# MV54 - Moteur de Rendu

Julien Barbier

29 mars 2023

# Prérequis

# Pré-requis

Le prérequis nécessaire pour le cours :

- Définition et caractéristiques d'un maillage 3D utilisé pour la réalisation de rendu informatique.
- Coordonnées homogènes

# Maillage 3D

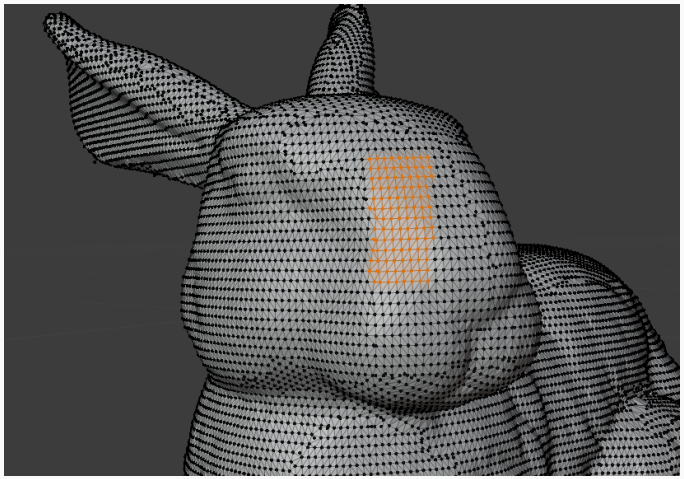


**Figure 1 :** Maillage 3D avec les vertices surlignées (Le lapin de Stanford)

- Objet informatique composé de vertices et de faces qui représentent des formes 3D.
- Un vertice est un des sommets du maillage 3D et possède différentes caractéristiques (Coordonnées de Texture, Normales, Tangente, bitangente (ou binormale)...).
- Une face est composée d'au moins 3 vertices et va permettre de remplir et texturer le maillage 3D.

# Coordonnées homogènes i

## Définition

- Dans un espace à $N$ dimensions, une coordonnée homogène est définie par un vecteur de taille $N + 1$. Par exemple si $A = (1; 1; 0)$ dans un monde euclidien à 3 dimensions, alors $A_{homogeneous} = (1; 1; 0; 1)$
- La composante ajoutée à une coordonnée homogène correspond à un facteur d'échelle des coordonnées. Ainsi on peut définir n'importe quelle coordonnée de l'espace sous la forme : $A = \begin{pmatrix} \alpha \\ \beta \\ \gamma \\ \lambda \end{pmatrix}$

# Coordonnées homogènes ii

## Définition

- Les coordonnées homogènes permettent de simplifier les transformations de translation, de mise à l'échelle et de rotation en utilisant plus qu'une matrice. Cette matrice est une matrice carrée de taille $N + 1$ ayant la forme :

$$\begin{pmatrix} R_{3,3} & T_{3,1} \\ 0_{1,3} & 1 \end{pmatrix}$$

  avec $R_{3,3}$ une matrice carrée de taille 3, $T_{3,1}$ un vecteur colonne de taille 3 et $0_{1,3}$ le vecteur nul en ligne.

- Appliquer une transformation à une coordonnée homogène par une matrice homogène se fait de la manière suivante :

$$A_{transform} = M * A_{original}$$

# Introduction au Moteur de Rendu

# Moteur de Rendu i

## Définition

- Permet de faire le rendu des différents maillage d'une scène sur dispositif d'affichage.
- S'occupe des calculs de lumières, d'ombres et des différents post effect.
- Correspond à la partie Vue dans l'architecture MVC.
- Peut-être temps réel ou non.


Rendu de Sponza avec Godot 4.0

# Moteur de Rendu ii


Rendu réalisé avec Cycles (`https://docs.blender.org/manual/fr/dev/render/cycles/introduction.html`)

## Les différentes API utilisés dans le rendu

- Temps réel :
  - OpenGL (Linux, Windows)
  - D3D (Windows)
  - Vulkan (Linux, Windows, MacOS (via MoltenVK))
  - Metal (MacOS seulement)
- Non-temps réel :
  - RenderMan (Moteur de rendu de Pixar)
  - Cycles (Moteur de rendu de Blender (OpenSource))
  - MoonRay (Moteur de rendu de Dreamworks (OpenSource))

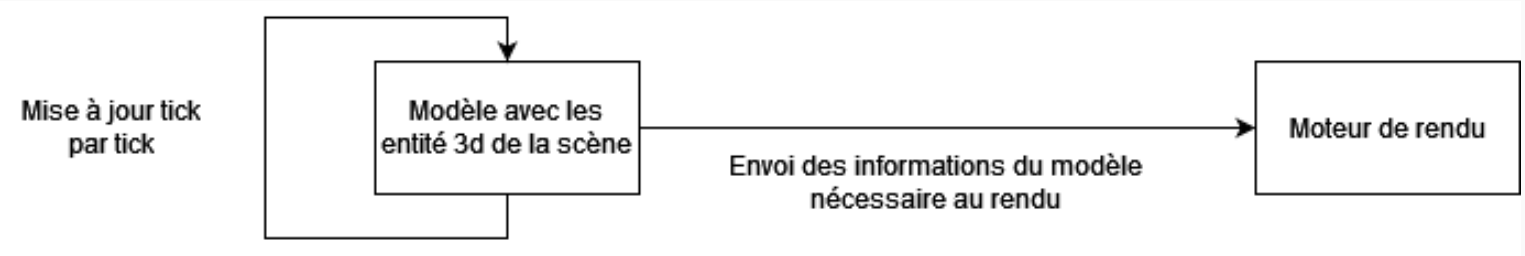



**Figure 2 :** Principe de l'utilisation d'un moteur de rendu

# Comment ça marche

```
define RenderTargets to render
define Rasterizer state //On verra ca par la suite
define OM state //On verra ca par la suite
for mesh in meshes
    define parameters to render the mesh
    for light in Lights
        define light parameter
        Render(mesh, light)
    end for
end for
```

Algorithme simplifié de rendu le plus simple (forward rendering).

# Comment ça marche

Pour comprendre le fonctionnement d'un moteur de rendu, il faut donc comprendre :

- le fonctionnement d'une passe de rendu
- le rendu des maillages avec des lumières
- le flux de passes de rendu nécessaire pour la réalisation d'une scène 3D complexe.

# Concept de base du rendu

# Introduction

Différents concepts existent dans le rendu pour réaliser une passe de rendue :

- Les buffers qui permettent de stocker de l'information qui sera envoyée au GPU
- Les textures qui permettent d'habiller les maillages et qui servent de support au rendu
  - La texture de rendue final est appelé le backbuffer et est fourni par le constructeur du casque ou par l'API de la fenêtre.
- La présence de pipelines de rendu niveau hardware dans les GPU

# Pipeline de Rendu i

**Définition**

- Permet de faire le passage de maillage enregistré dans la VRAM à la visualisation de ce maillage sur une texture.
- Est souvent réalisé côté GPU qui est doté d'une ou plusieurs pipelines de rendu niveau hardware (driver GPU).
- Possède plusieurs passes programmables via l'utilisation de Shaders.

