Game Rendering 101

D’un modèle 3D vers une image 2D

Présentation

Qui je suis?

- “Dev à tout faire” chez Headbang Club depuis 6 ans
- Spécialisé en portage, ship 17 releases sur 6 jeux (Nintendo Switch, PS4, PS5, Xbox One, Xbox Series, Windows Store) + le portage d’un moteur de jeu entier sur Xbox One.
- Unity, Unreal, MonoGame et surtout moteur maison
- Affinité avec la programmation graphique

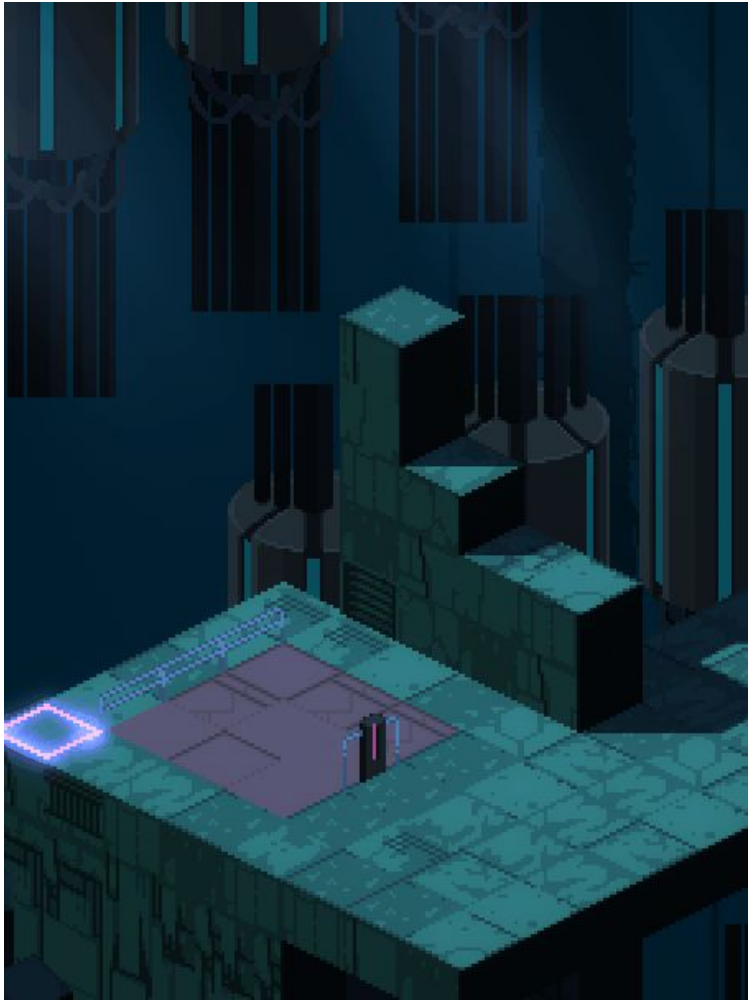

WINDJAMMERS 2
™

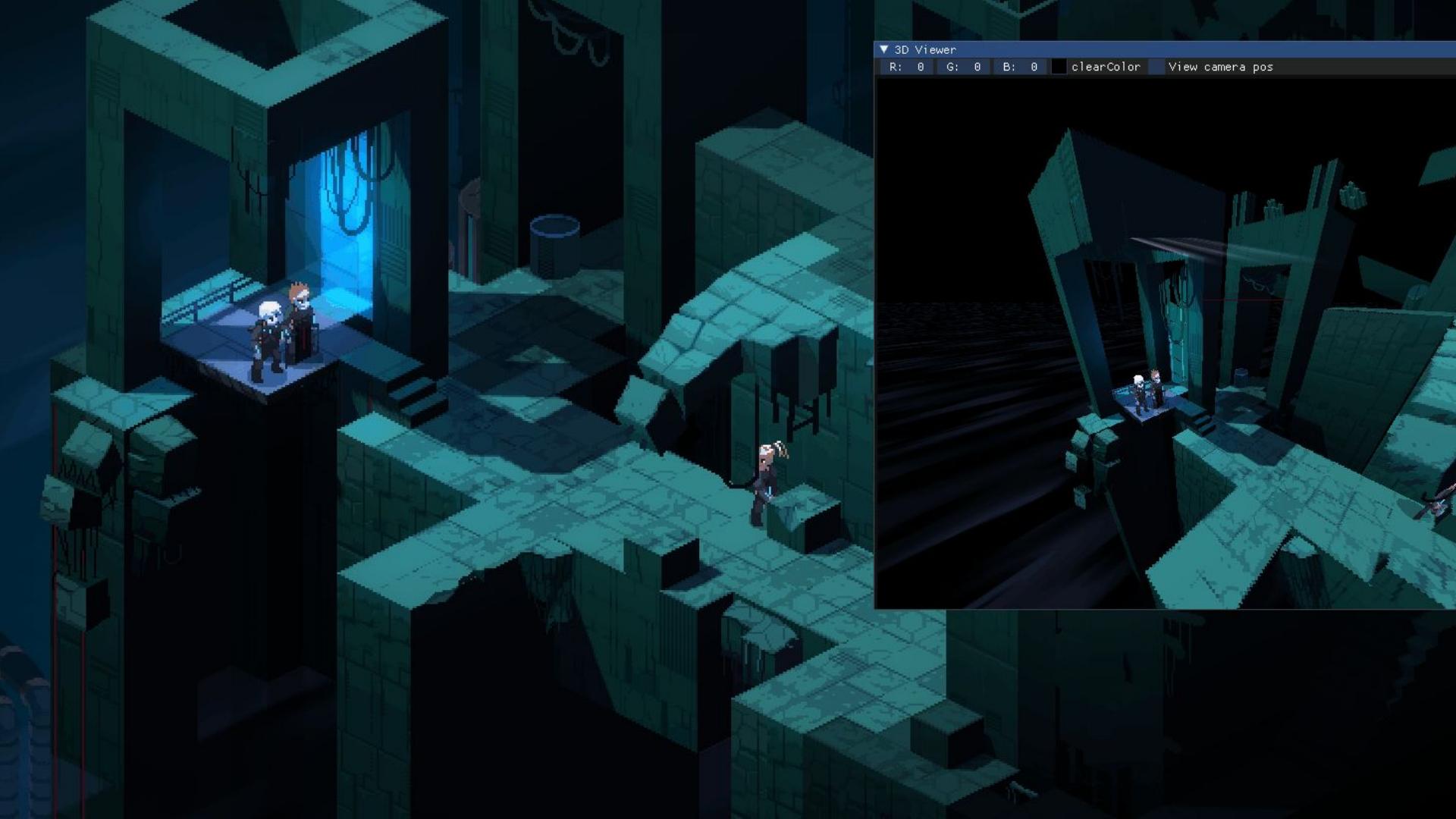

SHOTGUN KING
THE FINAL CHECKMATE


WARHAMMER
40,000
MECHANICUS

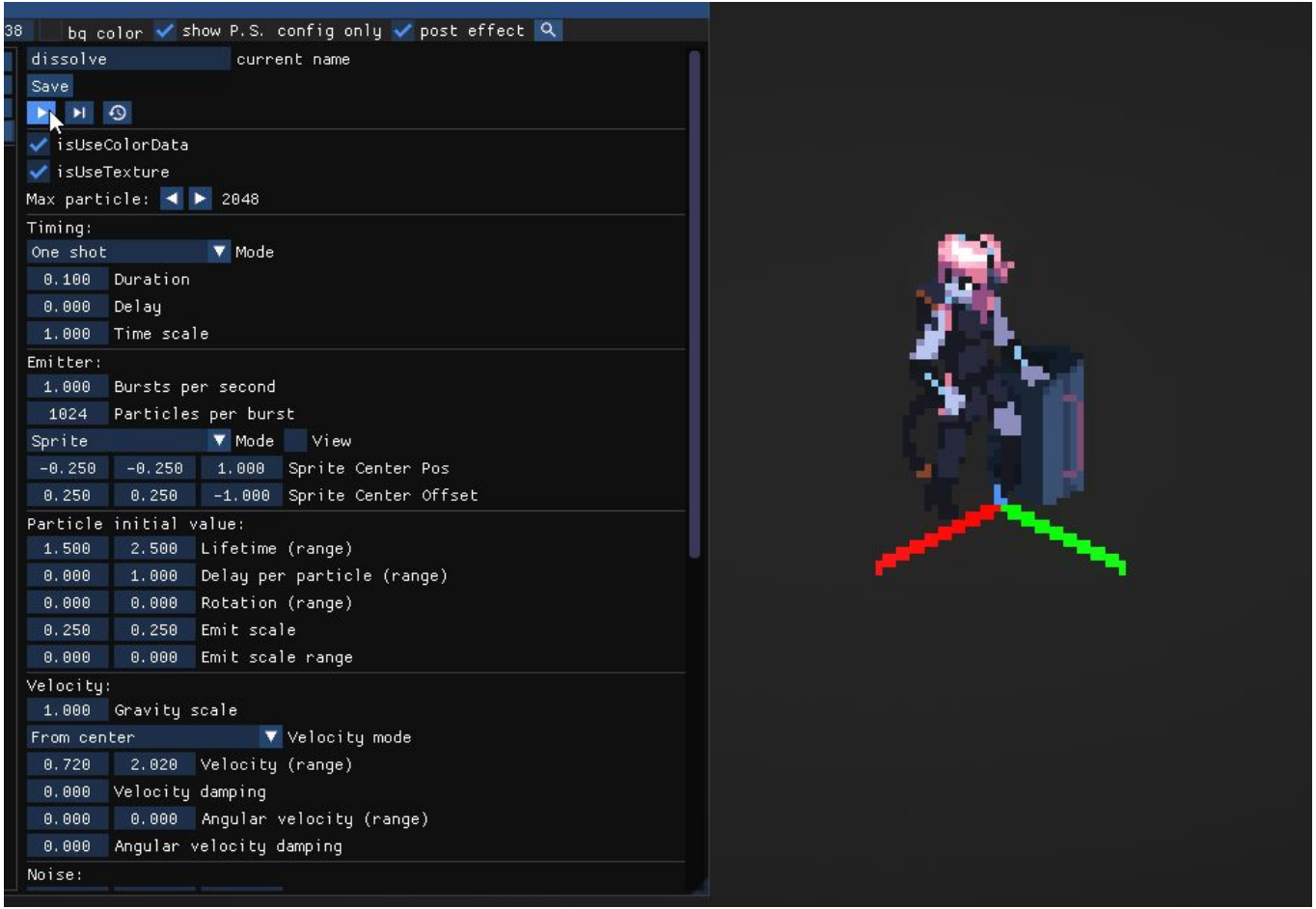

BIG
HELMET
HEROES


DEATH
TOWER




3D Viewer
R: 0
G: 0
B: 0
clearColor
View camera pos


World infos
Enable Debug
Game
Modes
Turns
Data
Explore
Inputs
Entities
Post Processing Presets
Light
R: 77
G: 77
B:102
