

Unity shaders

Gil Damoiseaux

LE PIPELINE DE RENDU

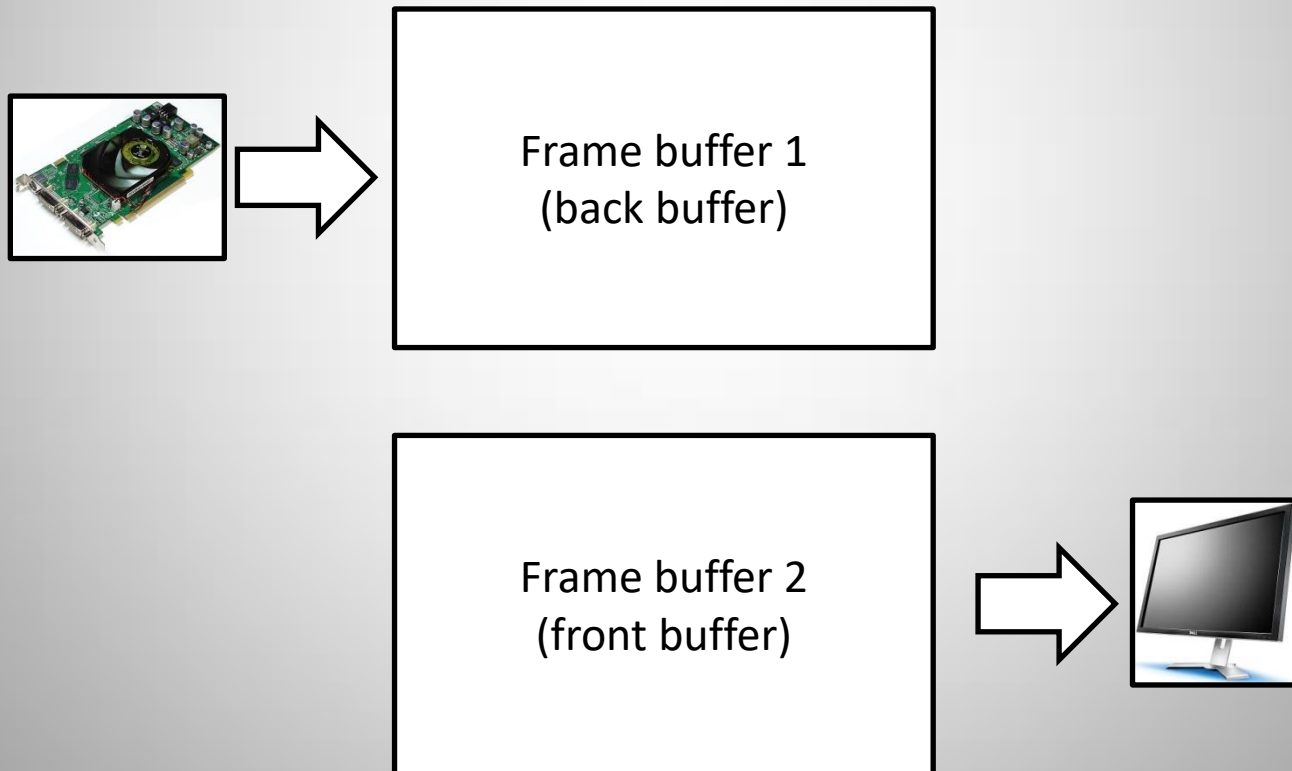
Frame buffer & Z-buffer

- **Le frame buffer**

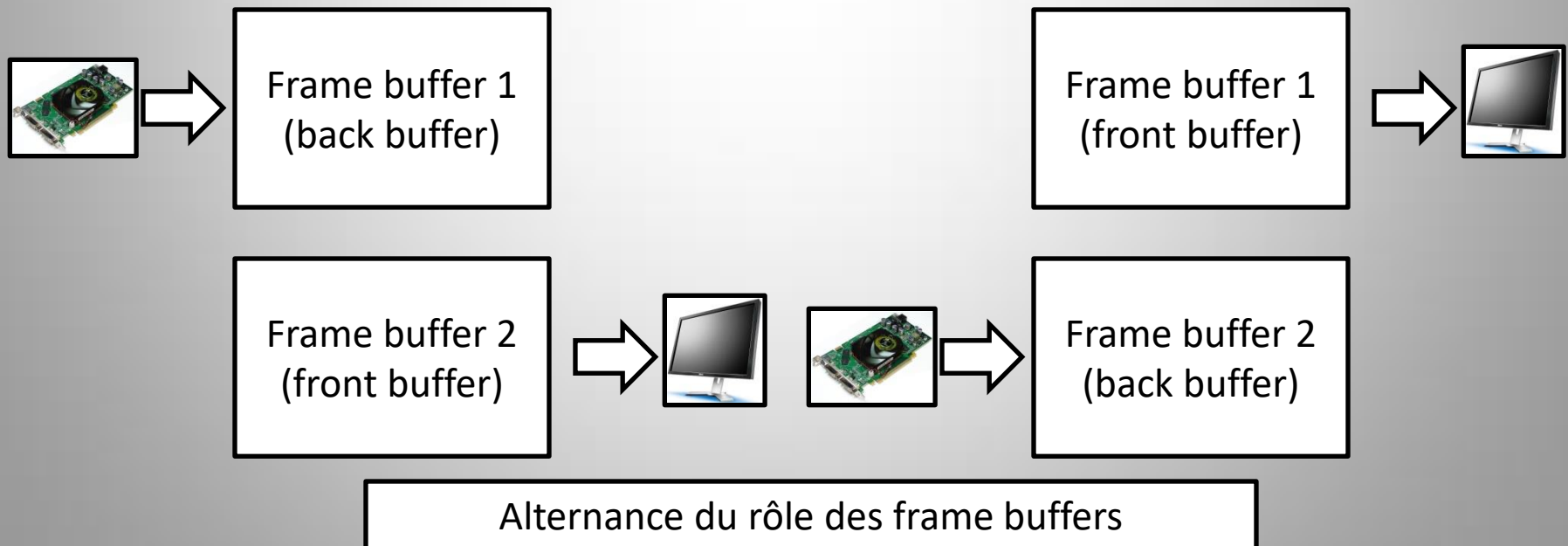
- C'est une surface/texture 2D dans laquelle est rendu l'image 3D
- Il y a deux usages de frame buffers :
 - Le front buffer est la zone mémoire qui est actuellement envoyée vers le moniteur ou vers l'écran via
 - Un DAC(digital to analog converter) lorsque l'on travaille en analogique (VGA, Component, Composite)
 - Un transfert mémoire lorsque l'on travaille en digital (DVI - HDMI)
 - Le ou les back buffers sont les zones mémoires dans lesquelles la carte graphique rend la scène.

- Double buffering:

- Un back buffer et un front buffer
- Une image est envoyée à l'écran pendant que la suivante est calculée.
- Il n'y a pas d'interférence entre l'envoi des données et le rendu de la scène.

