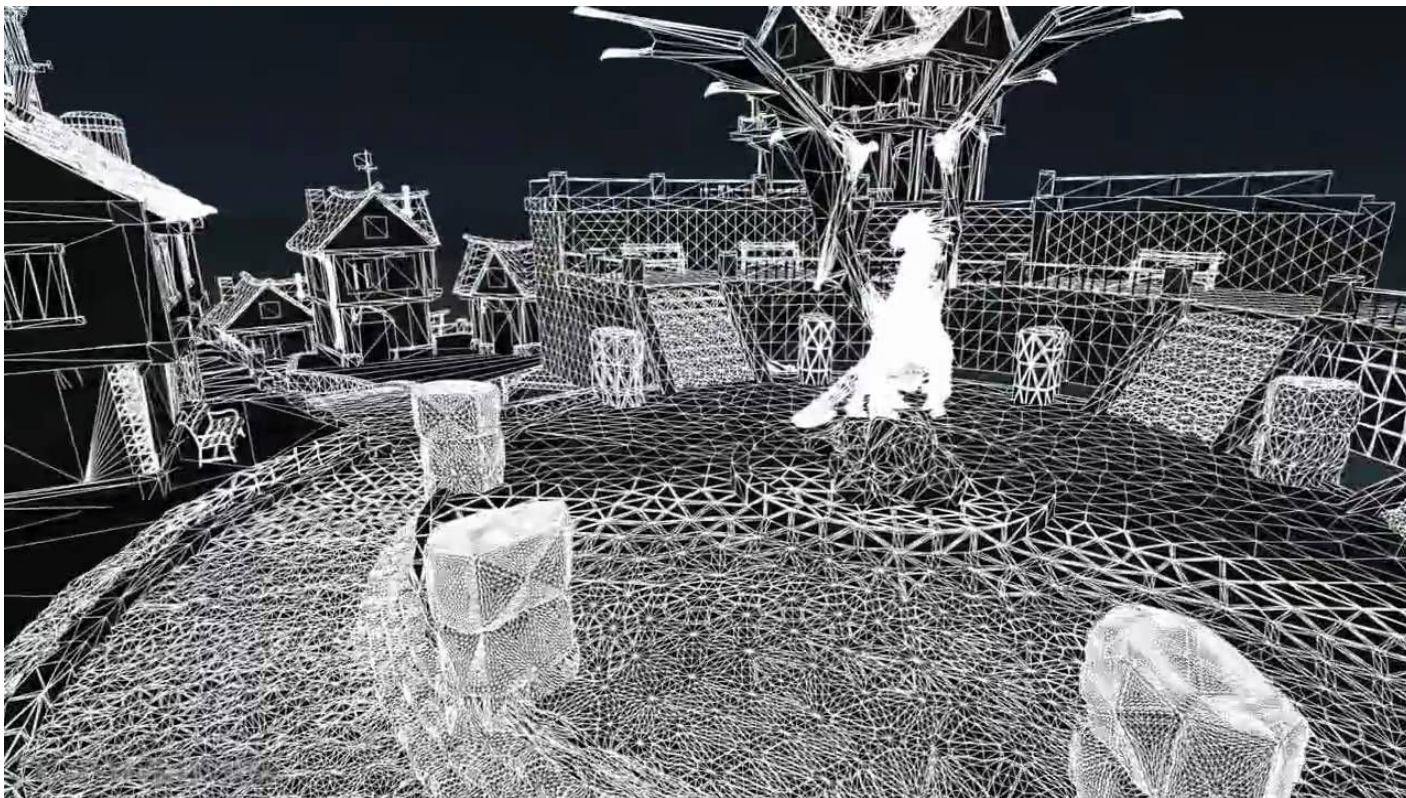
L'étape de tessellation va nous permettre de rajouter des vertices à notre maillage procéduralement, c'est une étape partiellement programmable avec 2 shaders et une fonction fixe:

- Hull Shader: considère les vertices comme des "points de contrôle" et produit en sortie les paramètres pour l'étape de tessellation.
- Tessellator: une fonction fixe qui va sous-diviser la géométrie existante et créer des nouvelles vertices et primitives.
- Domain Shader: modification des vertices nouvellement créée.