Ambient color
Directional Light:
-89.0°
Azimuth
0.800
Altitude
R:223
G:223
B:191
Color
0.800
Intensity
Point Lights:
#0:
2487.900
Intensity
R:255
G: 0
B: 0
Color
1.700
2.200
-4.300
Position
25.000
Distance
#1:
0.000
Intensity
#2:
0.000
Intensity
#3:
0.000
Intensity
TRpg
Debug
allowAIStart
Archived Steps


38
bq color
show P.S. config only
post effect
dissolve
current name
Save
isUseColorData
isUseTexture
Max particle:
2048
Timing:
One shot
Mode
0.100
Duration
0.000
Delay
1.000
Time scale
Emitter:
1.000
Bursts per second
1024
Particles per burst
Sprite
Mode
View
-0.250
-0.250
1.000
Sprite Center Pos
0.250
0.250
-1.000
Sprite Center Offset
Particle initial value:
1.500
2.500
Lifetime (range)
0.000
1.000
Delay per particle (range)
0.000
0.000
Rotation (range)
0.250
0.250
Emit scale
0.000
0.000
Emit scale range
Velocity:
1.000
Gravity scale
From center
Velocity mode
0.720
2.020
Velocity (range)
0.000
Velocity damping
0.000
0.000
Angular velocity (range)
0.000
Angular velocity damping
Noise:

Planning
Structure du cours

Planning du cours

Lundi
- Matin: Introduction et présentation du pipeline graphique
- Après-midi: mise en place du projet et premier dessin 3D

Mardi
- Matin: Matrices et modèle 3D
- Après-midi: Textures, Shaders

Mercredi
- Matin: Lancement du TP Minicraft, architecture du projet
- Après-midi: Chunk batching

Planning du cours

Jeudi
- Matin: Génération procédural et ImGui
- Après-midi: Lumière et transparence

Vendredi
- Matin: Suivi TP et physique
- Après-midi: Suivi TP et culling

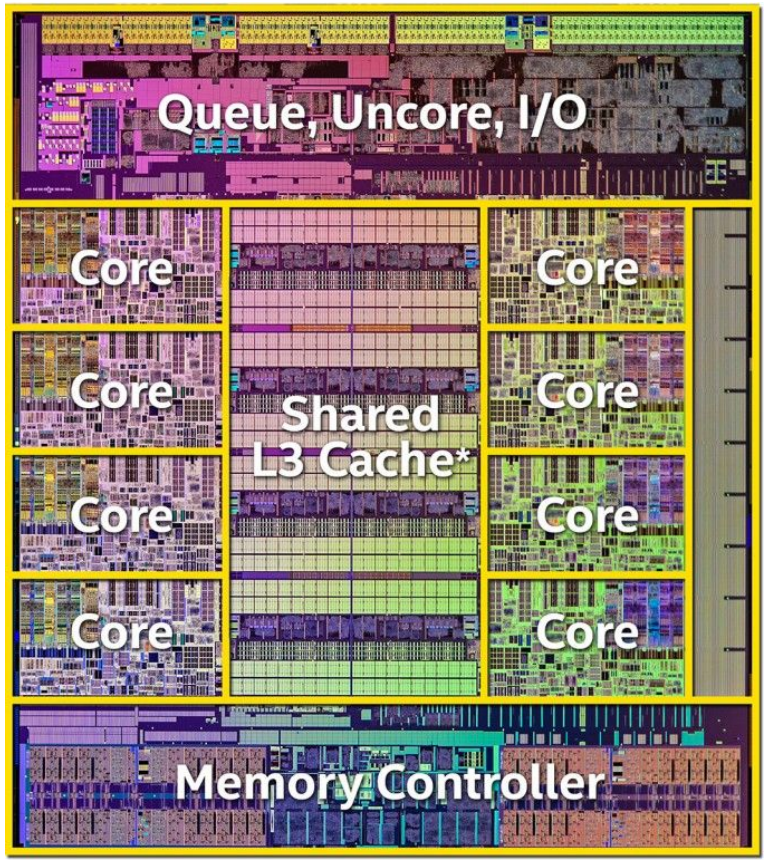
Mercredi/Jeudi
- Matin: Suivi TP
- Après-midi: Suivi TP

Introduction

GPU quésaco?