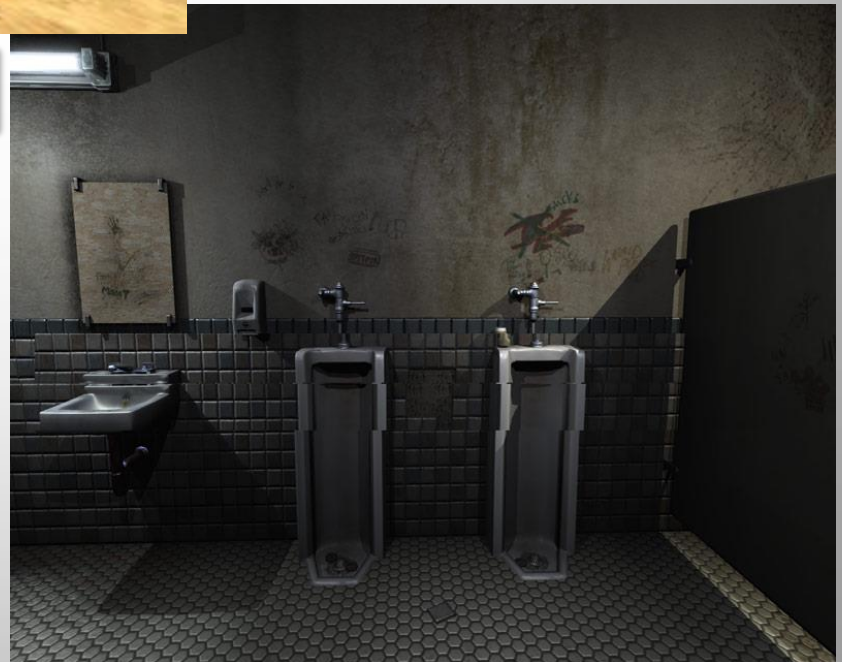


- Généralement on attend la Vsync (synchro verticale) pour basculer du front au back buffer. Ce signal (Vsync) est émis par le moniteur et dit à la carte graphique quand il a fini de faire le rendu de l'écran. L'attente du Vsync va donc :
 - » forcer le framerate à être un multiple de la fréquence de rafraichissement de l'écran : Si l'écran est dans un mode à 60Hz vous pourrez voir votre scène modifiée à 60, 30, 20, 15 ... etc FPS (frames par seconde)
 - » C'est une attente active, la carte graphique ne sait plus rien faire tant que le backbuffer n'est pas libéré.
- Sans attendre la Vsync, on tournera au framerate le plus haut que le moteur puisse rendre, **mais** on aura des problèmes de tearing (déchirement) d'images ... ce qui n'est généralement pas très joli.



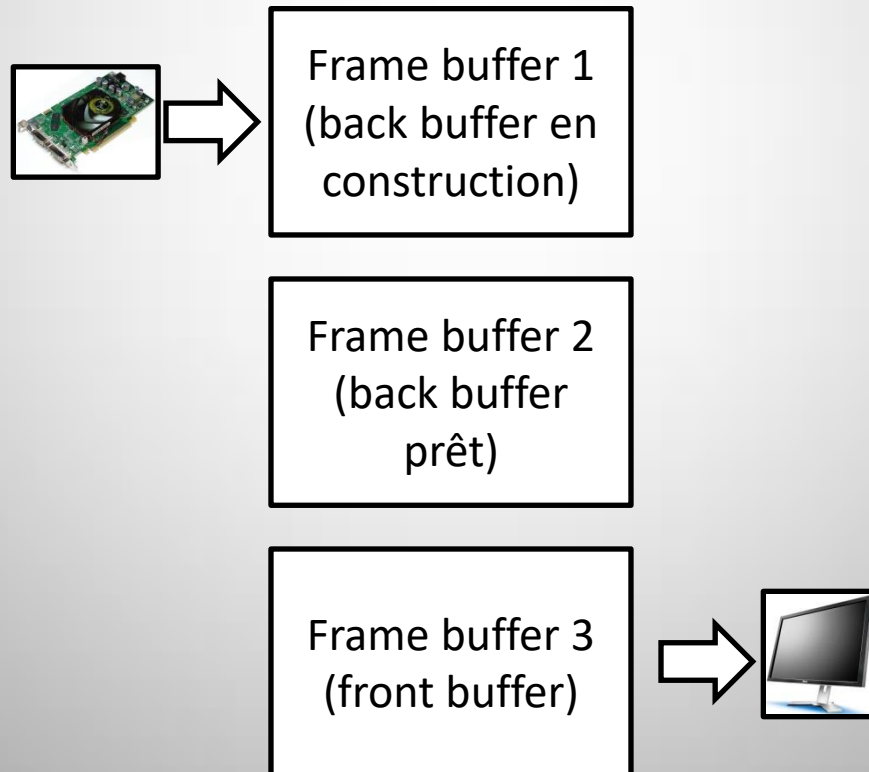


Effets de tearing

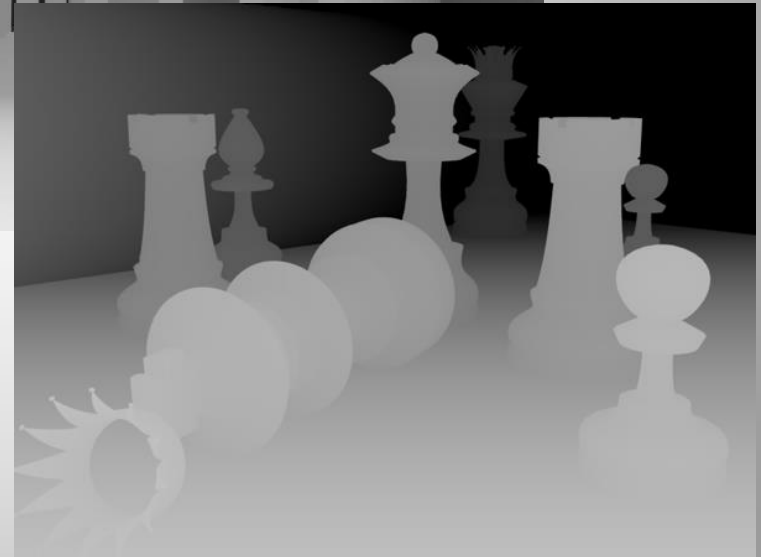


- Triple buffering:

- Deux back buffer et un front buffer
- Une image est envoyée à l'écran, une est prête à être envoyée et une autre est calculée.
- Cette méthode permet d'éviter une attente active trop longue en calculant une image tout en sachant qu'une autre peut être basculée en front à tout moment.
- Lorsque l'on attend la Vsync avec cette méthode, on optimise le nombre d'image que la machine sait rendre à un moment donnée tout en évitant les problèmes de tearing.



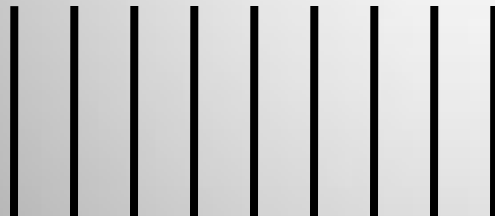
- **Le Z-buffer**
 - Contient les informations de profondeur de l'image lors d'un rendu



– Le Z buffer :

- Évite de devoir trier les faces des différents objets
- Règle les problèmes d'interpénétration de faces
- Est très peu coûteux par rapport à des algorithmes de tri et de découpage des faces.
- A une précision de 16, 24 ou 32 bits
 - 16 n'est plus utilisé car beaucoup trop faible
 - 24 l'est souvent car il n'est pas rare de devoir utiliser ce format pour pouvoir profiter du stencil buffer
 - 32 est la valeur standard quand on n'a pas besoin d'un stencil buffer
- [Voir démo](#)

- A des valeurs réparties de manière non pas linéaires mais logarithmiques ... Pourquoi?
 - Les objets plus proches de la caméra demandent plus de précision que ceux situés plus loin car ils sont plus gros à l'écran à cause l'effet de la perspective ... Une répartition linéaire serait un « gâchis » de l'espace du z-buffer
 - Si l'on veut absolument garder une répartition linéaire des valeurs (pour un jeu isométrique par exemple) on peut toujours utiliser le W-buffer qui est l'équivalent linéaire du Z-buffer

