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

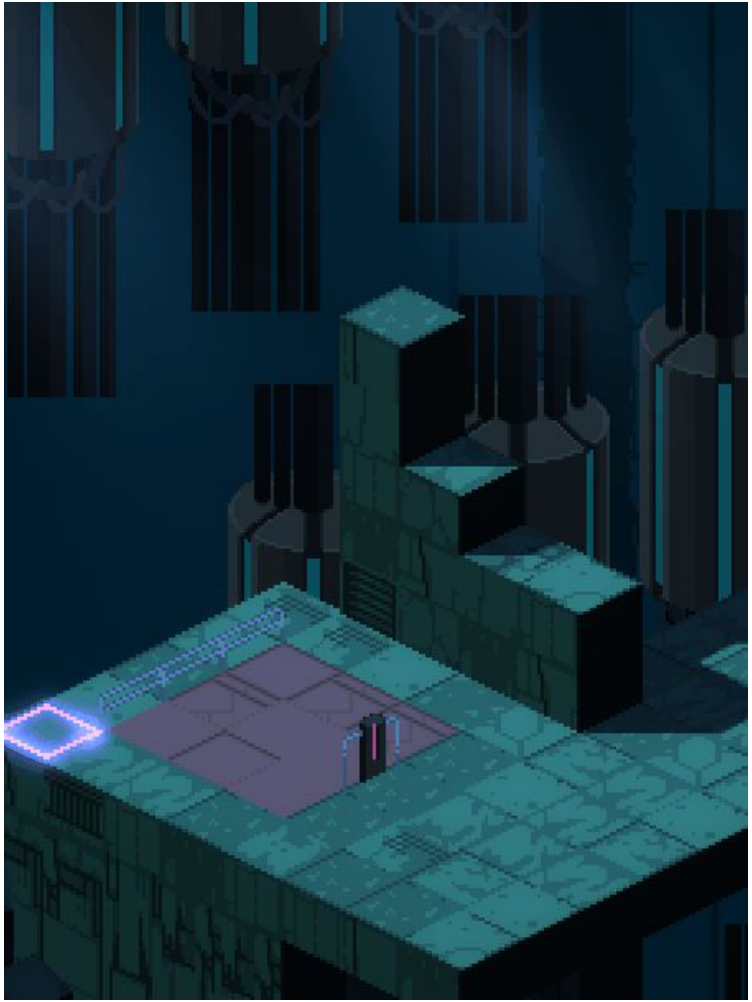

WINDJAMMERS 2
™

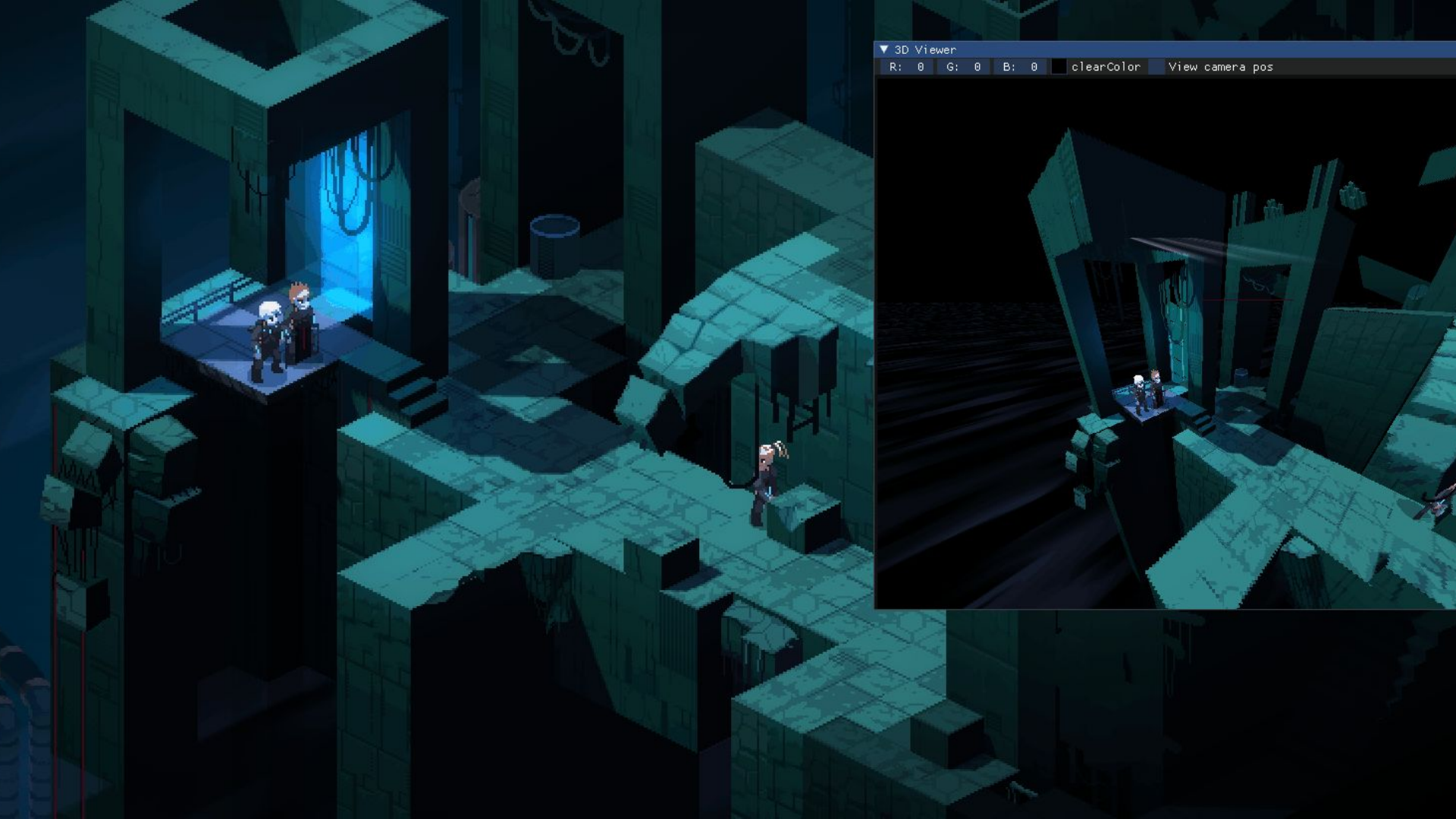