Architecture CPU

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



Architecture GPU

- 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.

Microsoft® DirectX®11

OpenGL ES, NVN, AGC, GNM, …

DirectX 11

Ce cours utilisera **DirectX 11**

Plusieurs raisons:
Meilleurs outils, meilleurs compatibilité, API plus simple à utiliser (orienté objet, gestion mémoire simplifié, 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 utile:

- `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

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 lumières, transparence etc)

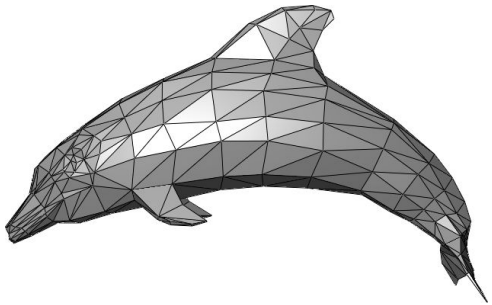
Intéressons nous au **maillage**

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:

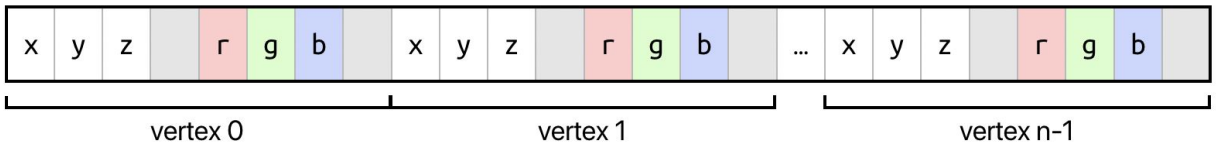
- Une position
- Une ou plusieurs coordonnées de textures
- Une couleur
- Une normal/tangente/binormal
- Des poids et des indices pour les os de l’armature
- À-peu-près tout ce que vous voulez vraiment



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**) addresseFuturBuffer); // 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'a 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.

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

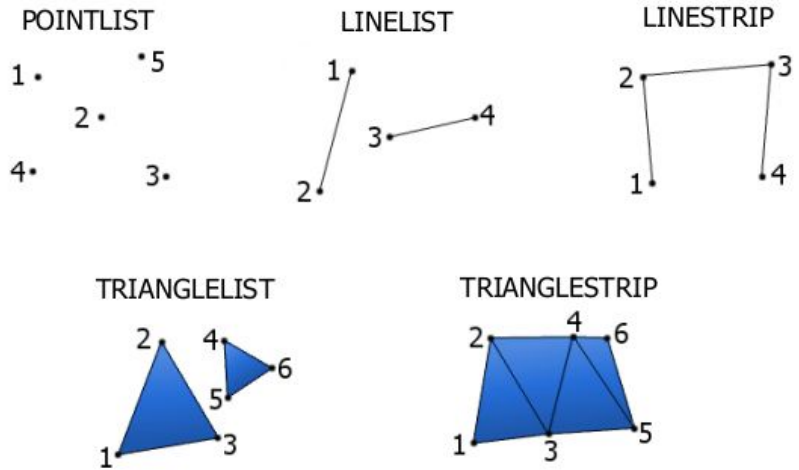
Modification des primitives soit en les subdivisant (tessellation) soit en les transformant complètement (geometry shader).

Rôle de l'Input Assembler

On a maintenant un tas de vertex, mais il ne sont pas encore connecté 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)



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 etc. Pour cela on va définir un Input Layout.

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, (UINT) inputSlot,
    (UINT) alignement, // optionnel, sinon D3D11_APPEND_ALIGNED_ELEMENT
    (UINT) inputSlotClass, // typiquement D3D11_INPUT_PER_VERTEX_DATA
    (UINT) incrementParInstance}, // inutile avec PER_VERTEX_DATA