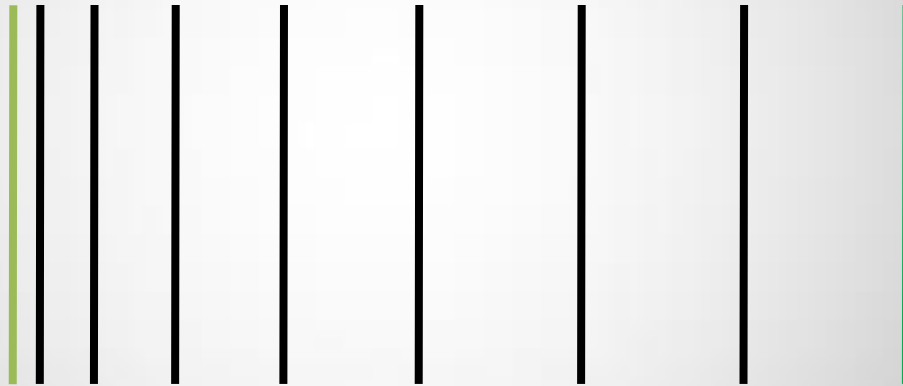


Répartition linéaire des données de profondeur

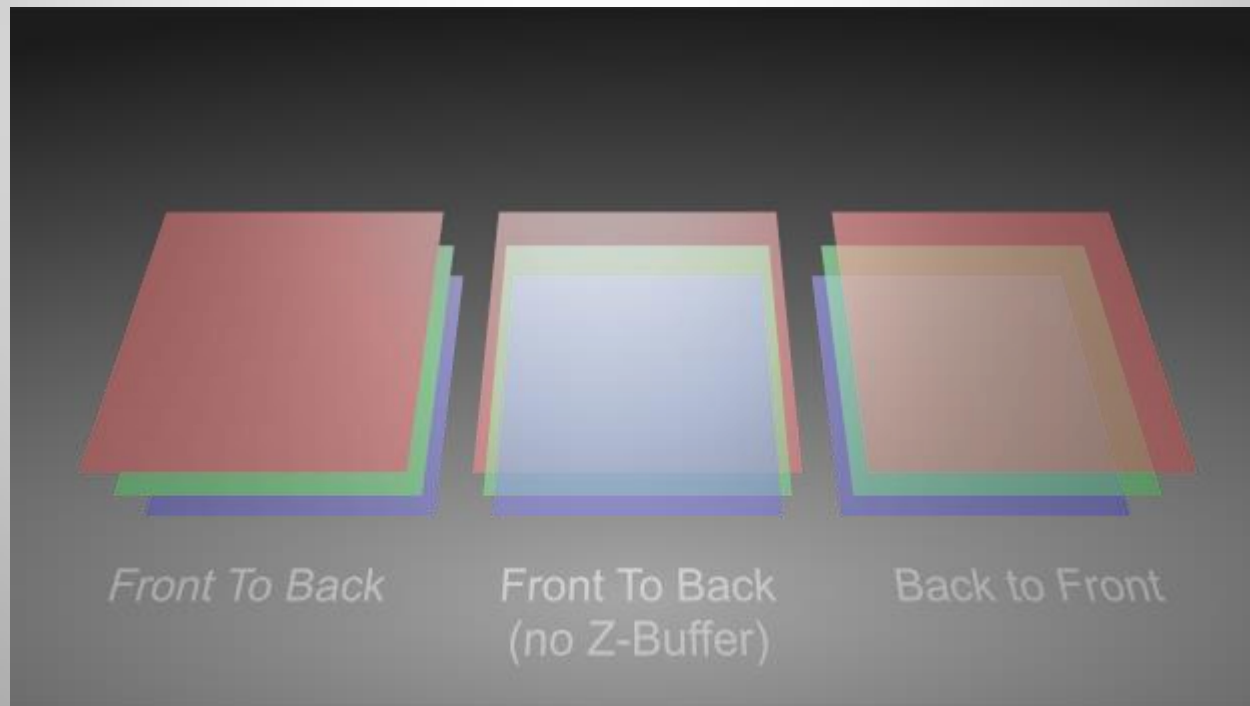


Répartition logarithmiques des données de profondeur

- A un range délimité par deux plans
 - Le near plane (plan le plus proche de la caméra)
 - Le far plane (plan le plus éloigné de la caméra)
 - [Voir démo](#)

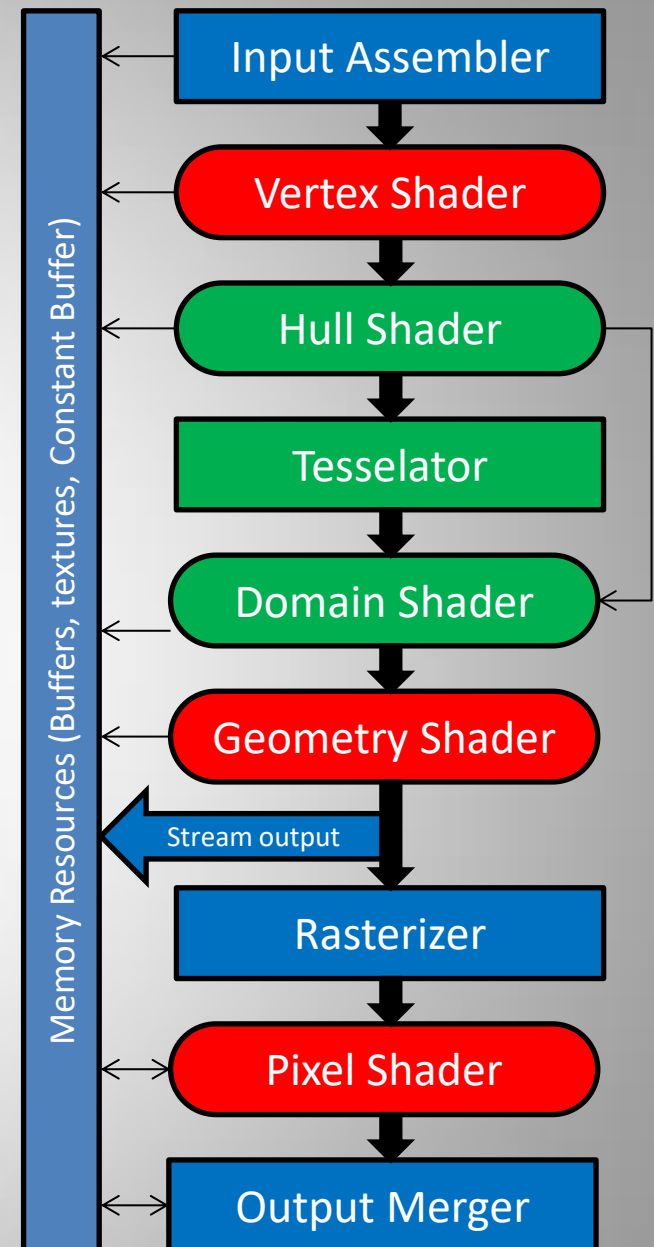
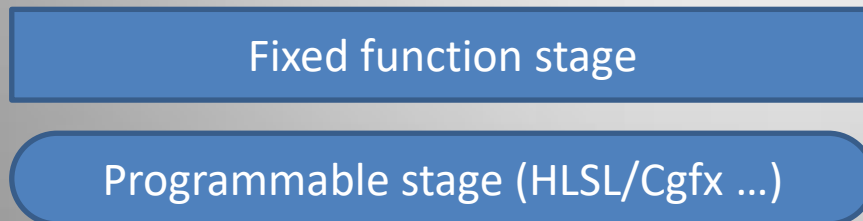


- Les problèmes liés à l’affichage des objets transparents :
 - Un objet transparent ne peut pas écrire dans le z buffer
 - Mais doit pouvoir lire dans un z buffer
 - Les objets doivent être triés de l’arrière vers l’avant, sans quoi les effets de transparence seraient combinés dans un ordre quelconque.
- [Voir démo](#)



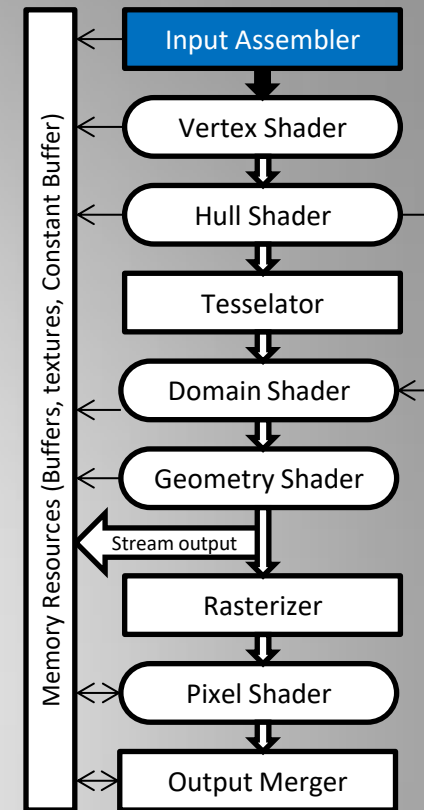
Shaders pipeline

- Voici une vue d'ensemble des différents étages de rendu sur du hardware de génération DirectX 11.
- Tous les étages peuvent être configurés à travers l'API DirectX ou OpenGL et certains étages peuvent être programmés via un langage tel que le HLSL/CgFX, GLSL, WebGL ...
- Nous allons décortiquer chacun de ces étages ...