Tessellation Pipeline



Tessellation



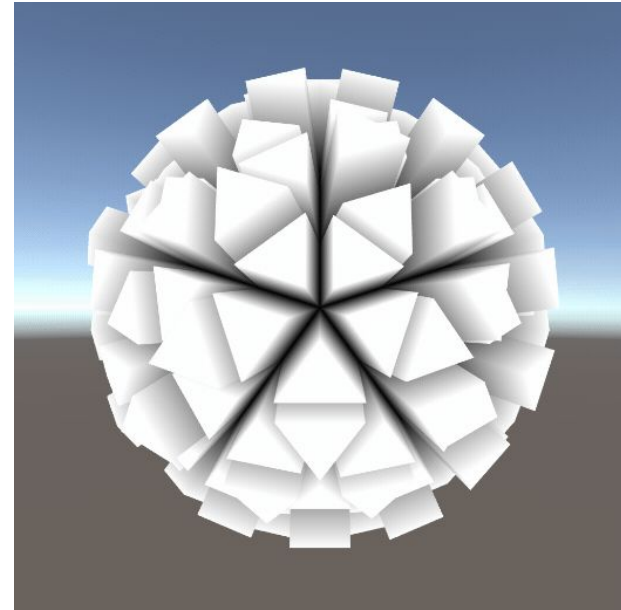
Geometry Shader

Étape entièrement programmable qui va nous permettre de manipuler et de modifier nos primitives.

Il prends en entrée **toutes** les vertices d'une primitive (contrairement à un vertex shader) et renvoie en sortie de nouvelles vertices/primitives (topologie et nombre d'output défini à l'avance).

Utile pour faire des effets d'extrusion, généré de l'herbe, etc

De moins en moins utile de nos jours, on préférera utiliser des Compute Shaders car ils nous donnent plus de contrôle



Vertex Shader

Transformation des vertices grâce à un programme défini par l'utilisateur (un shader).

Tessellation/Geometry Shader

Modification des primitives soit en les subdivisant (tessellation) soit en les transformant complètement (geometry shader).

Rastérisation

Passage de primitives vectorielles vers image matricielle.
Interpolation des données de vertex.

Effectue également le clipping, la transformation écran et la division par W pour la perspective.

Pixel Shader (aka Fragment Shader)

Calcul de la valeur de chaque pixel obtenu à l'étape précédente grâce à un programme défini par l'utilisateur (un shader).

Output Merger (aka Color Blending)

Combine le pixel fraîchement généré avec le contenu déjà présent à l'écran à cet endroit. Configurable pour effectuer différent type d'effets.

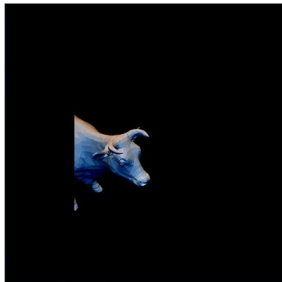
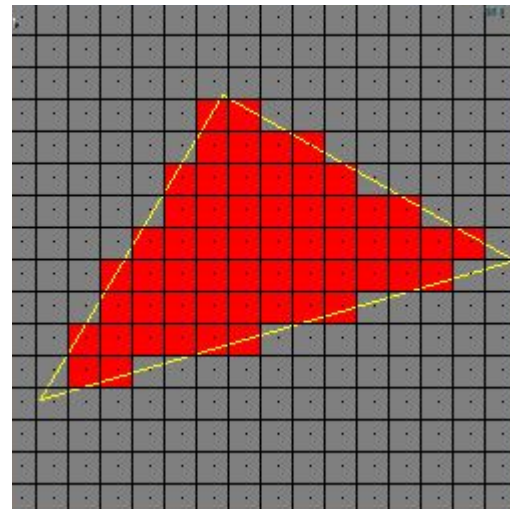
Permet également de tester le buffer de profondeur.