    // insérer les autres types de données envoyé
}
```

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) nombreElements,
    (void*) vertexShaderBytecode, (size_t) bytecodeLength,
    (ID3D11InputLayout**) addresseFuturlayout); // 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(float) * 3 };
UINT offsets[] = { 0 };
context->IASetVertexBuffers(0, 1, vbs, strides, offsets);
```
```

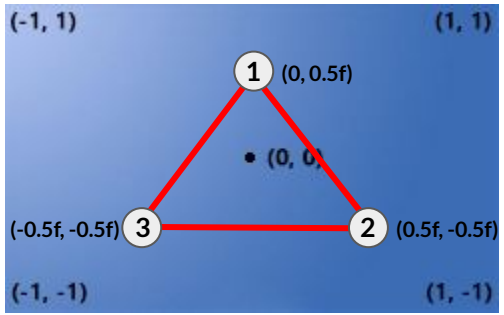
TP 1

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



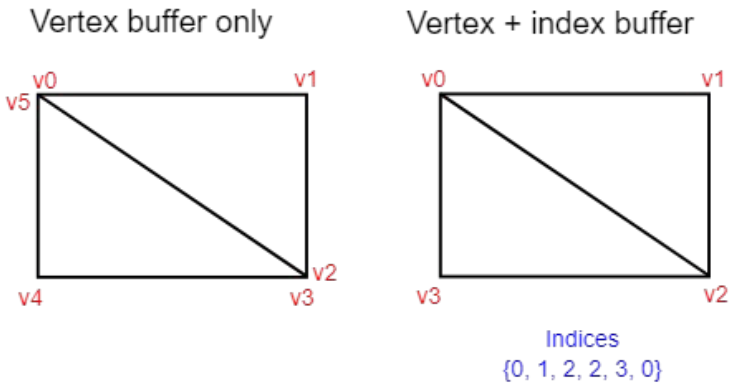
Index Buffer

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.



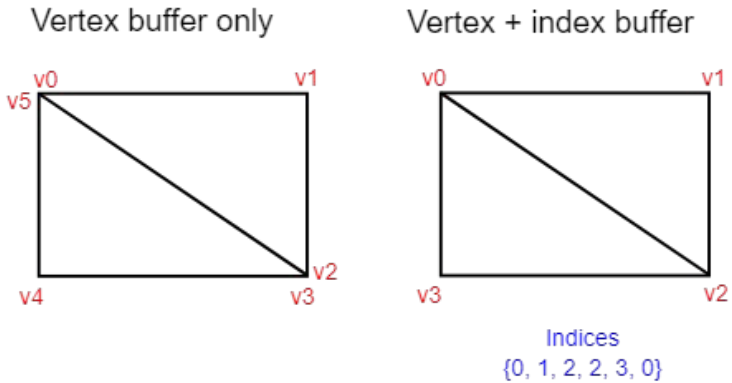
TP 2

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 `IASetIndexBuffers` puis appelez `context->DrawIndexed(6, 0, 0)` pour afficher votre triangle
  `DXGI_FORMAT_R32_UINT`



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 Z pour la perspective.

Mais juste avant:

Une petite pause math rapide

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

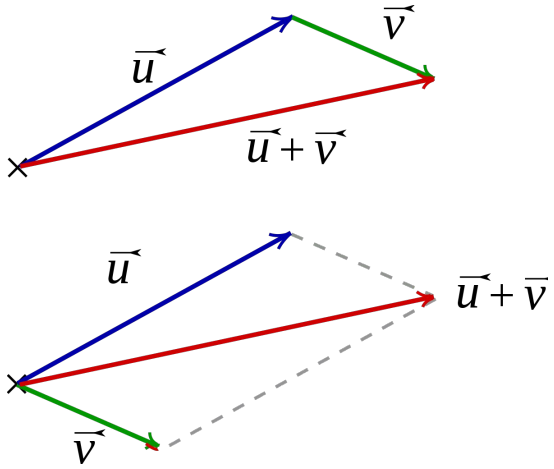
- **Addition/Soustraction** par un autre vecteur
- **Multiplication/Division** par un scalaire
- Produit scalaire *(dot product)*
- Produit vectoriel *(cross product)*
- Obtenir la norme (longueur) du vecteur

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

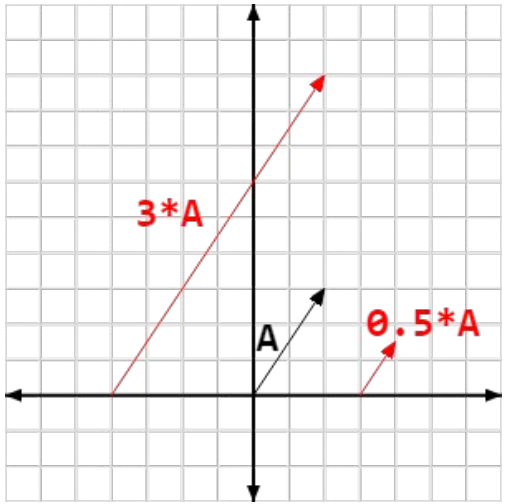
Un vecteur peut être décomposé en somme de vecteur aligné sur les axes du plan, en cela additionner 2 vecteur revient à ajouter les coordonnées



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)

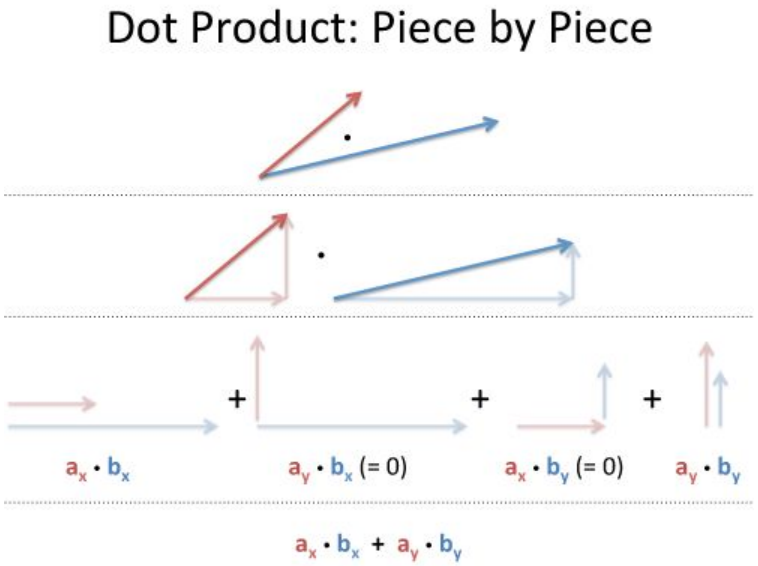
Produit scalaire

Le produit scalaire se forme comme ceci:

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

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

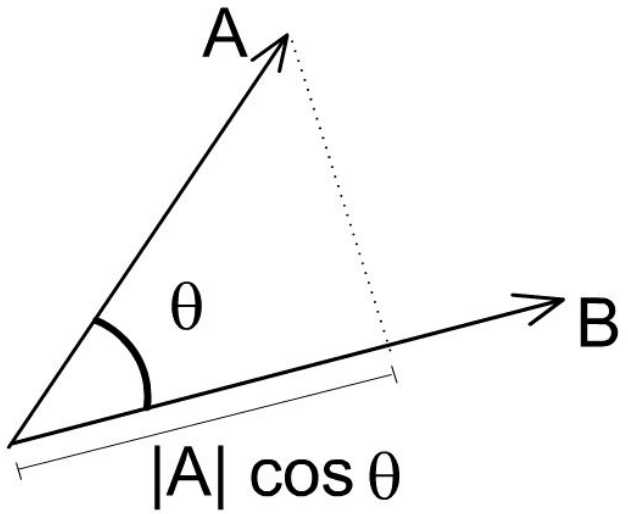
Notamment **2 vecteur orthogonaux ont un produit scalaire égale à 0**. Pratique pour simuler un ombrage (plus sur ça quand on arrivera au pixel shader)



Produit scalaire

Autre représentation du produit scalaire:

$$u \cdot v = ||u|| \cdot ||v|| + \cos(\theta)$$

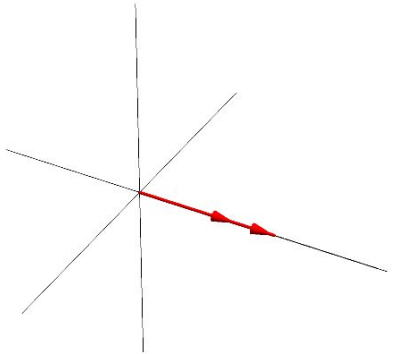