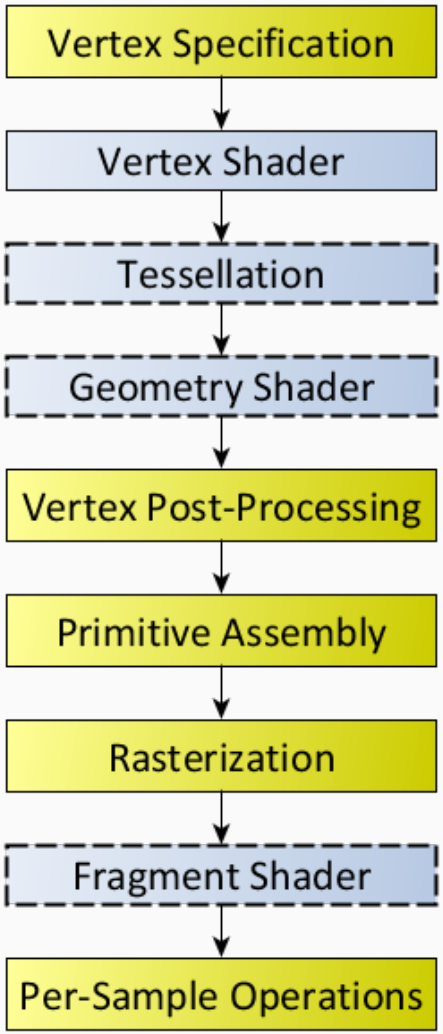



**Figure 3 :** Pipeline de Rendu sous OpenGL (venant du wiki Khronos OpenGL)

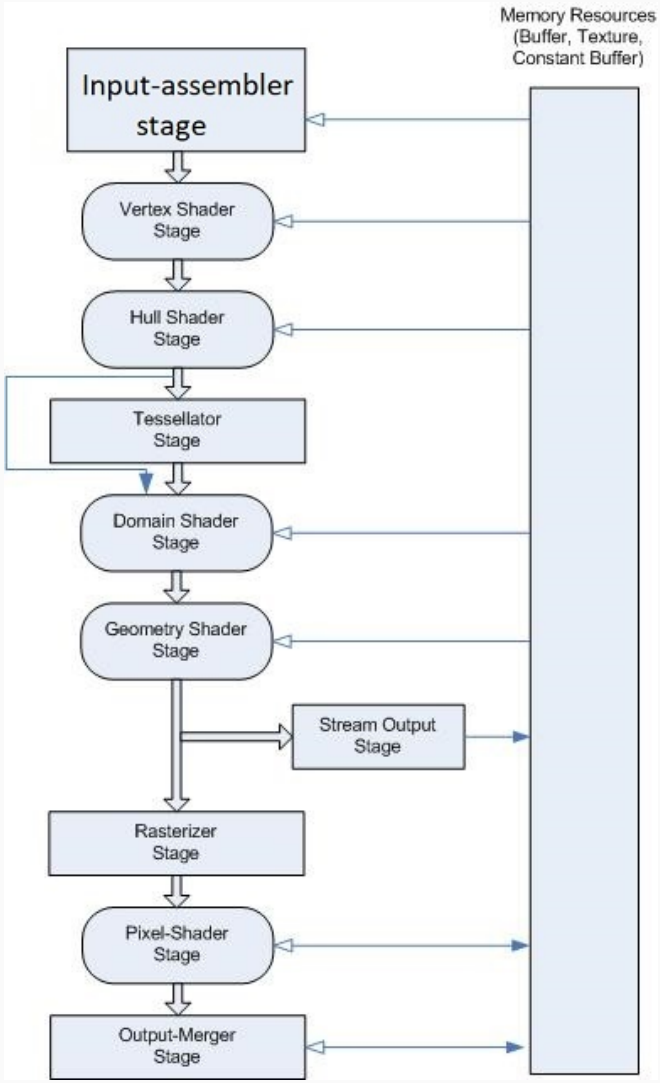



**Figure 4 :** Pipeline de rendu sous D3D11 (venant de la documentation Microsoft)

## Shader : Kesaco

- Un programme GPU utilisé pour modifier les primitives de rendu (vertices, pixels...).
- Compilé en runtime (peut être précompilé pour accélérer le processus)
- Plusieurs types de shader disponible :
  - Vertex Shader
  - Geometry Shader
  - Fragment Shader (Pixel Shader en D3D)

# Vertex Shader

**Définition**

- Fait la transformation d'un vertices de l'object space vers le clip-space.
- Permet la manipulation des vertices
- S'applique à chaque vertex une par une ($O(n)$)

# Exemple de Vertex Shader

```
1  struct VS_INPUT
2  {
3    float4 vPosition : POSITION;
4    float3 vNormal : NORMAL;
5    float4 vBlendWeights : BLENDWEIGHT;
6  };
7
8  struct VS_OUTPUT
9  {
10   float4  vPosition : SV_Position;
11   float4  vDiffuse : COLOR;
12
13 };
14
15 float4x4 mMVP;
16
17 VS_OUTPUT VS_Transform(VS_INPUT sIn)
18 {
19   VS_OUTPUT sOut;
20   sOut.vPosition = mul(mMVP, sIn.vPosition);
21   sOut.vDiffuse = float4(1.0, 0.0, 0.0);
22   return sOut;
23 }
24
```

Exemple de Vertex shader réalisant la transformation d'un vertice dans le clip-space