Rôle du rasterisateur

Jusqu'à présent nous sommes resté dans un monde géométrique, la rasterisation est une fonction fixe du pipeline qui va déterminer quel pixels sont à colorier à partir de la géométrie fourni.

C'est à cette étape que s'effectue le backface culling, et le scissor culling.

C'est aussi le rasterisateur qui va effectuer le passage de coordonnées NDC à des coordonnées viewport. Ainsi que faire la division par w pour la perspective.



Clipping with scissor test



No scissor test

Paramétrage de rastérisation en DirectX11

Même s'il s'agit d'une fonction fixe elle est paramétrable.

Nous pouvons décrire grâce à `D3D11_RASTERIZER_DESC` les paramètres de rastérisation puis utiliser:

```
ID3D11Device::CreateRasterizerState (
    (D3D11_RASTERIZER_DESC*) descriptionRasterisateur,
    (ID3D11RasterizerState**) adresseFuturRSState);
```

Pour créer un `ID3D11RasterizerState*`.

On pourra ensuite la passer au context avec :

```
ID3D11DeviceContext::RSSetState ((ID3D11RasterizerState*) state)
```

Viewport & Scissor

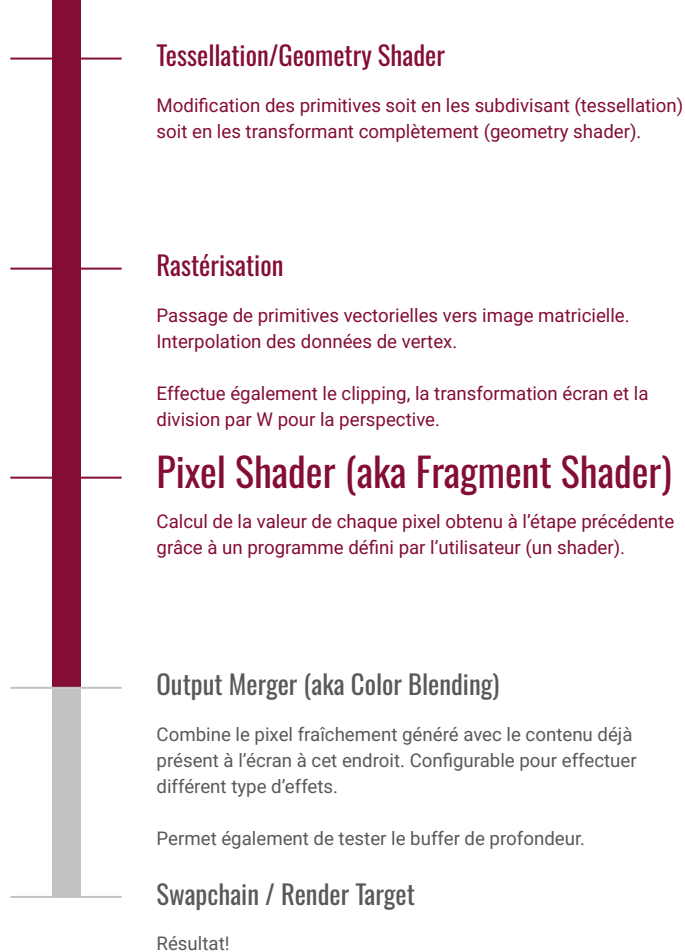
Nous avons également:

```
ID3D11DeviceContext::RSSetViewports( // déjà de base dans Game.cpp!  
    (UINT) nombreViewports,  
    (D3D11_VIEWPORT*) tableauViewports);
```

Qui permet de définir la taille de notre fenêtre. Cela prend un tableau en paramètre, cependant par défaut tout sera tracer uniquement dans le premier viewport (un geometry shader est nécessaire pour tracer dans les autres).

```
Et: ID3D11DeviceContext::RSSetScissorRects(  
    (UINT) nombreRectangles,  
    (D3D11_RECT*) tableauRectanlge);
```

Qui permet de définir quelle zone de l'écran est traçable (par défaut tout l'écran l'est, cette fonction doit être activé dans `D3D11_RASTERIZER_DESC`). Même contrainte que plus haut, seul le premier rectangle sera utilisé si aucun geometry shader n'est utilisé.