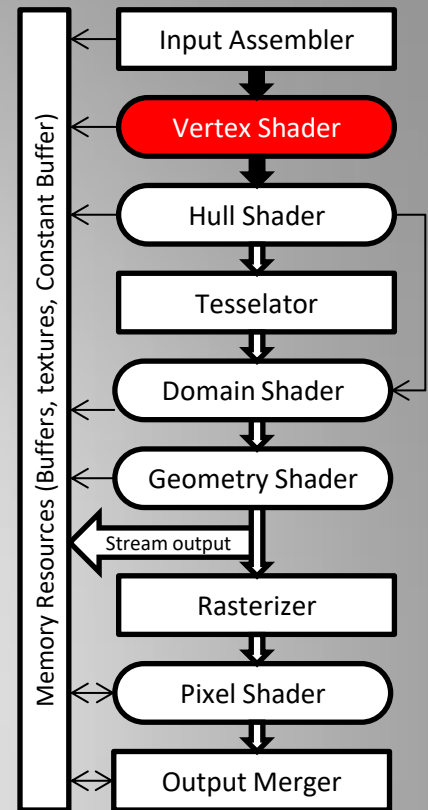
Input Assembler

- C'est le point d'entrée du pipeline, il alimente le pipeline avec des primitives de base :
 - Triangle
 - Line
 - Point
 - Patch (Uniquement quand la tessellation est active.)
- Les informations qui sortent de l'IA sont envoyées dans le vertex shader.

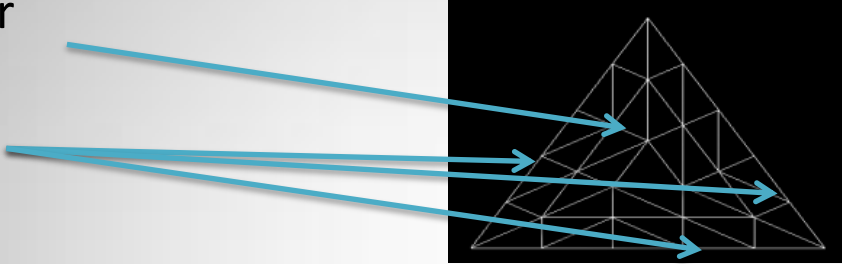


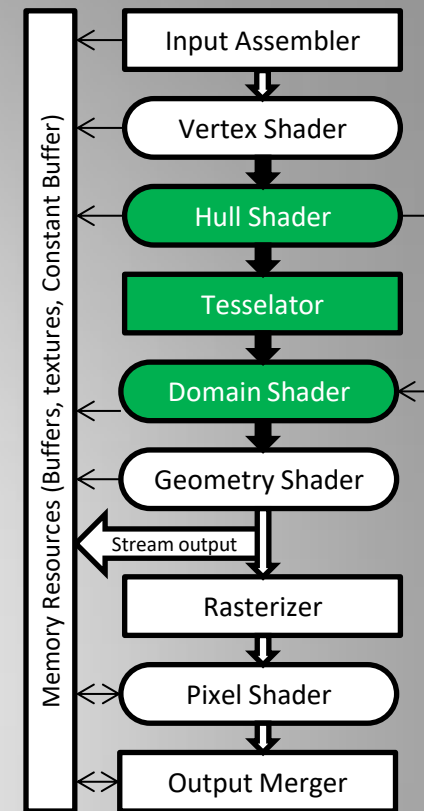
Vertex shader

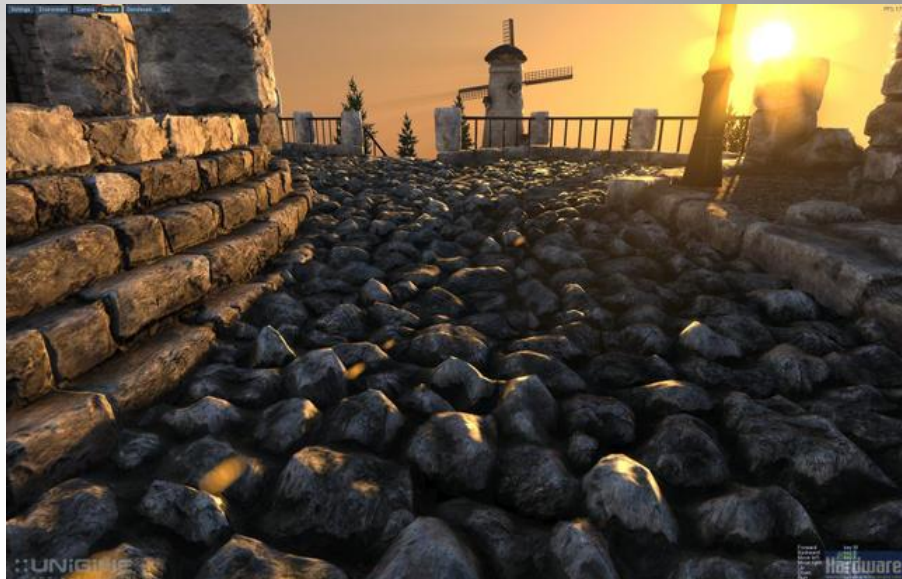
- Le vertex shader est l'étage de traitement des vertices.
- C'est généralement dans ce shader que nous allons effectuer les opérations suivantes :
 - Transformations (model->world->view->screen)
 - Skinning
 - Lighting
 - Complet dans le cas d'éclairage aux vertices ou
 - Préparation des données pour faire de l'éclairage par pixel.
- Chaque vertex est composé au maximum de 16 vecteurs de 32bits de 4 composants chacun (en entrée et en sortie) et au minimum d'un float (en entrée et en sortie)
- Les vertex shaders peuvent accéder aux texture via des instructions spécifiques (load) ne requérant pas les informations de dérivées en espace écran, par exemple pour faire du vertex displacement.



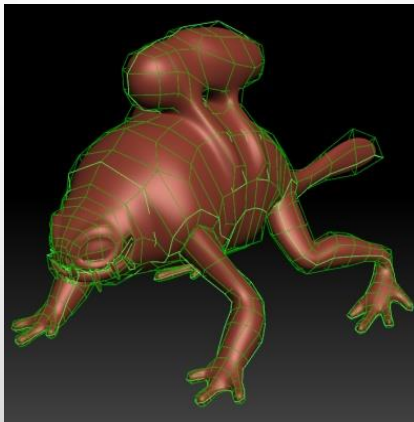
Tessellation stages

- Hull shader : Calcule les facteurs de tessellation. Il y a 4 facteurs par triangles.
 - 1x intérieur
 - 3x edges
- 
- Il est séparé en deux fonctions :
 - Génération des données constantes par primitive
 - Génération des données par vertices définissant la primitive
 - Tesselator : Calcule les coordonnées barycentriques des nouveaux point, c'est à dire exprimées de manière normalisées par rapport aux sommets du triangle.
 - La tessellation peut être discrète, continue ou pow2 en fonction des besoins
 - Domain shader : Interpole les différents attributs en fonction des coordonnées barycentriques générées par le Tesselator. Il est appelé une fois par nouveau vertex.