SHOTGUN KING
THE FINAL CHECKMATE


WARHAMMER
40,000
MECHANICUS

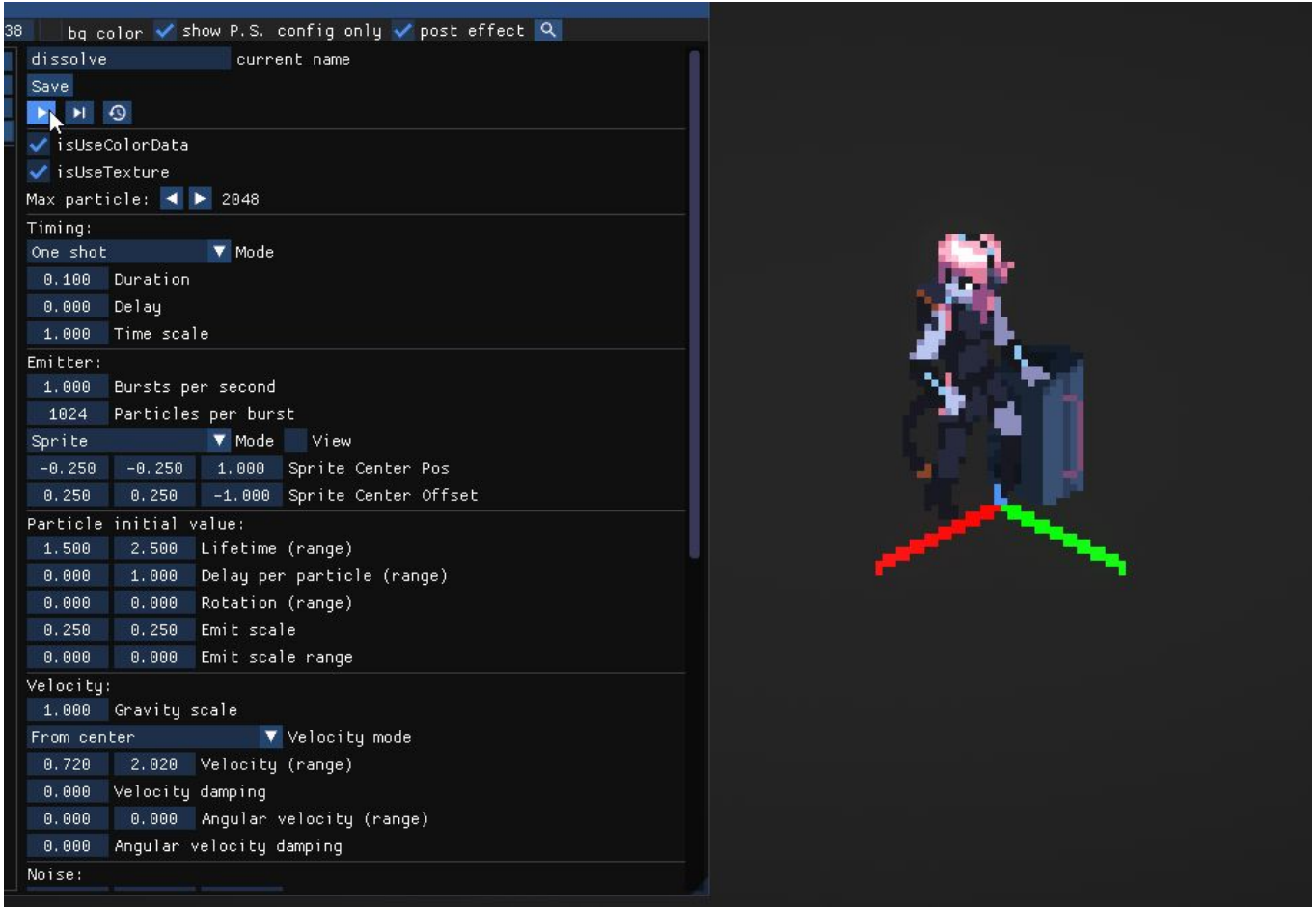

BIG
HELMET
HEROES