Pixel Shader

Nous arrivons bientôt au bout de la chaîne de rendu! Maintenant que la rasterisation a défini quel pixels sont à afficher, le GPU va pouvoir invoquer une instance du Pixel Shader pour chacun de ces pixels!

Les attributs que l'on a passé en sortie de notre Vertex Shader vont être **interpolé** tout le long de la primitive par la rasterisation. Cela va nous permettre d'avoir automatiquement des dégradés de couleur quand on utilise **COLOR0** par exemple, mais c'est aussi grâce à cela que nous allons pouvoir sampler des textures en interpolant des **TEXCOORD**!

*Side note: même si 80% du temps nous allons sampler les textures via une coordonnées UV passé en attribut **TEXCOORD** absolument rien ne nous empêche de la sampler avec n'importe quel variable qui nous passe sous la main!*

Semantics

Depuis le début du cours nous utilisons des mots clefs dans nos shaders pour communiquer l'usage de chacun de nos attributs (`POSITION0`, `COLOR0`, etc).

Une liste complète de ces attributs sont contenu dans la doc :

<https://learn.microsoft.com/en-us/windows/win32/direct3dhls/dx-graphics-hlsl-semantics>

Certaines de ces semantics vont permettre de communiquer avec le GPU et de paramétrer le pipeline, ce sont celles qui commence par `SV_` pour System Values.

Je vous recommande de lire ce petit bout de doc pour vos expérimentation!

TP 6

Cube coloré

HLSL - Pixel Shader

Créez une classe Cube qui aura comme méthode:

- Un constructeur qui prends une position en paramètre
- `void Generate(DeviceResources* deviceRes)` qui sera en charge de créer les buffers et de les remplir
- `void PushFace(Vector3 pos, Vector3 up, Vector3 right)`
- `void Draw(DeviceResources* deviceRes)`

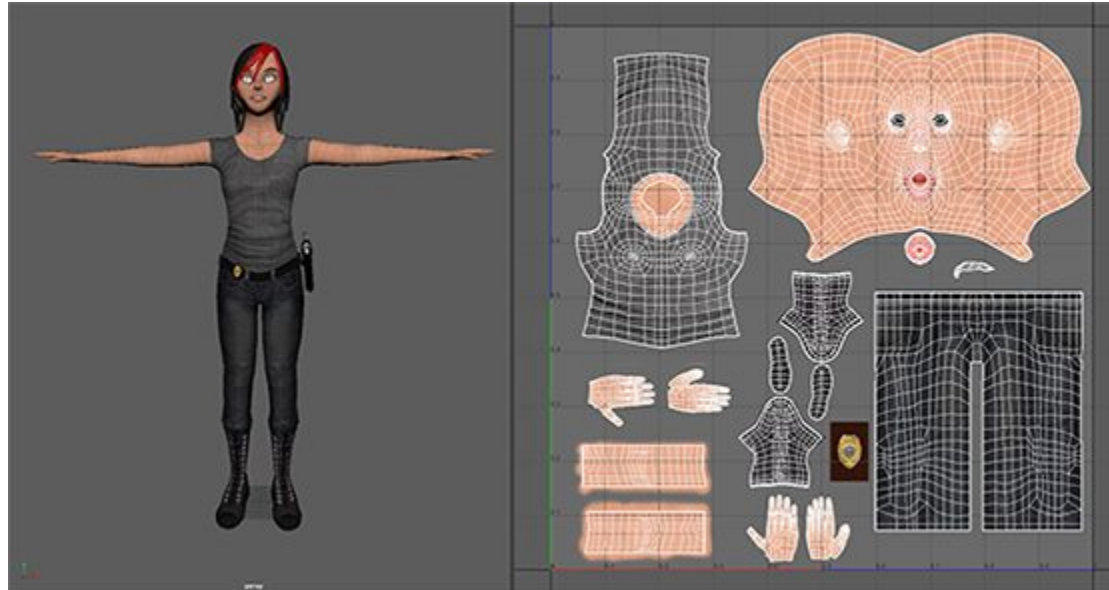
Afin de préparer le terrain pour les textures nous allons passer des coordonnées de textures à notre cube, utilisez donc `VertexLayout_PositionUV` pour votre VB.