Produit vectoriel

Le produit vectoriel se forme comme ceci:

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

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



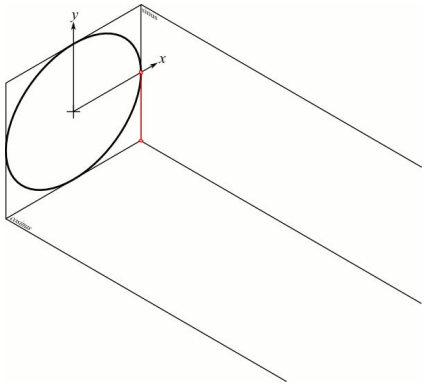
Il permet (entre autre) de calculer la normal d'un plan défini par 2 vecteurs

Rotations

Rotation en 2D

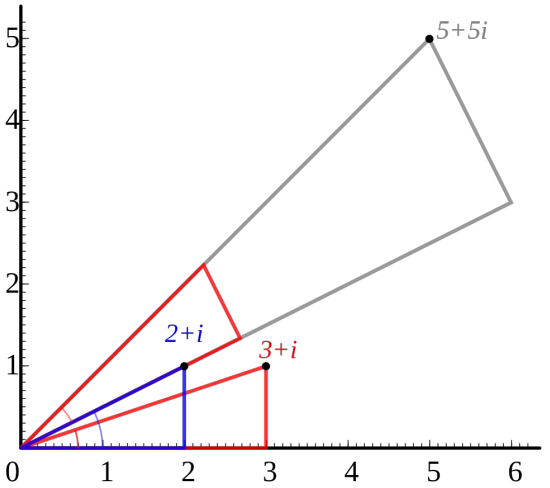
On va utiliser les principes de bases de la trigonométrie pour effectuer une rotation en 2D. Il s'agit de la même formule que pour obtenir un point autour d'un cercle, seulement adapté pour faire tourner n'importe quel point

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



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.

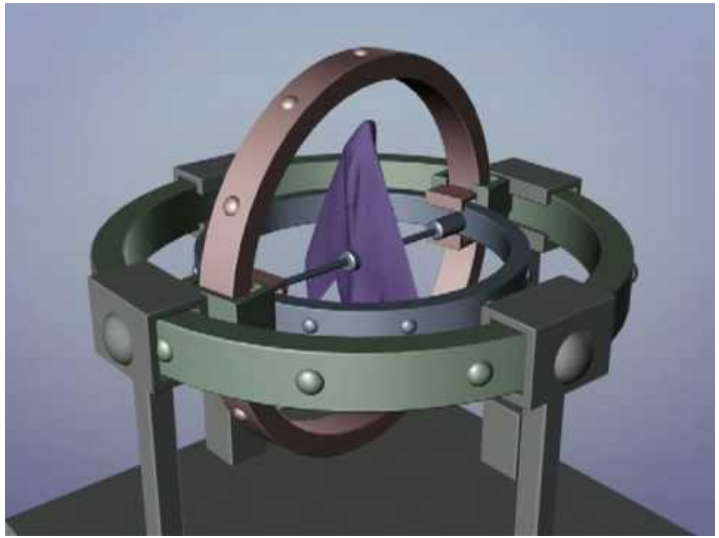
$$i^2 = j^2 = k^2 = ijk = -1$$

https://eater.net/quaternions

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

Pourquoi s'embêter avec **des Quaternions**?

Les angles d'Euler (les angles *classiques* affiché dans l'éditeur) possède un soucis 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:

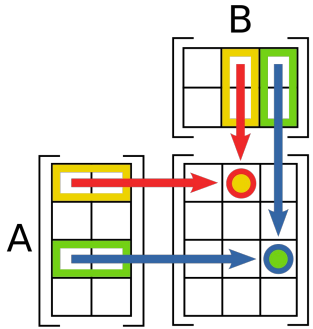
- Produit matriciel
- Transposition
- Inversion

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 x 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*

$$\begin{matrix} & \vec{b_1} & \vec{b_2} \\ & \downarrow & \downarrow \\ \vec{a_1} \rightarrow & \begin{bmatrix} 1 & 7 \end{bmatrix} & \cdot \\ \vec{a_2} \rightarrow & \begin{bmatrix} 2 & 4 \end{bmatrix} & \end{matrix} \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} \end{bmatrix}$$

$$A \qquad B \qquad C$$

Produit matriciel : pratique

Sur papier, calculez la matrice *C* résultat de l'opération *A x B*

$$\begin{matrix} & & \vec{b_1} & \vec{b_2} \\ & & \downarrow & \downarrow \\ \vec{a_1} \rightarrow \\ \vec{a_2} \rightarrow \end{matrix} \begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \vec{a_1} \cdot \vec{b_1} & \vec{a_1} \cdot \vec{b_2} \\ \vec{a_2} \cdot \vec{b_1} & \vec{a_2} \cdot \vec{b_2} \end{bmatrix}$$

$A \qquad B \qquad C$

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

Attention l'opération de multiplication n'est PAS commutatif : *A x B ≠ B x 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?