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

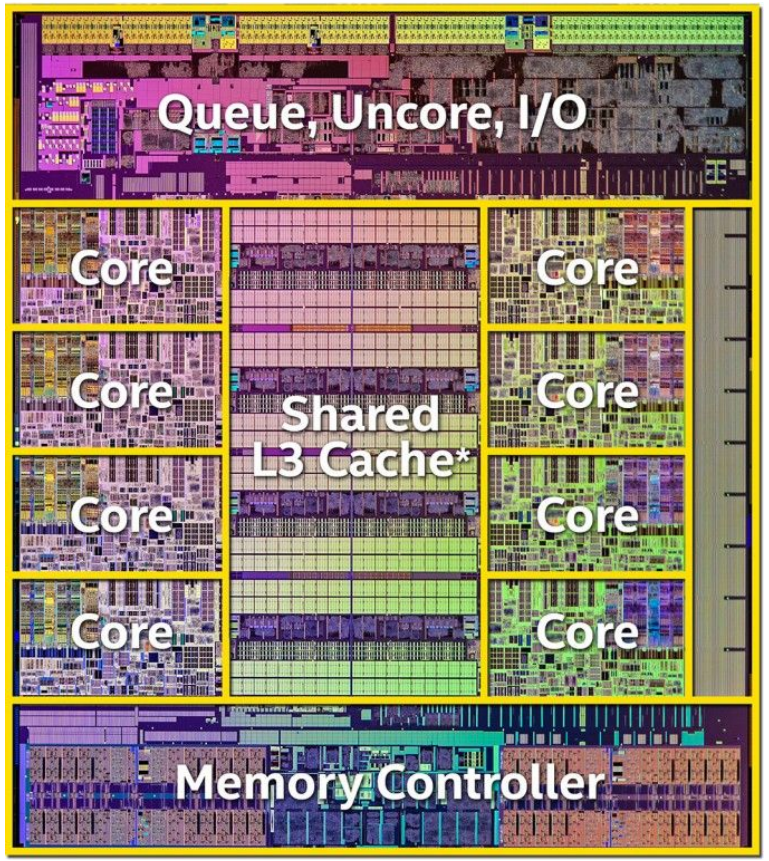
Vendredi
- Matin: Suivi TP et physique
- Après-midi: Suivi TP et culling