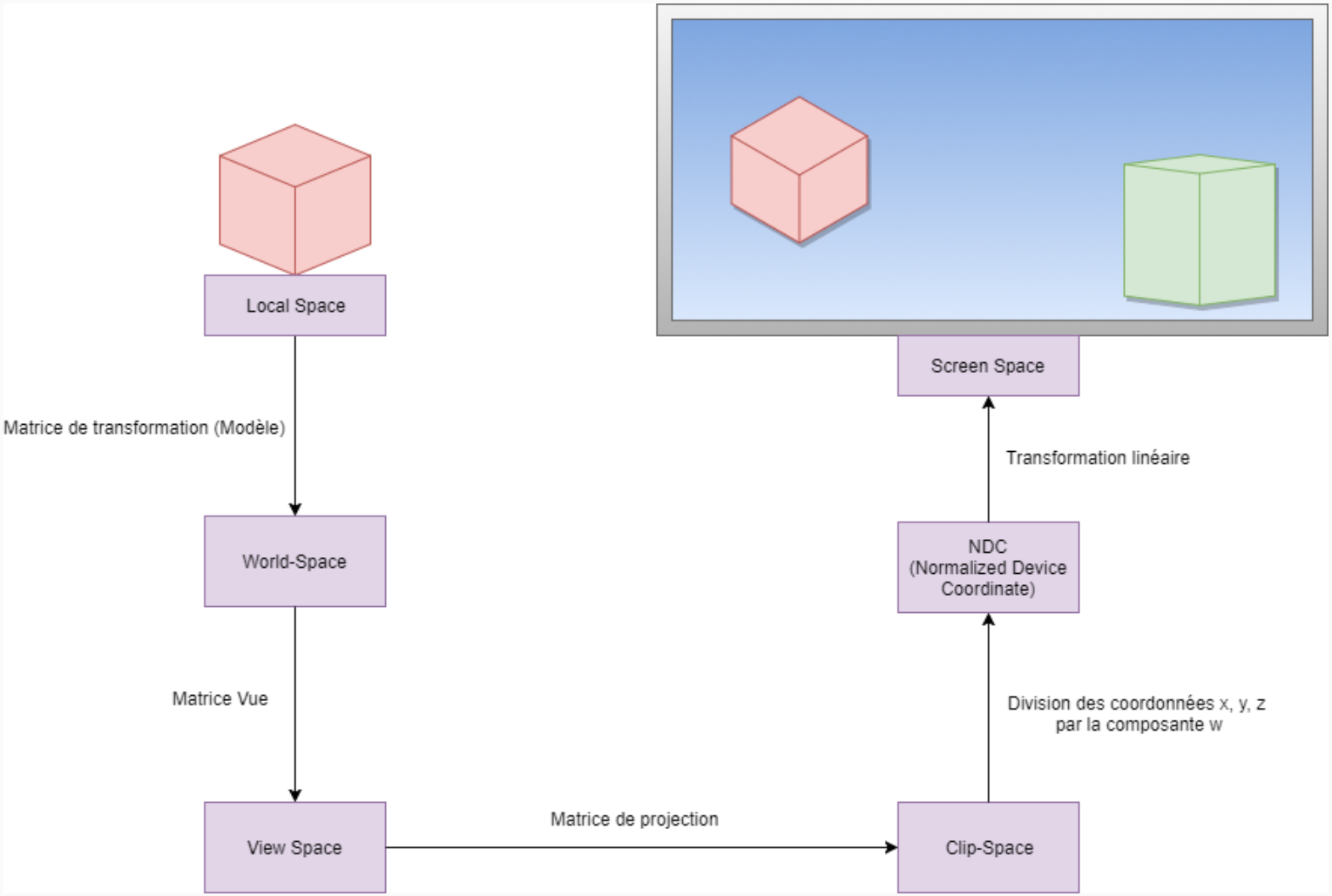



**Figure 5 :** Les différentes transformations appliquées à un maillage durant le rendu

# NDC

**Définition**

- Signifie "Normalized Device Coordinate"
- Peut différer entre les différentes API.
- Permet de définir une limite dans l'espace des objets à rendre à l'écran.

**Exemple**

L'espace de définition du NDC d'OpenGL va de [-1 ;-1 ;-1] à [1 ;1 ;1].

L'espace de définition du NDC de D3D11 (Direct 3D 11) va de [-1 ;-1 ;0] à [1 ;1 ;1].

# Geometry Shader

**Définition**

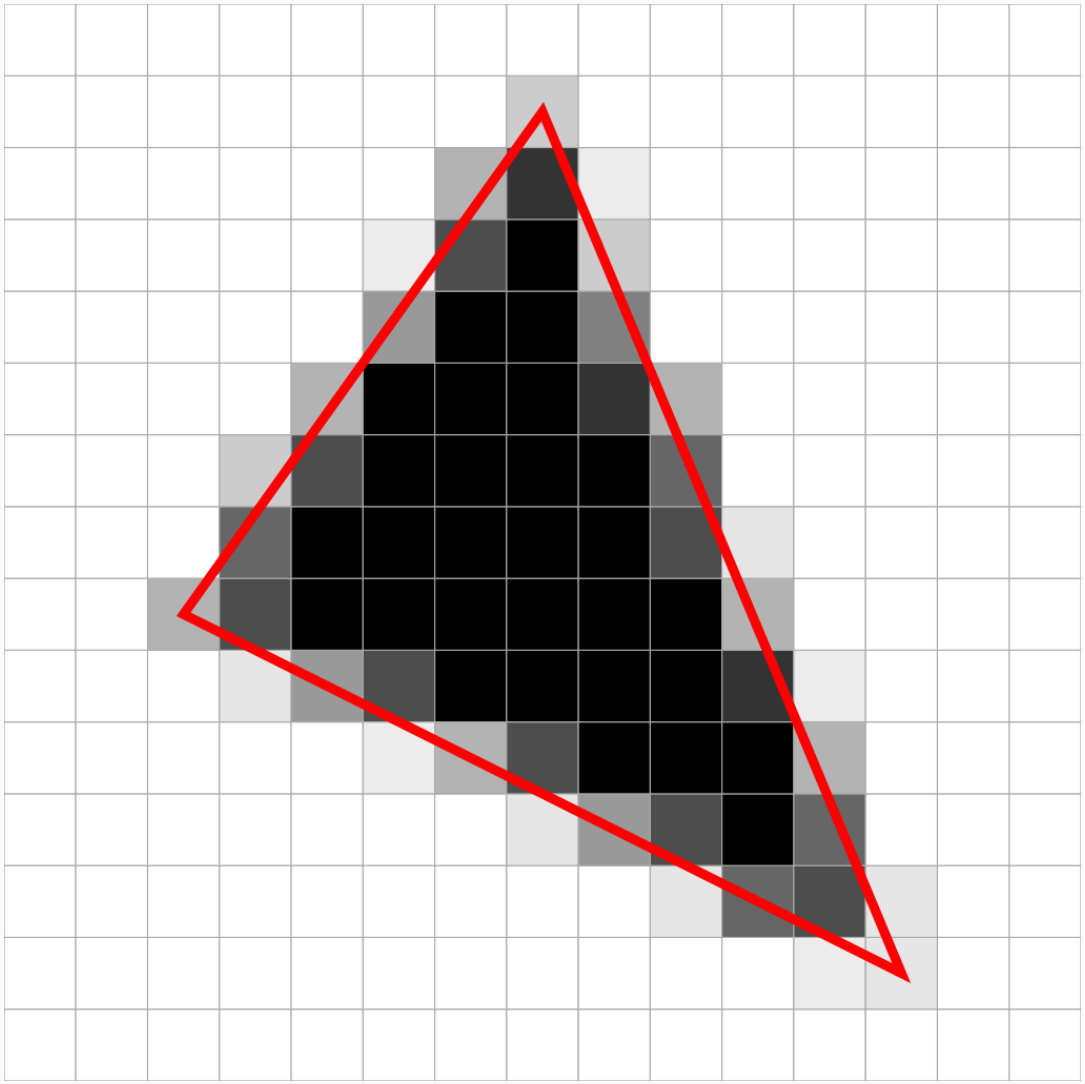
- Permet d'appliquer des transformations primitives par primitives (edges, faces...)
- Permettait de faire du rendu sur plusieurs render targets en une passe avec différentes positions de vertices (plus nécessaire depuis D3D11 extension 3 et avec une extension OpenGL à partir de la version 4.1).
- Le mieux : ne pas l'utiliser car très gourmand en temps de calcul

# Rasterisation

## Définition

- Étape qui permet de convertir les primitives en "pixel" (en fragment sous OpenGL).
- Va linéariser les paramètres des vertices à l'intérieur d'une face (interpolation linéaire)
- Va réaliser la conversion du clip space vers le Viewport space.
- Peut réaliser une étape d'early Z rejection (Rejet du pixel car présence d'un pixel déjà traité avec une comparaison de profondeur négative)
- Va clip les primitives qui sont en dehors de l'espace de définitions

# Rasterisation



Exemple de rasterisation d'un triangle avec antialiasing (2D ici mais la 3D fonctionne de la même manière sans anti-aliasing) (`https://commons.wikimedia.org/wiki/File:Rasterisation-triangle_example.svg`)