=

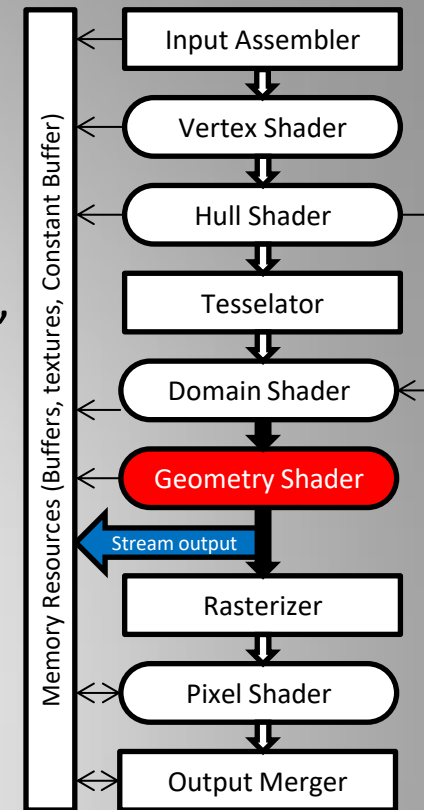


+



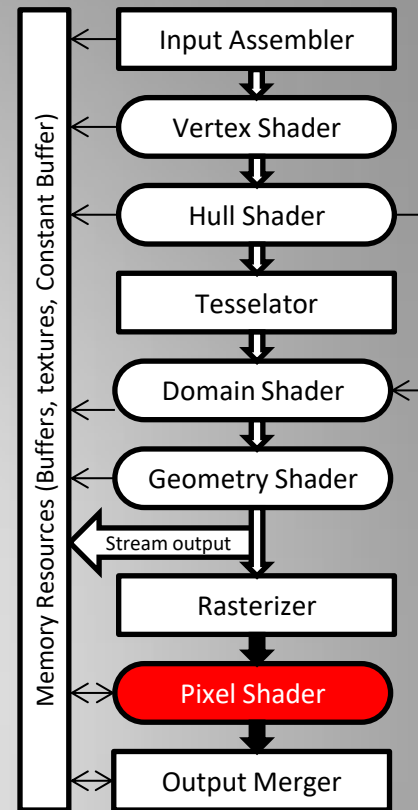
Geometry shader

- Ce type de shaders traite des primitives complètes pour en générer d'autres. Il est exécuté une fois par primitive (triangle, ligne ...) et permet, à partir de celle d'origine, de générer n'importe quel type de primitives dans la liste suivante :
 - PointStream
 - LineStream
 - TriangleStream
- Les utilisations possible sont : Génération de sprites à partir d'un nuage de points, generation de géométrie pour simuler de la fourrure ou des cheveux, végétation ...



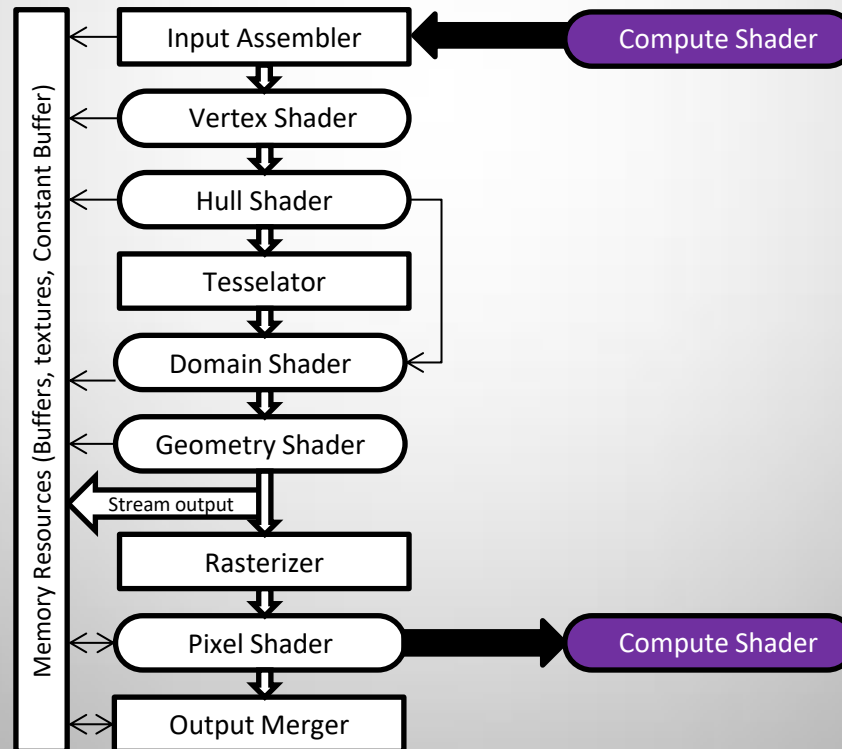
Pixel shader

- Ce shader est en bout de chaîne et est exécuté une fois par pixel (ou plus, en cas de supersampling)
- Avant d'entrer dans ce shader, on vérifie qu'il passe le Z test, sauf si celui-ci modifie le Z via son code.
- Les dérivées en espace écran des données entrant dans le shader sont calculées. Cela permet, par exemple, de choisir le bon niveau de mipmap lors de l'accès à une texture.



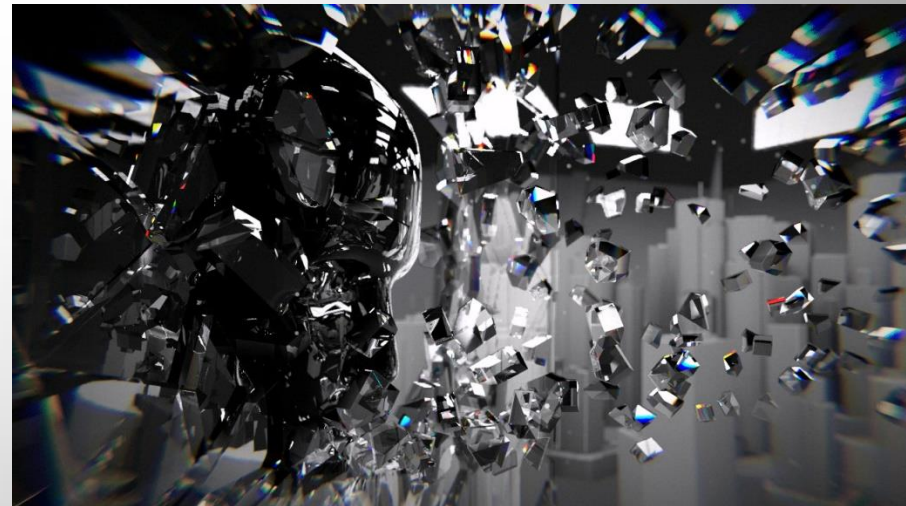
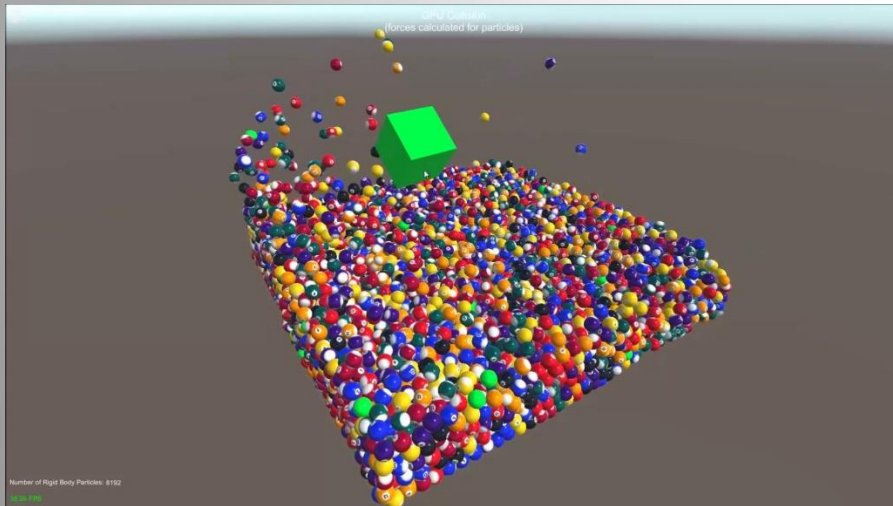
Compute shader

- Ce type de shader peut venir se greffer à plusieurs endroits dans un pipeline ...
- En début de pipeline, en modifiant des vertex buffers pour, par exemple, simuler des comportements de particules.
- En fin de pipeline pour effectuer un post process sur l'image générée.