Modifiez dans `basic_vs.hlsl` les structs Input et Output ainsi que main() afin de fournir le nouvel attribut au pixel shader.

Modifiez dans `basic_ps.hlsl` Input et main pour affichez à l'écran les coordonnées de texture.

Texture & UVs

Afin de plaquer une texture sur notre objets 3D nous allons avoir besoin de coordonnées 2D pour faire la correspondance et pour savoir quelle couleur affiché.



Textures & Samplers en DirectX11

Pour créer nos textures on va laisser faire DirectXTK avec:

```
#include "DDSTextureLoader.h"
```

```
DirectX::CreateDDSTextureFromFileEx(  
    (ID3D11Device*) d3dDevice,  
    (const wchar_t*) fileName,  
    (size_t) maxsize, // 0 pour ne pas limiter la taille  
    (D3D11_USAGE) usage, // généralement D3D11_USAGE_DEFAULT  
    (unsigned int) bindFlags, // généralement D3D11_BIND_SHADER_RESOURCE  
    (unsigned int) cpuAccessFlags, // généralement 0  
    (unsigned int) miscFlags, // généralement 0  
    (DDS_LOADER_FLAGS) loadFlags,  
    (ID3D11Resource**) outTexture,  
    (ID3D11ShaderResourceView**) outTextureView,  
    (DDS_ALPHA_MODE*) outAlphaMode) // optionnel
```

Textures & Samplers en DirectX11

Et pour définir quelles textures et quels samplers sont utilisés dans le pixel shader:

```
ID3D11DeviceContext::PSSetShaderResources (  
    (UINT) slot,  
    (UINT) nombreRessources,  
    (ID3D11ShaderResourceView**) tableauRessources);
```

Et:

```
ID3D11DeviceContext::PSSetSamplers (  
    (UINT) slot,  
    (UINT) nombreSamplers,  
    (ID3D11SamplerState**) tableauSamplers);
```

Texture & Samplers

```
Texture2D tex : register(t0); // t0 représente le slot 0 de
SamplerState samplerState : register(s0);

struct Input {
    float4 pos : SV_POSITION;
    float2 uv : TEXCOORD0;
};

float4 main(Input input) : SV_TARGET {
    float4 color = tex.Sample(samplerState, input.uv);
    return color;
}
```