# Pixel Shader

**Définition**

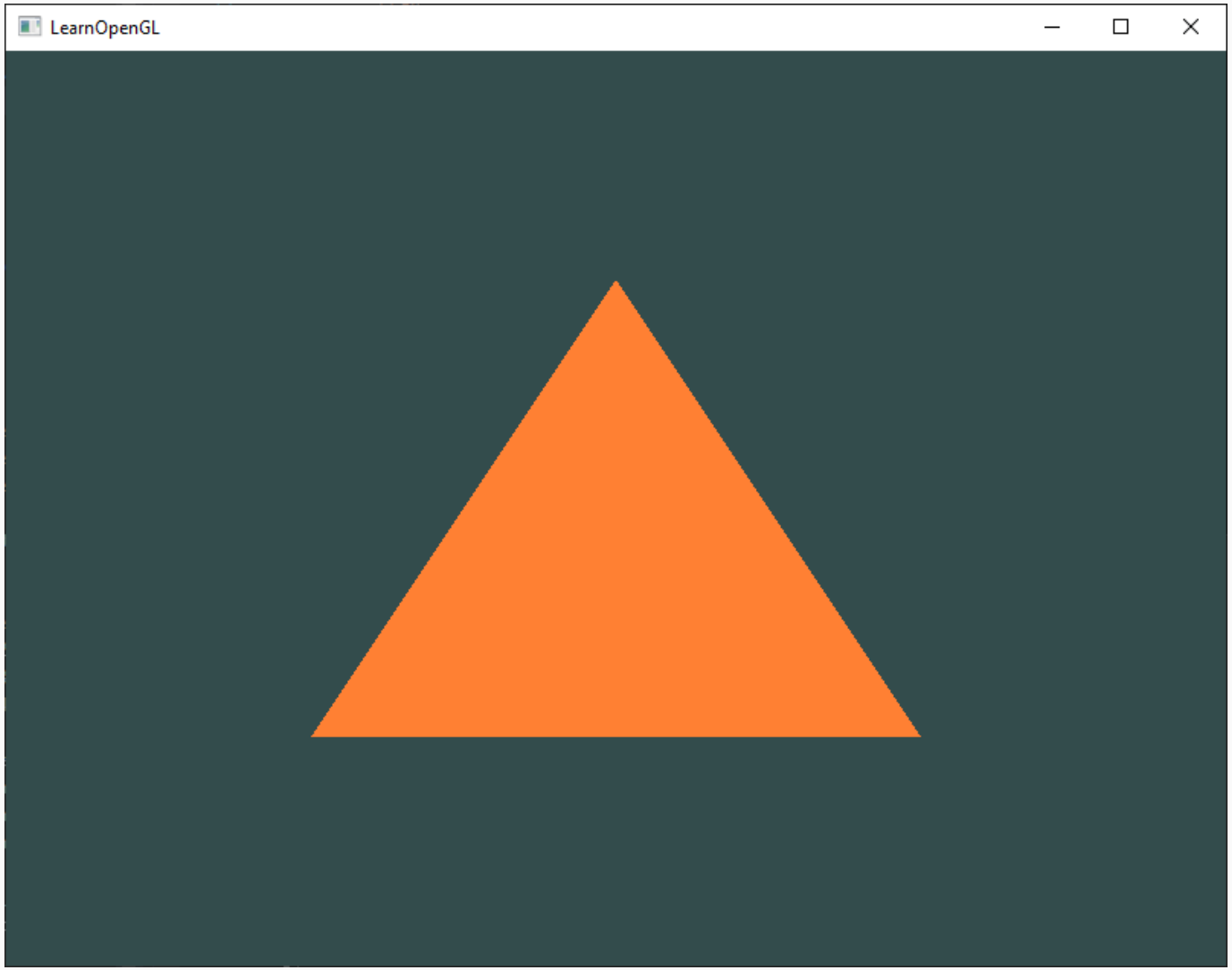
- Permet de définir la couleur d'un pixel
- S'applique à chaque pixel
- Dernière étape programmable de la pipeline
- Utilisée dans l'application des lumières, des ombres, du packing d'informations des maillages...
- Souvent la passe qu'il faut optimiser

# Exemple de Vertex Shader

```
1  struct FS_Input
2  {
3    float4  vDiffuse : COLOR;
4  };
5
6  struct FS_Output
7  {
8    float4 vColor : SV_Target0;
9  };
10
11 FS_Output main(FS_Input sIn)
12 {
13   FS_Output sOut;
14   sOut.vColor = sIn.vDiffuse;
15   return sOut;
16 }
17
```

Exemple de Fragment shader rendant la diffuse envoyé en paramètre.

# Exemple de résultat du pixel shader




Triangle rendu avec OpenGL en utilisant un pixel shader
(`https://learnopengl.com/Getting-started/Hello-Triangle`)

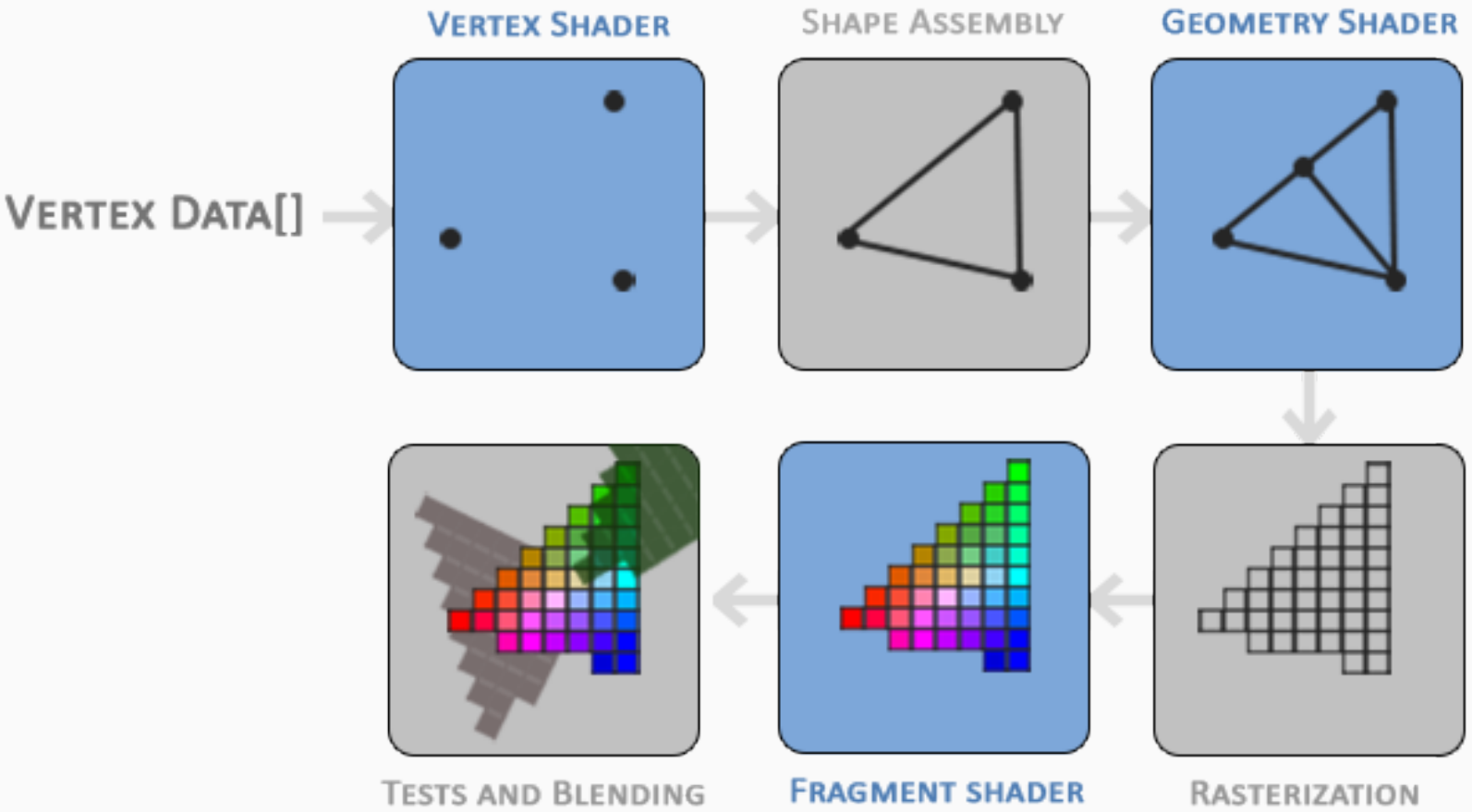
# Après le Pixel Shader

Différentes opérations sont réalisées après le pixel shader

- L'alpha-blending si activé : permet de faire un blend entre la couleur du pixel de la texture et la couleur de sortie du Pixel Shader.
- Depth test : Permet de vérifier que le pixel rendu n'est pas derrière le pixel déjà présent.
- Stencil Test : Permet de faire un test avec le Stencil (différents opérateurs disponibles)

Toutes les opérations sont réglables.