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

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 2x2 pour ca.

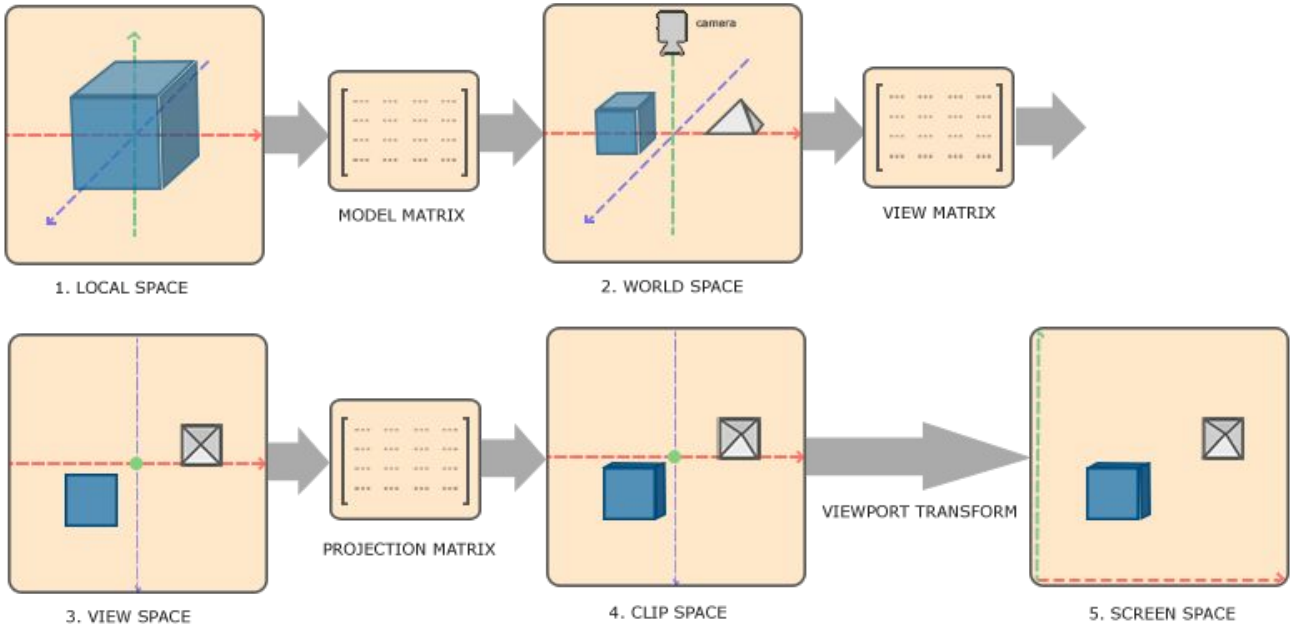
La solution: ajouter une composante et passer sur une matrices 3x3

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

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

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:

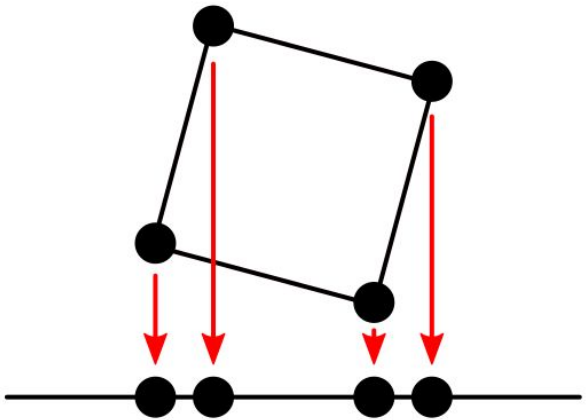


Projection orthographique

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

On obtient une image aplati sans profondeur ni perspective

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

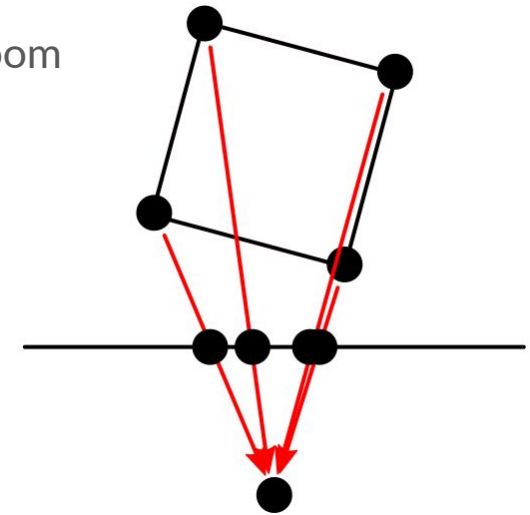


Projection perspective

En projection perspective on ignore prends en compte la composante Z lors de la projection pour rétrécir les objets à l'arrière et agrandir ceux a l'avant

On peut manipuler ainsi la longueur focale et faire des effets de zoom

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$



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 va nous faciliter nos transformation: `Matrix::CreatePerspectiveFieldOfView`, `Matrix::CreateWorld`, `Matrix::CreateLookAt`, etc.

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 Z pour la perspective.

Vertex Shader

Shaders - késako?

Un Shader est un programme définit 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, …

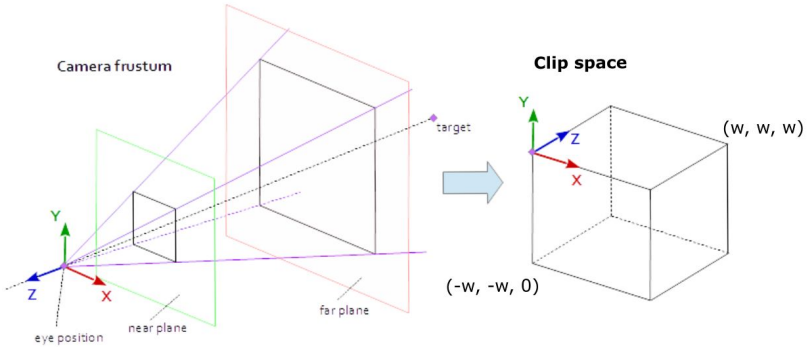
Vertex Shader - Utilité

Le Vertex Shader est celui qui va appliquer une transformation à chaque vertex en préparation de la rastérisation.

Il prends en entrée chaque vertices de nos primitives fraîchement assemblé et renvoie en sortie toutes les données nécessaire à la suite des opérations.

La rasterisation attends en sortie du vertex shader une position en Clip Space.

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).



Constant Buffer

Comment allons nous passer nos matrices à notre shader? Avec un autre type de buffer!
Il faudra juste fournir à sa déclaration `D3D11_BIND_CONSTANT_BUFFER`.

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