TP 7

Cube texturé

TP 8

FPS Camera

TP 9

Minicraft - World

TP 10

Minicraft - Chunk

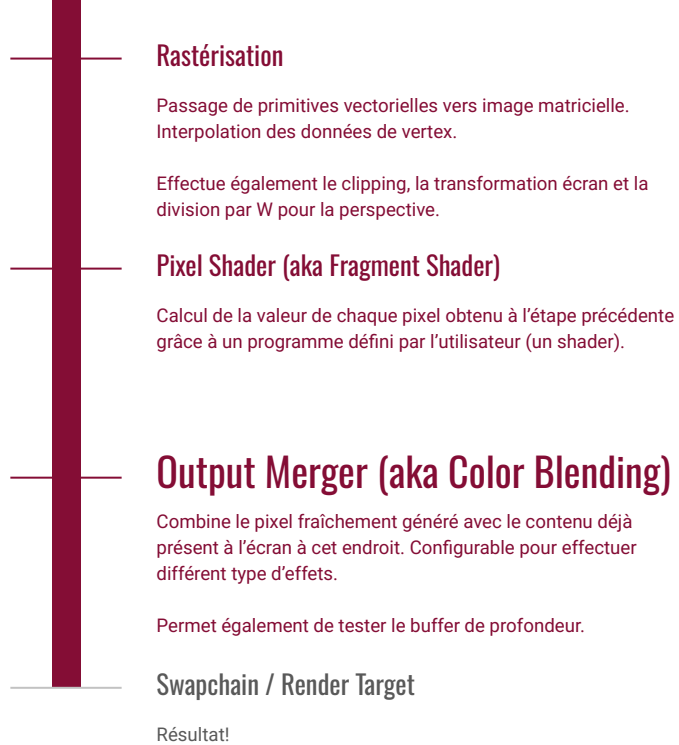
Analogie

Cube.h



Chunk.h





Blending

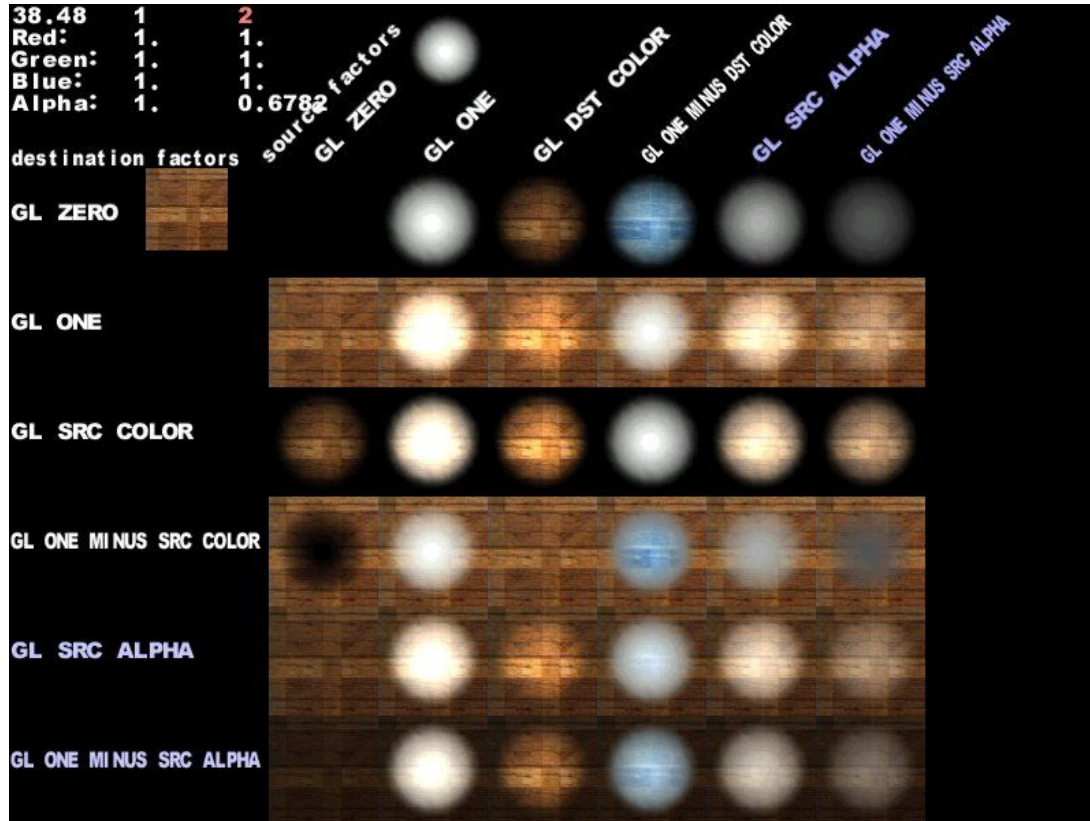
L'output merger est une étape configurable du pipeline, elle a pour but de stocker nos pixels fraîchement calculé dans notre **Render Target** (voyez cela comme une texture comme les autres mais dans laquelle on écrit).

L'une des opérations performé par l'output merger est le blending, il prends le nouveau pixel (la source) et le pixel déjà présent (la destination) et effectue ce calcul:

```
(source_R*SRC_BLEND_FACTOR) + (dest_R*DEST_BLEND_FACTOR) = resultat_R  
(source_G*SRC_BLEND_FACTOR) + (dest_G*DEST_BLEND_FACTOR) = resultat_G  
(source_B*SRC_BLEND_FACTOR) + (dest_B*DEST_BLEND_FACTOR) = resultat_B  
(source_A*SRC_ALPHA_BLEND_FACTOR) + (dest_A*DEST_ALPHA_BLEND_FACTOR) = resultat_A
```

Les facteurs et l'opération sont configurables pour effectuer tout un tas d'effets.