# Pour résumer




Présentation illustrée des différentes passes de rendu de la pipeline rasterizer (`https://learnopengl.com/Getting-started/Hello-Triangle`)

# Autres passes

- Tesselation (domain and hull shader in D3D) : Permet de subdiviser le maillage pour ajouter des détails au maillage (très utilisés pour les terrains)
- Mesh shader (D3D12 ultimate seulement) : Permet de réduire le nombre de vertices avant la passe de vertex shader.
- Compute Shader : Permet de faire des opérations côté GPU sans à devoir passer par tout la pipeline de rendu.

**Problématique**

- Comment envoyer les informations de vertices à la pipeline ?
- Comment envoyer des informations complémentaires permettant la réalisation de la passe ?
- Comment utiliser les informations transmises ?

# Buffers de rendu

## Définition

- Permet de stocker côté GPU des structures de données définies à l'avance.
- Existe en différents types de buffers :
  - Vertex Buffer (VBO en OpenGL) : Buffer qui définit les vertices
  - Index Buffer (VAO en OpenGL) : Buffer qui définit les indices des vertices.
  - Structured Buffer (ou Shader Storage Buffer Object en OpenGL) : Buffer qui permet de stocker des tableaux de structures de données pour le rendu (comme des matériaux...)
  - Constant Buffer (ou Uniform Buffer Object) : Buffer qui possède une liste de données. Très utile pour faire des économies d'envoi d'informations.

# Vertex buffer

**Définition d'un vertices dans un moteur de rendu**
Possède différentes propriétés :

- Coordonnée du vertice dans l'object space
- Coordonnée de texture
- le vecteur normal du vertice
- le vecteur tangent du vertice
- le vecteur bitangent (ou binormale) du vertice

```
1 struct Vertex
2 {
3   float3 vPosition : POSITION0,
4   float2 vTexCoord : TEXCOORD,
5   float3 vNormal : NORMAL,
6   float3 vTangeant : TANGEANT,
7   float3 vBiNormal : BINORMAL
8 };
9
```

Structure d'un vertice dans un langage GPU (HLSL)

**Problèmes**

- Taille d'un vertice : 3 * 4 octets = 12 octets minimum (plus si besoin).
- Pour des faces triangulaires, il est nécessaire de dupliquer voire de tripler les vertices. Demande pas mal de VRAM (Video RAM).

**Solutions**

Utilisation d'un Index Buffer qui permet de définir les indices des vertices pour constituer les faces du maillage

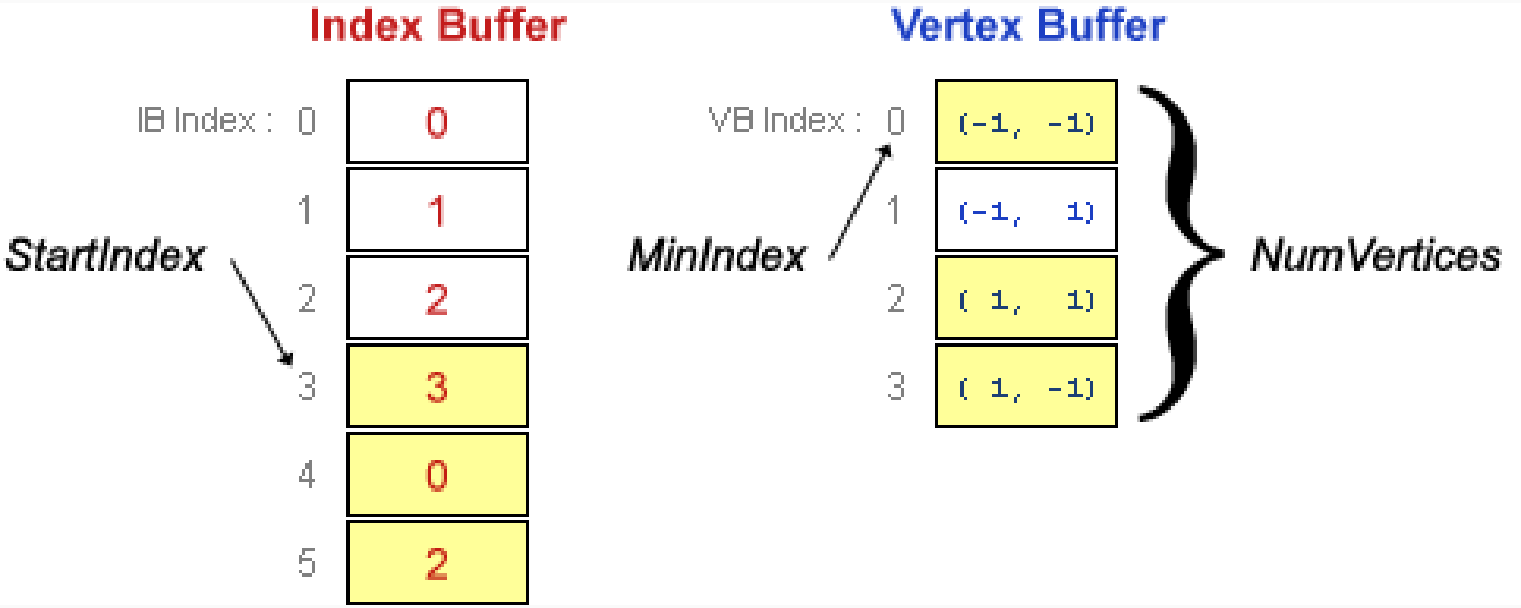



**Figure 6 :** Interaction entre le vertex buffer et l'index buffer (vient de https://docs.microsoft.com/en-us/windows/win32/direct3d9/rendering-from-vertex-and-index-buffers)

# Constant Buffer

**Définition**

- Contient une liste de vecteurs ou de valeurs qui sont constants dans la passe de rendu.
- Sert à envoyer un ensemble d'informations nécessaires au rendu avec un seul "bind"
- Peut être réutilisé sur plusieurs passes de rendu

```
1 cbuffer CameraBuffer
2 {
3   float4x4 mView,
4   float4x4 mProjection,
5   float4x4 mViewProj,
6   float3 vPosition
7 };
8
```

Définition d'un constant buffer (HLSL)

# Structured Buffer

**Définition**

- Permet de réaliser un tableau de structure de donnée d'informations dans le shader.
- Permet d'avoir différentes configurations de la structure de donnée.

```
1 struct Material
2 {
3   float4 vColorModulator,
4   float fMetallic,
5   float fRoughness
6 };
7 StructuredBuffer<Material> materials;
8
```

Définition d'un structured buffer (HLSL)

# Textures en rendu

## Définition

- Correspond à un buffer contenant des valeurs représentants des couleurs.
- Existe différents types de format de texture suivant un nommage standard : *CanauxNbBitsAutre(s)Canal(ux)NbBits* (RGBA8, RGB10A2, RGBA16f, R16...)
- Existe différents types de textures (1D, 2D, Texture Cube, 3D)
- Peut servir pour l'habillage de maillage (texture immuable) ou être utilisé pour le rendu. Dans le dernier cas, elles sont représentées dans des Render Targets (sous D3D) ou dans des Framebuffers (sous OpenGL).