Compute shader

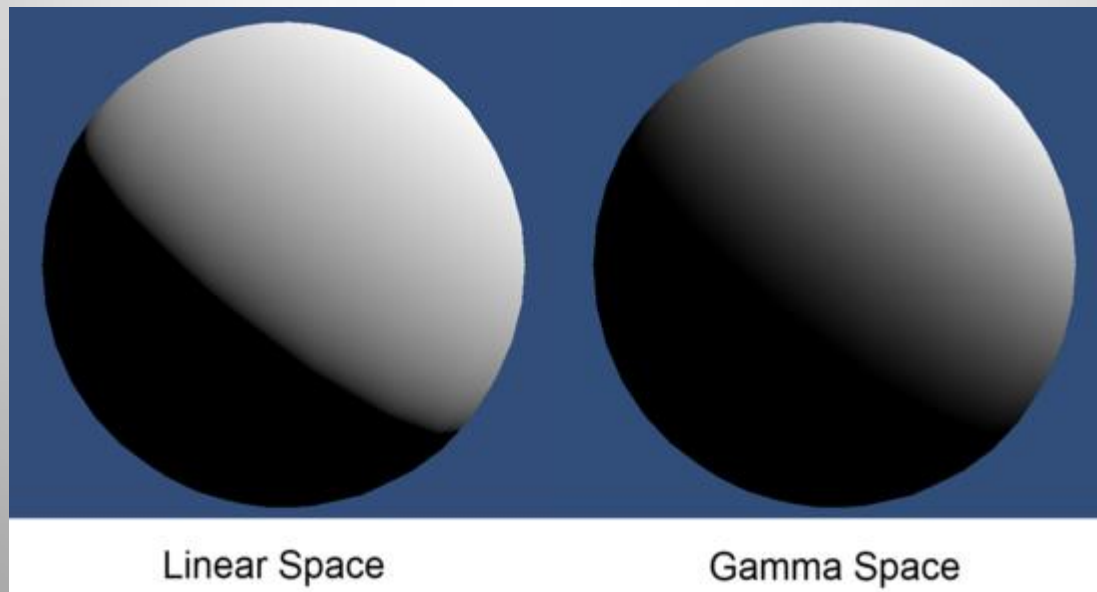
- Il permet de programmer des traitements de buffers de manière massivement parallèle.
- Ses utilisations possible :
 - Post processing (FFT, convolution, histogrammes ...)
 - Ray tracing, radiosit  ...
 - Physics (collision, particules ...)
 - AI
 - ...



PBR/PBS (PHYSICALLY BASED RENDERING-SHADING)

PBR_(physically based rendering) — PBS_(physically based shading)

- La plupart des moteurs modernes utilisent maintenant du PBR/PBS ... qu'est-ce que cela implique?
- Faire une approximation de ce que la lumière fait plutôt que de faire une approximation de ce que l'on pense intuitivement que la lumière fait.
- Travail dans un espace de couleur linéaire
 - On a eu très longtemps tendance à travailler dans l'espace de couleur sRGB comme si celui-ci était linéaire, ce qui n'est pas le cas

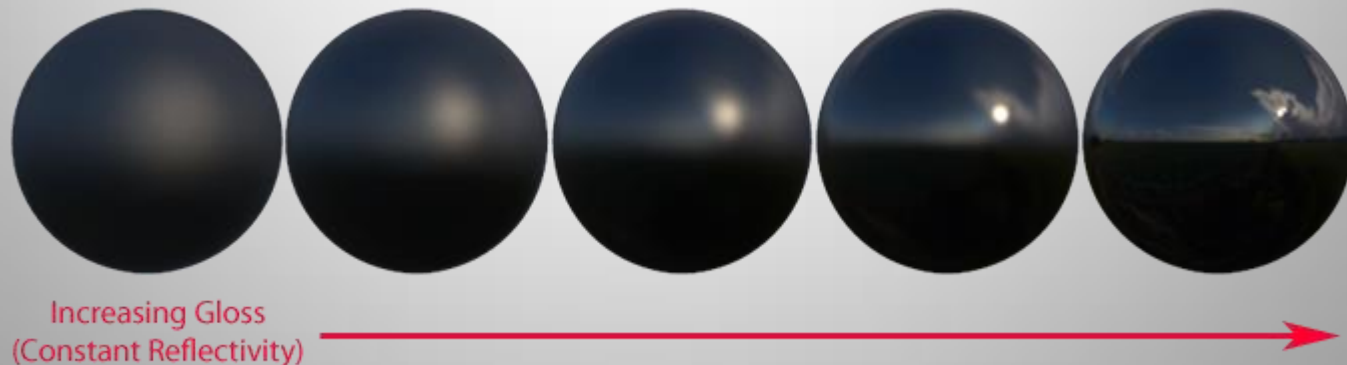


- Conservation de l'énergie

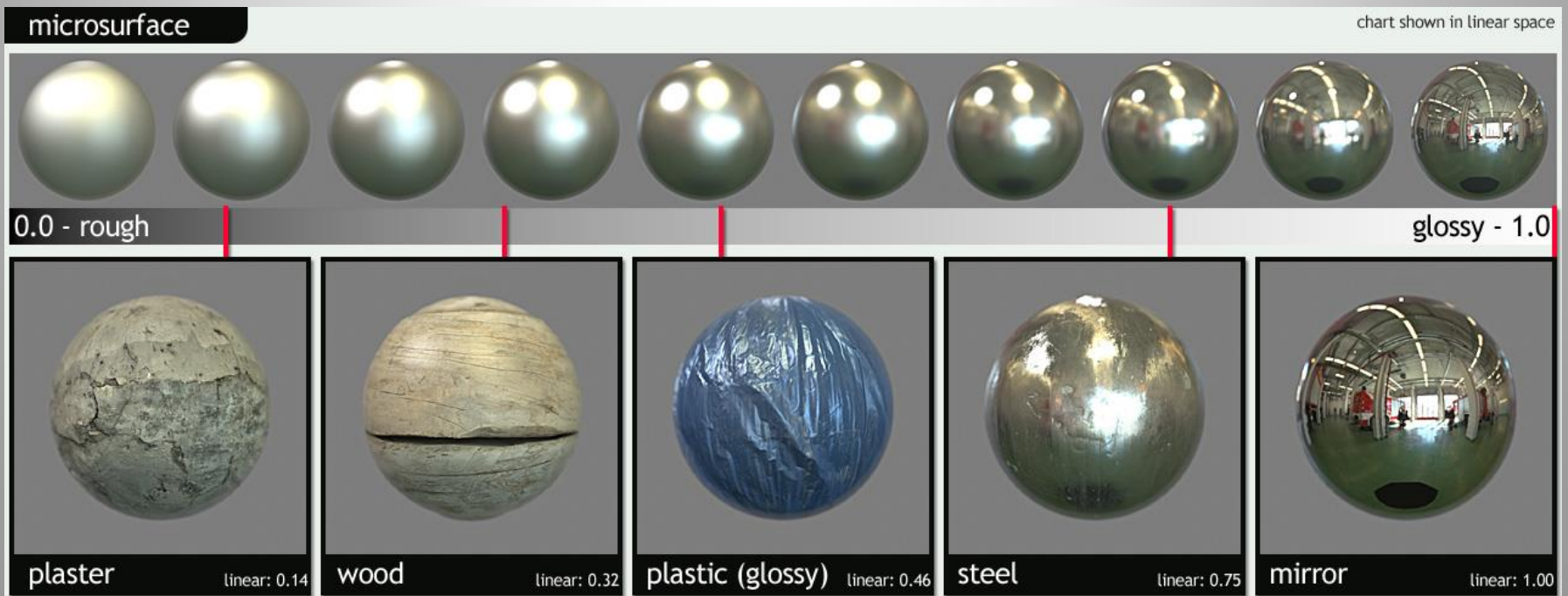
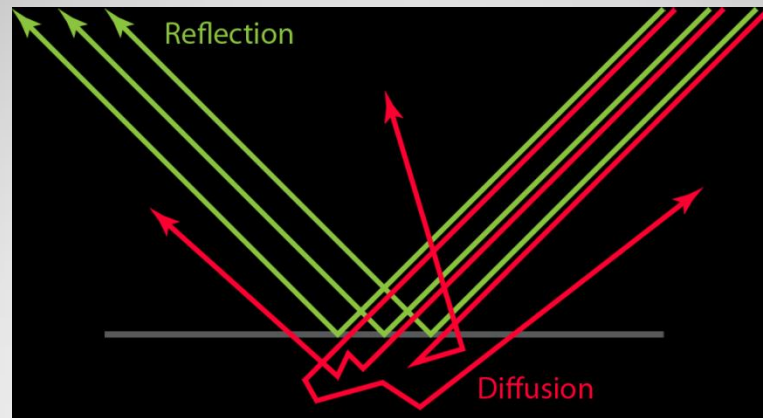
- Un objet ne peut refléter plus de lumière qu'il n'en reçoit.
- Si l'intensité spéculaire est plus importante, il y aura moins d'intensité diffuse