Blending



Le mode que l'on va utiliser le plus est l'alpha blending.

En utilisant les facteurs

`D3D11_BLEND_SRC_ALPHA` et `D3D11_BLEND_INV_SRC_ALPHA` nous faisons un lerp d'une couleur à l'autre selon la transparence!

Si nous avons des alpha prémultipliée il nous faudra utiliser

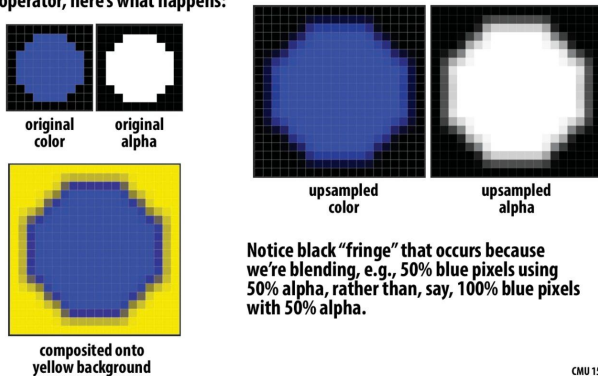
`D3D11_BLEND_ONE` et `D3D11_BLEND_INV_SRC_ALPHA`

Premultiplied Alpha

Pour stocker nos textures nous allons “prémultiplier” la transparence. Autrement dit nous allons appliquer l’alpha directement dans la valeur RGB. L’intérêt sera d’avoir un filtrage correct et d’éviter les soucis de frange quand plusieurs sprites se superpose.

A problem with non-premultiplied alpha

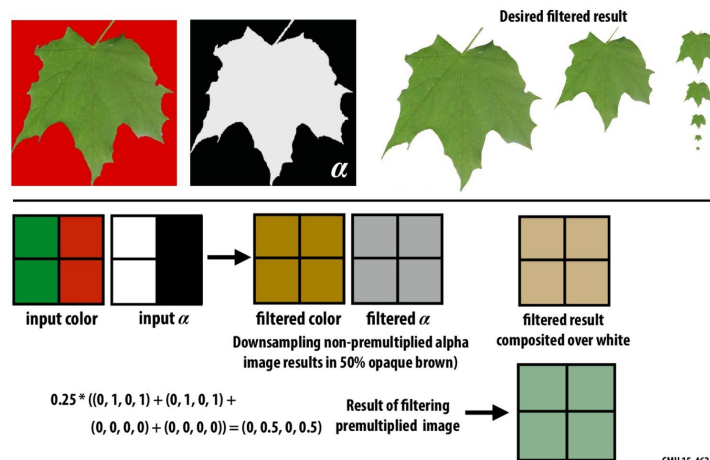
- Suppose we upsample an image w/ an alpha mask, then composite it onto a background
- How should we compute the interpolated color/alpha values?
- If we interpolate color and alpha separately, then blend using the non-premultiplied “over” operator, here’s what happens:



CMU 15-462/662

Similar problem with non-premultiplied alpha

Consider pre-filtering (downsampling) a texture with an alpha matte



CMU 15-462/662

Paramétrage de l'output merger en DirectX11

Nous pouvons décrire grâce à `D3D11_BLEND_DESC` les paramètres de blending puis utiliser:

```
ID3D11Device::CreateBlendState(  
    (D3D11_BLEND_DESC*) descriptionBlendState,  
    (ID3D11BlendState**) adresseFuturBlendState);
```

Pour créer un `ID3D11BlendState*`.

On pourra ensuite la passer au context avec :

```
ID3D11DeviceContext::OMSetBlendState(  
    (ID3D11BlendState*) blendState,  
    (const FLOAT[4]) blendFactor,  
    (UINT) sampleMask)
```

Le paramètre `blendFactor` est optionnel et nous permet de définir quels valeurs seront utilisés avec un `D3D11_BLEND_BLEND_FACTOR` et `D3D11_BLEND_INV_BLEND_FACTOR`