# Texture de rendu

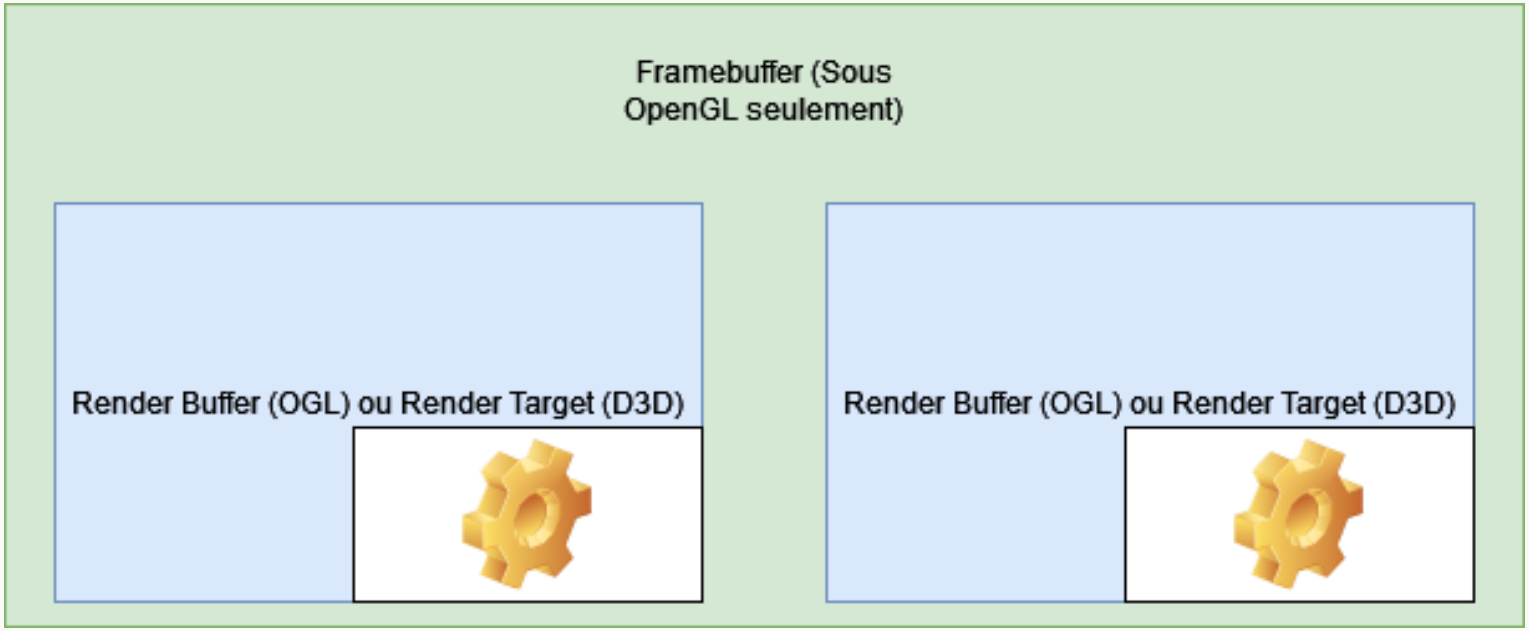



**Figure 7 :** Représentation des textures de rendus sous OpenGL (avec équivalent D3D11)

# Texture de profondeur

## Définition

- Permet de stocker les informations de profondeurs de la scène rendu (information de profondeur = Z dans le nom de la texture)
- Permet d'éviter d'écraser des parties d'une texture lors du rendu si le maillage est derrière un autre.
- Peut posséder une composante utilisée en fonction des besoins : le Stencil (noté S dans le nom de la texture)
- Possède différents formats : Z16, Z24S8, Z32f, Z32S8X24.

# Exemple de texture de profondeur

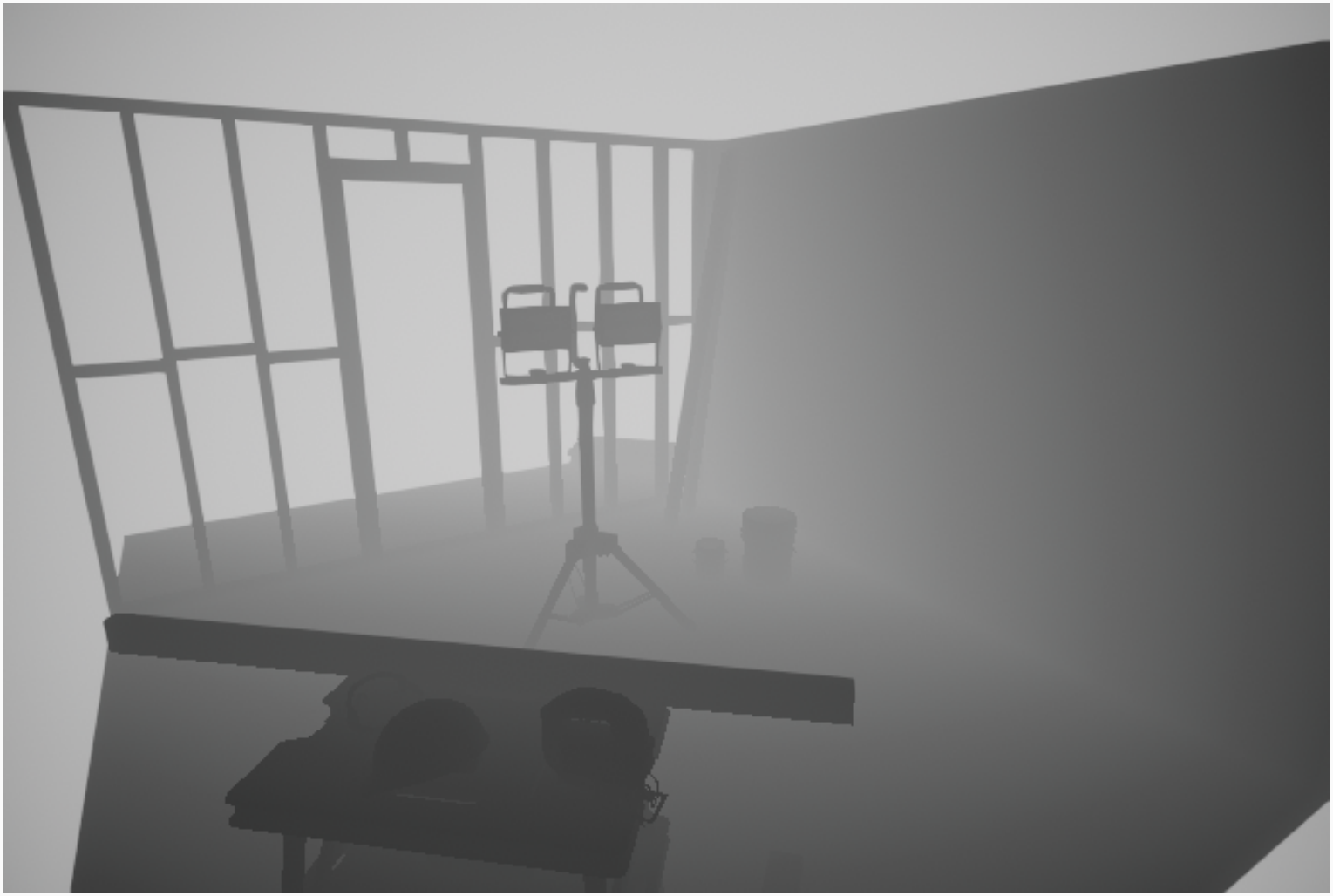


**Figure 8 :** Exemple de texture de profondeur

# Le Backbuffer

**Définition**

- Correspond à la texture finale du rendu
- Fournit par la swapchain fournie lors de l'initialisation de la fenêtre de l'application
- Possède différents formats supportés
- Une étape de flip/swap est nécessaire pour qu'elles soient envoyées à l'écran.