- Si la roughness est plus importante, la quantité de lumière réfléchie sera toujours la même et donc celle-ci sera plus étalée.



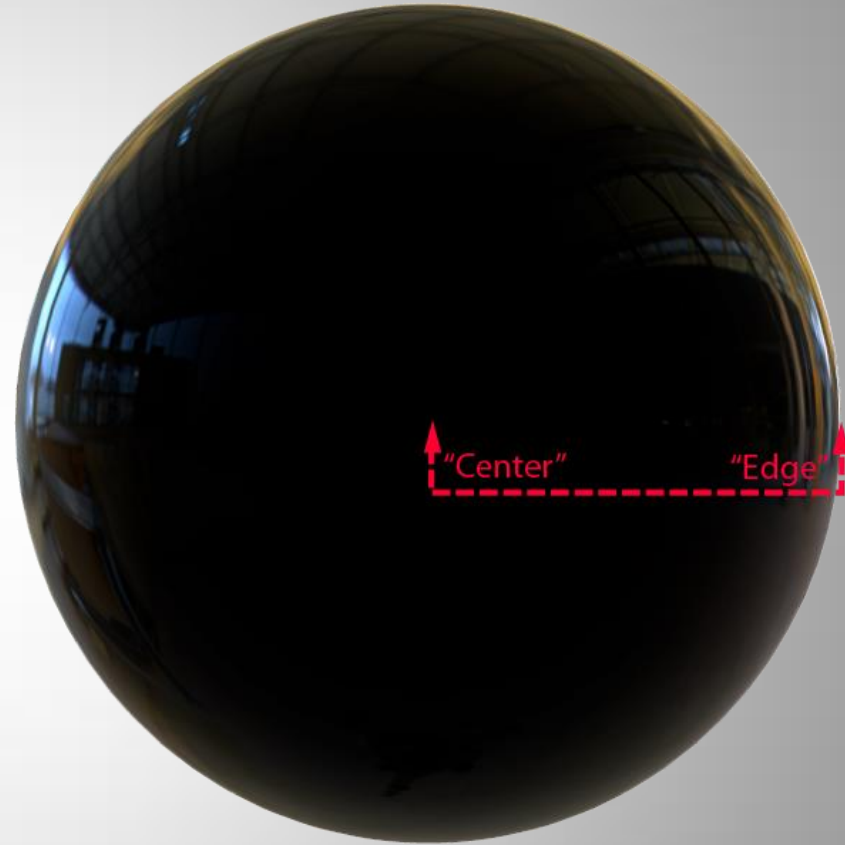
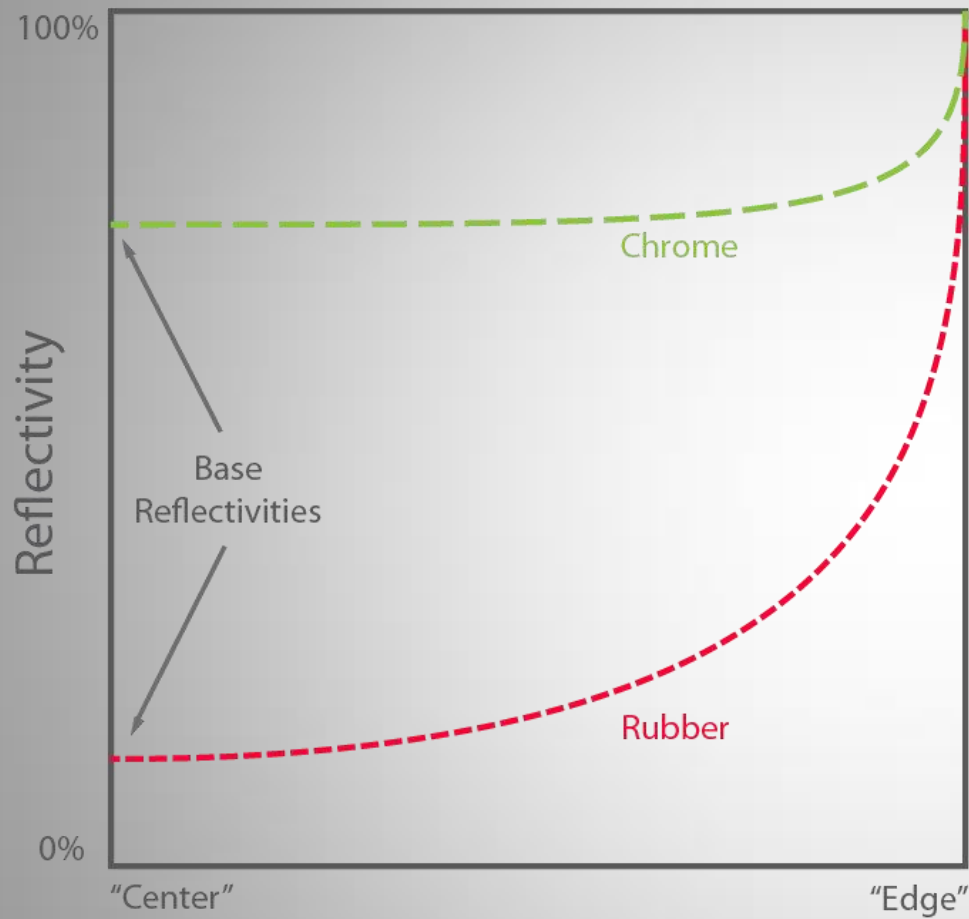
- Utilisation d'un modèle de type microsurface. On contrôle donc généralement la roughness ou smoothness du matériel et non plus la glossiness/specular intensity