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)

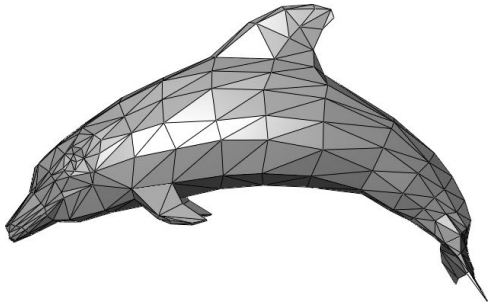
Intéressons nous au **maillage**

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:

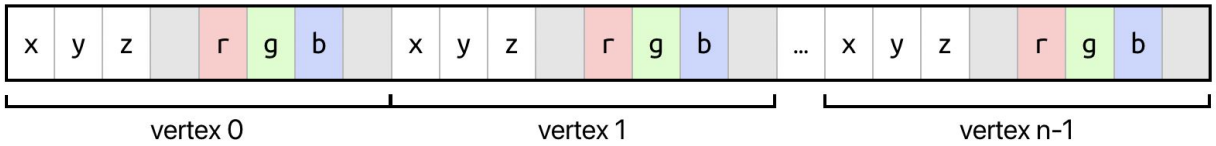
- Une position
- Une ou plusieurs coordonnées de textures
- Une couleur
- Une normal/tangente/binormal
- Des poids et des indices pour les os de l’armature
- À-peu-près tout ce que vous voulez vraiment



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

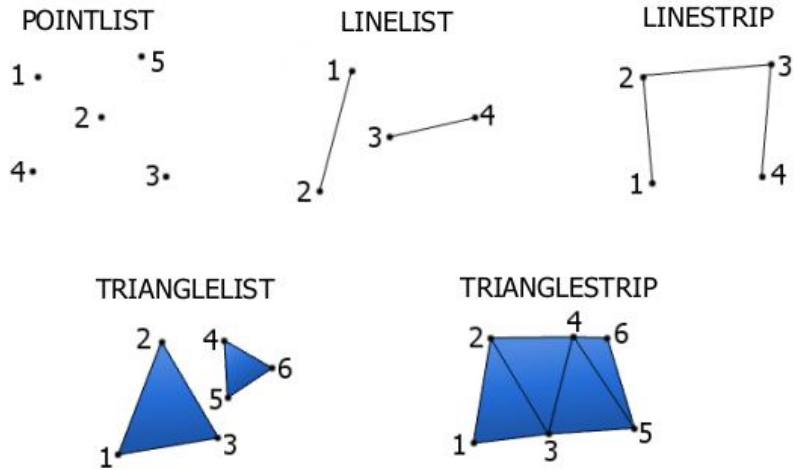
Modification des primitives soit en les subdivisant (tessellation) soit en les transformant complètement (geometry shader).

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() };
int strides[] = { sizeof(float) * 3 };
int 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