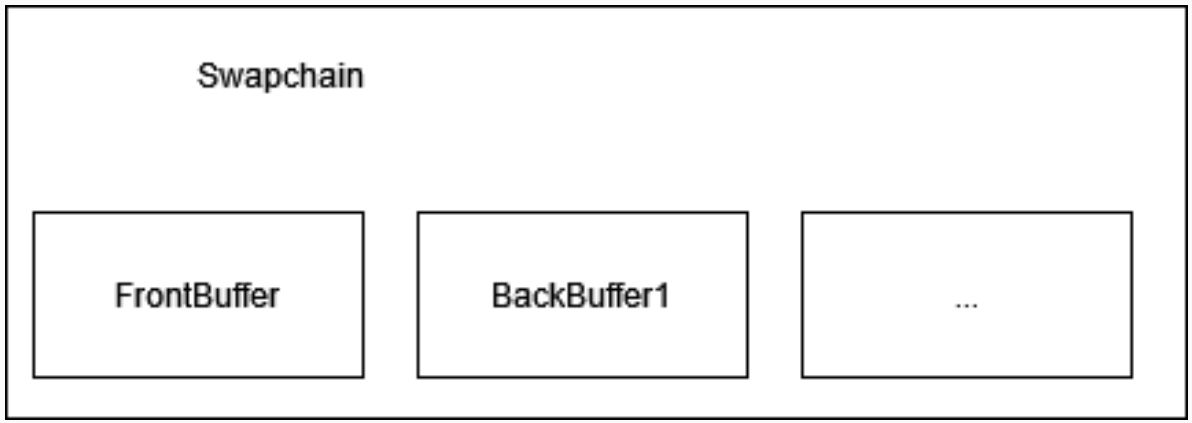



**Figure 9 :** Présentation simplifiée d'une swapchain

# Apport d'un moteur 3D

# Les objectifs d'un moteur 3D

- Rendre plusieurs entités à l'écran en un temps très restreint (60fps = 16ms)
- Avoir des effets de rendus convaincants voire réalistes
- Laisser le plus de marge possibles aux graphistes.

# Problèmes

- Plus il y a de maillages, plus la complexité de la scène grandit, plus le temps de calcul sera important.
- Il faut éviter d'avoir un trop grand nombre de passe de rendu (1000 drawcalls kill)
- Obtenir des effets de rendus réalistes demande des calculs imposants (ray-tracing, PBR...)
- Les graphistes ont tendances à faire des maillages assez imposants ou à dupliquer ces derniers.

# Optimisations du rendu de plusieurs maillages

# Optimisations du rendu de plusieurs Maillages

## Tri des maillages dans la scène

- Pas de solutions miracles, mais des solutions adaptées en fonction des besoins
- Rendu des maillages dits "opaque" peut être réalisé dans n'importe quelle ordre
  - Worst case catastrophique si on rend des maillages les plus loins au plus proches
  - Une solution est de les trier et de les rendre du plus proche au plus loin.
    - Tri en fonction de la distance avec la caméra -> Le tri peut vite devenir couteux.

**Rejet des maillages qui ne sont pas dans le frustum de la caméra**

- Mise en place du frustum culling côté CPU
  - On prend la bounding box ou la bounding sphere du maillage
  - On regarde si celle-ci intersecte avec le frustum de la caméra.
  - Si oui, on l'ajoute dans la file des maillages à rendre
  - Si non, on ne le rend pas.

**Et les maillages semi-transparents ?**

- Pas vraiment de solution d'optimisations (hors frustum culling)
- Depth Write désactivée, mais Depth Read activé
- Rendre du plus loin au plus proche (pour l'alpha blending) ou du plus proche au plus loin.
    - Il existe des solutions pour éviter ceci, mais il ne s'agit que d'approximations plus ou moins de bonne qualité

# Optimisation avec plusieurs entités qui possède le même maillage

## Problèmes

- Plusieurs entités 3D de la scène utilise le même maillage
- Plusieurs passes de rendu en utilisant les mêmes paramètres

## Solution

- L'instancing
  - Permet le rendu de plusieurs instances d'un même maillage en une passe de rendu
  - Nécessite l'utilisation de structured buffer pour stocker la matrice world de chaque instance
  - Génère un instance id dans la pipeline de rendu.