- Différenciation entre les surfaces métalliques et diélectriques



- Tenir compte du facteur de fresnel



MATERIAUX ET EFFETS SPÉCIAUX

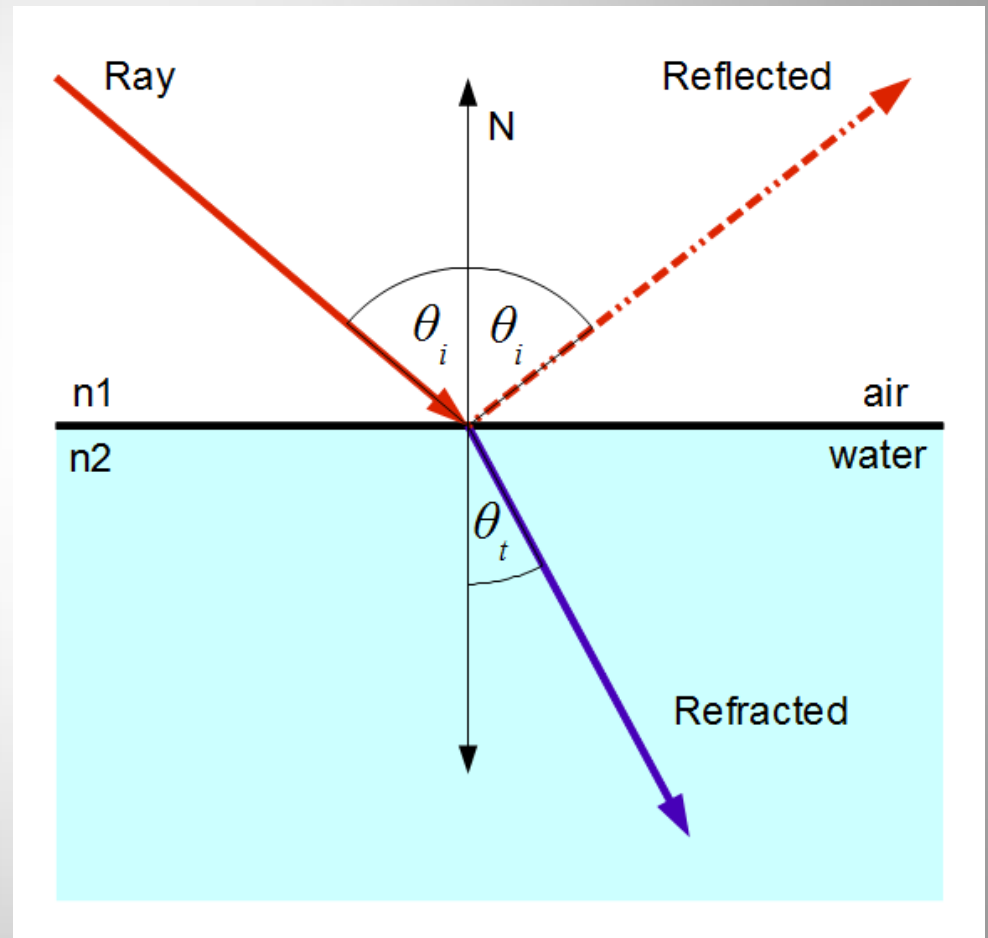
Rendu d'eau

- Le rendu d'eau réaliste est assez complexe et implique la combinaison de plusieurs techniques pointues.
- Reflexion, réfraction, caustiques, simulation physique de liquides ...



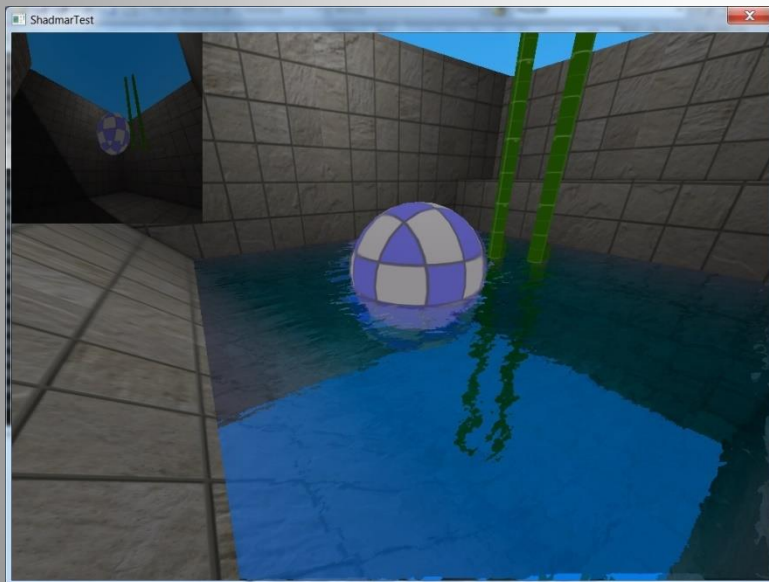
Réflexion et réfraction

- Le rendu d'eau est caractérisé par une combinaison de réflexion et réfraction.
- La somme des deux effets vaut toujours 100%, et ce qui n'est pas réfléchi est réfracté et vice-versa.
- On utilise généralement un coefficient de fresnel pour balancer les deux phénomènes : une vue rasante implique une réflexion forte, une vue perpendiculaire une réfraction forte.



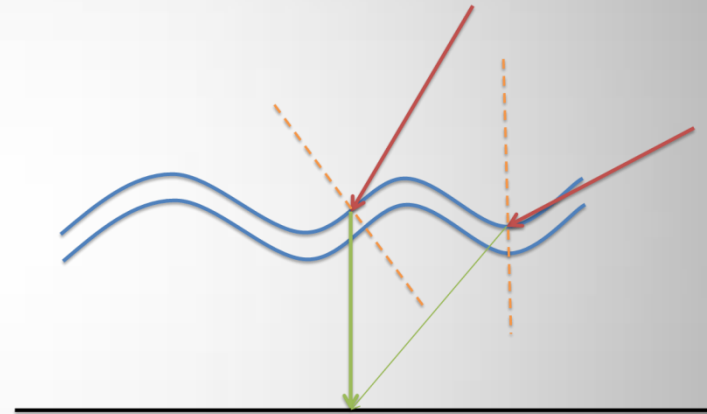
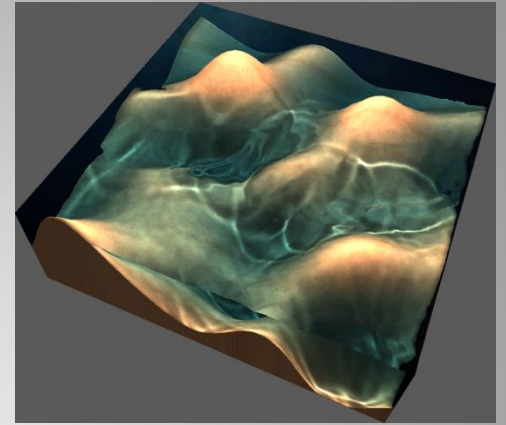
Profondeur

- Idéalement, un accès au zbuffer permettra de connaître la profondeur de l'eau en un point donné, ce qui va permettre de moduler :
- L'intensité de la réfraction (peu de réfraction en eaux peu profondes, beaucoup plus en eaux profondes).
- La couleur de l'eau (très transparent quand il y a peu d'eau, plus coloré en eaux profondes)



Caustiques

- Les caustiques sont relativement complexe a rendre en temps réel, car il s'agit de la concentration de rayons lumineux venant de directions différentes.
- La solution la plus simple consiste à utiliser une texture animée qui sera projetée sur le fond de l'eau, c'est incorrect physiquement, mais souvent convaincant.
- Il est aussi possible d'avoir des caustiques hors de l'eau (projection de la réflexion sur les murs d'une piscine par ex.) ... le même principe peut être utilisé.



Vagues

- Plusieurs solutions sont possibles :
- Une génération simple de combinaison de sinusoides déphasées pour simuler basses, moyennes et hautes fréquences, ou encore du bruit de perlin.
- Une génération plus complexe de type Gerstner qui permet de simuler les pincements des vagues dus aux flux et reflux.
- Une simulation de fluide qui permettra d'avoir des navires laissant une trainée derrière eux, des éclaboussures ou d'autres phénomènes plus complexes.