```
struct MatrixData {
    Matrix mModel;
    // ...
};
```

Les Constant Buffers vont nous permettre de définir des datas qui seront constante pour chaque invocation du vertex shader (aka chaque vertex va recevoir le même buffer).

- `VSSetConstantBuffers(`
  `(UINT) slot, (UINT) nombreDeBuffers,`
  `(ID3D11Buffer**) buffersArray)`

Constant Buffer

Pour modifier le contenu de votre Constant Buffer à chaque frame vous pouvez appeler

```
ID3D11DeviceContext::UpdateSubresource(
    (ID3D11Resource*) ressourceDeDestination,
    (UINT) sousRessource,
    (D3D11_BOX*) emplacementDestination, // optionnel, nullptr sinon
    (void*) data,
    (UINT) rowPitch, // taille de ligne, utile pour res. 2D/3D uniquement
    (UINT) depthPitch); // taille de ligne*colonne, res. 3D uniquement
```

HLSL

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 un Input et renvoie un Output, ces struct sont intégralement personnalisable.

Nous avons pour l'instant comme input:

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

Et comme output:

```
struct Output {
    float4 pos : SV_POSITION;
};
```

HLSL

Pour passer en paramètre un constant buffer on rajoute dans notre HLSL un `cbuffer`:

```
cbuffer MatrixData: register(b0) { // b0 représente le slot 0
    float4x4 Model;
    float4x4 View;
    float4x4 Projection;
};
```

Pour multiplier un vecteur par une matrice on utilise la fonction `mul(vector, matrix)`

TP 4

Matrice de transformation