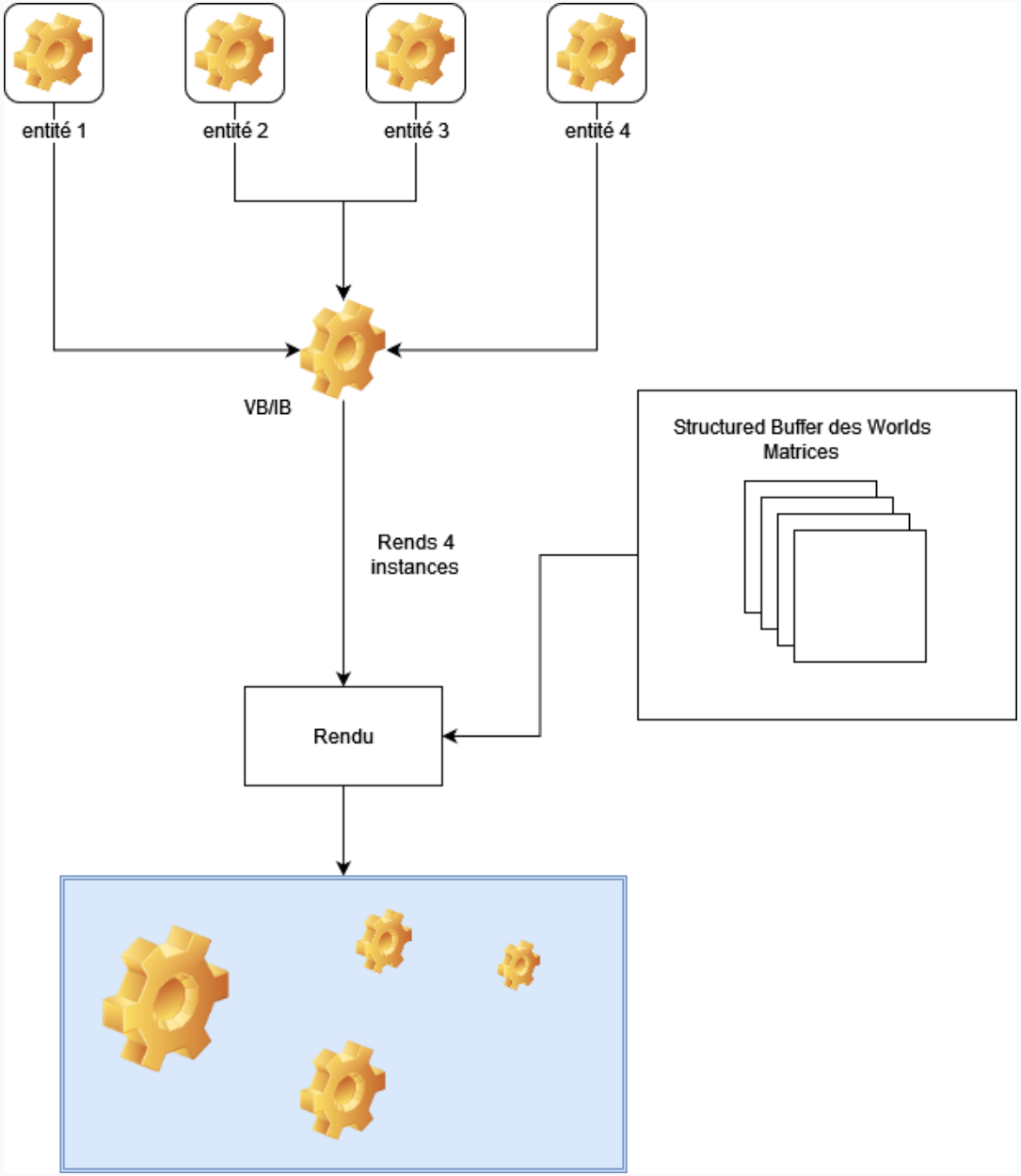



**Figure 10 :** Exemple de fonctionnement de l'instancing

# Différence API entre un rendu non instancié et un rendu instancié i

**En non instancing**

- Nécessité d'envoyer soit dans un constant buffer soit en uniform la matrice world
- shader plus simple
- Appeler la fonction de rendu sans le paramètre sur le nombre d'instance.

# Différence API entre un rendu non instancié et un rendu instancié ii

**En instancing**

- Nécessité de bind le structured buffer contenant les différentes matrices world dans la pipeline de rendu
- Nécessite d'adapter les shaders pour prendre en compte l'instance id
- Nécessité de gérer le nombre d'instances et les instances à rendre
- Appeler la fonction de rendu adéquate (ajout d'un paramètre pour le nombre d'instance)

# Les limites de l'instancing

- Non adapté dans certains cas (même maillage mais pas les mêmes textures utilisées)
- Non présent sur d'anciennes versions d'OpenGL et de D3D (arrivé en OpenGL 3.1 et en D3D11)

# Moteur 3D et Réalité Virtuelle / Réalité Augmentée

# Problématique avec la Réalité Virtuelle

- Texture de rendu très grande (certains casques font du 1440x1600px par œil (HTC Vive Pro))
- Rendu en double dans le cas de casque stéréoscopique
  - Les casques stéréoscopiques possèdent 2 matrices view projections pour les 2 yeux.

# Solutions

- Utilisation de rendu en VPRT qui permet de rendre en une passe de rendu sur les 2 textures à la fois
- Utilisation de l'instancing (quasi obligatoire)
- Rendu dans une résolution moindre puis Upscaling
- Réduction de la qualité du rendu (réduction du nombre de lumières, utilisation de maillage low-poly...)

# VPRT

## Définition

- Permet de rendre en une passe un maillage sur 2 textures à la fois avec une position de vertices différentes.
- Nécessite de doubler le nombre d'instances d'un maillages (si on utilise l'instancing).
- Envoi des 2 views proj dans le shader (soit en structured buffer soit en uniform de tableau de matrices).
- Utilisation d'un geometry shader pour choisir la bonne render target en fonction de l'instance id.
- Choix de la render target en fonction de l'instance id possible dans le vertex shader si disponible (D3D11 option 3 et extension OpenGL)

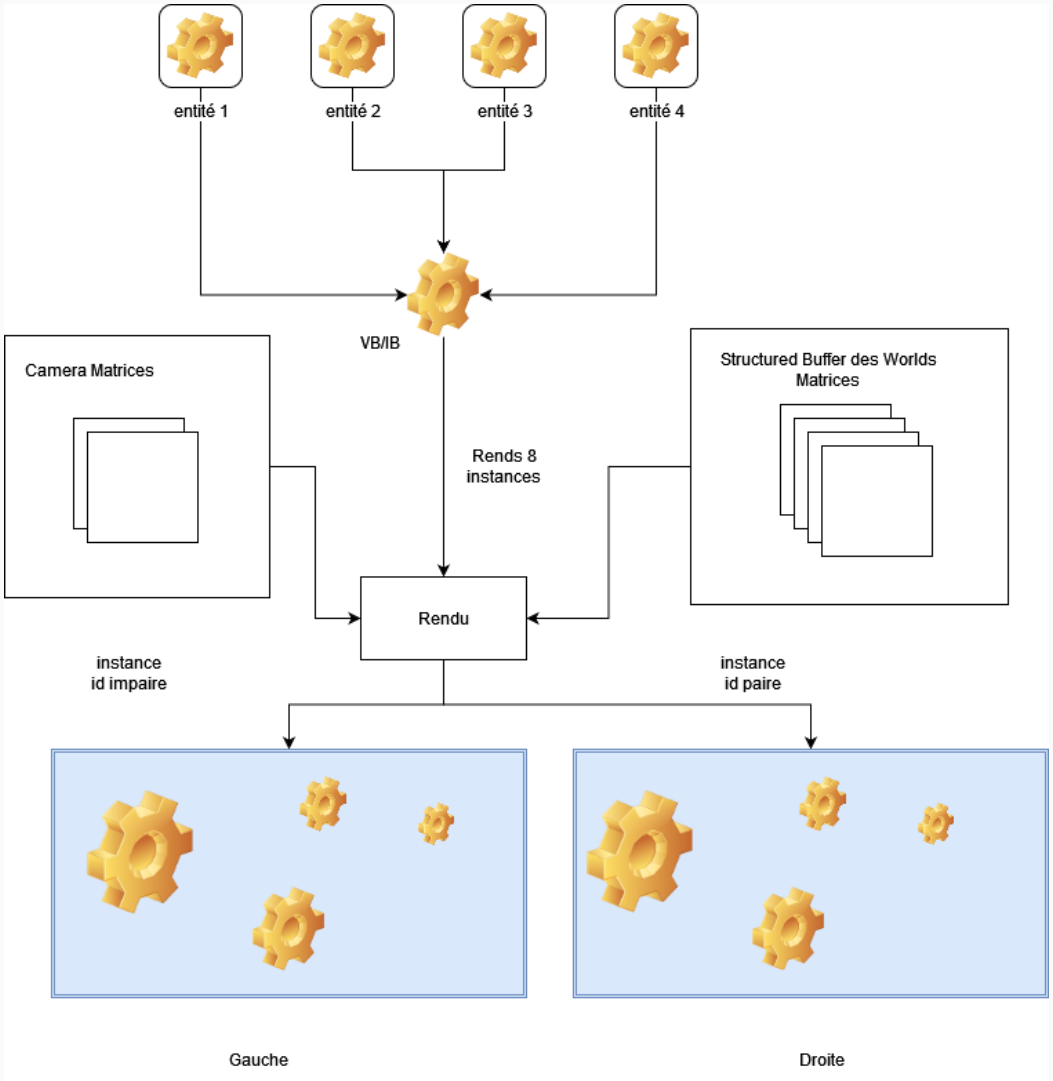



**Figure 11 :** Fonctionnement VPRT avec l'instancing