

GÉOMÉTRIE ALGORITHMIQUE

TEXTURES ET SHADER

INTRODUCTION

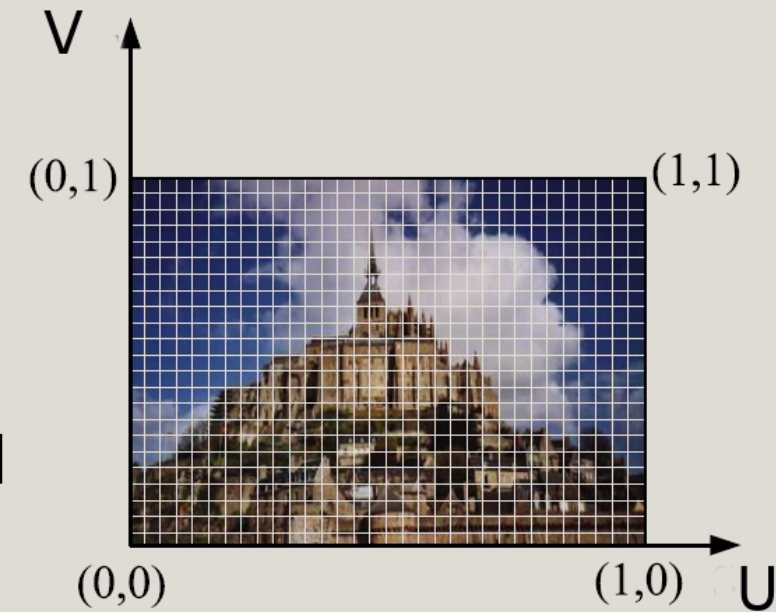
- Les objets 3D disposent que plusieurs attributs visuels qui ont été vu aux chapitres précédent :
 - Une forme : prise en charge par la modélisation et représentée par projection
 - Un comportement simple vis-à-vis de la lumière pris en compte par l'illumination
- Pour obtenir des rendus plus réalistes deux éléments sont apparus plus tardivement dans l'histoire de la représentation 3D :
 - La possibilité d'associer à chaque sommet des triangles des attributs : couleur, réflexion, réfraction, ombre, etc...
 - La possibilité d'associer à chaque pixel à l'intérieur d'un triangle des attributs par interpolation : uv (Coordonnées textures), normal, etc...
 - Un langage de programmation pour manipuler ces informations

TEXTURE MAPPING

Comment sont appliquées les textures ?

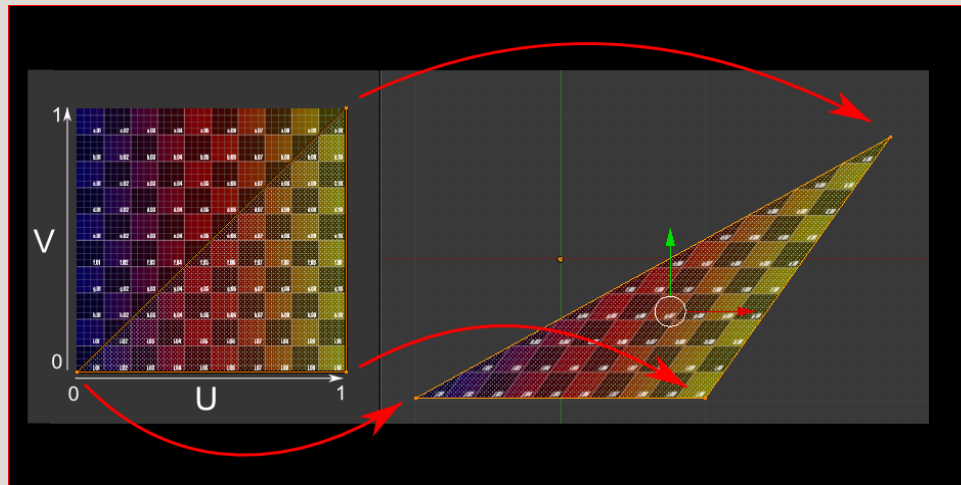
TEXTURE

- Une texture est une «image», c'est-à-dire une grille de pixels.
- Les pixels de la texture sont appelés texels (pour les différencier des pixels de l'écran graphique).
- Chaque texel est localisé dans la texture par ses coordonnées u et v .
- Toute l'image de la texture est décrite par $u \in [0,1]$ et $v \in [0,1]$



APPLICATION DE LA TEXTURE

- Consiste à associer à chaque point affiché un texel (donne au point sa couleur ou d'autres attributs).
- L'association se fait simplement en indiquant (ou en calculant) les coordonnées (u, v) du texel voulu.
- En OpenGL, l'association se fait seulement aux sommets.
- Lors du remplissage des triangles, chaque pixel est associé à un texel par interpolation des coordonnées de textures des sommets.



COORDONNÉES DE TEXTURE

Comment définir les coordonnées des textures ?

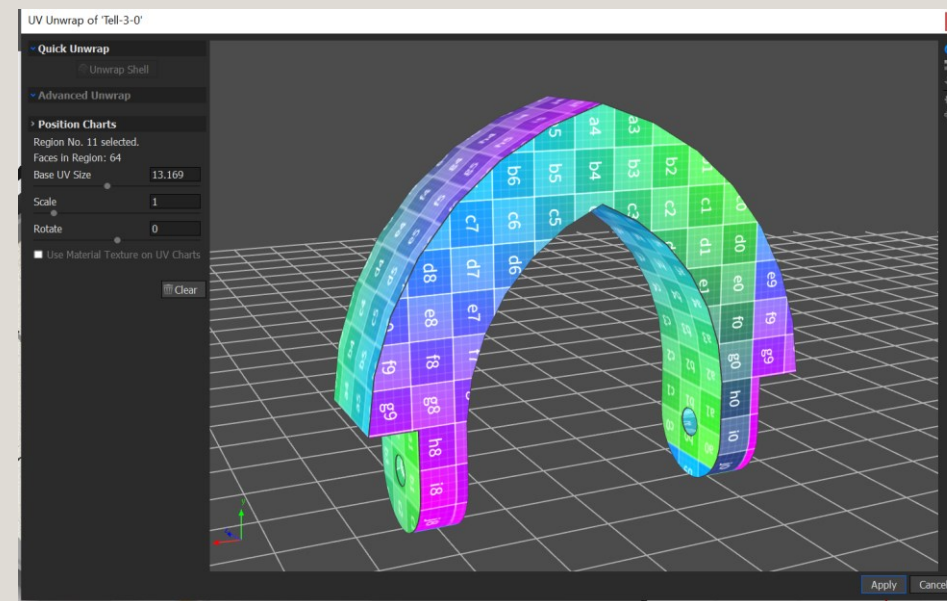
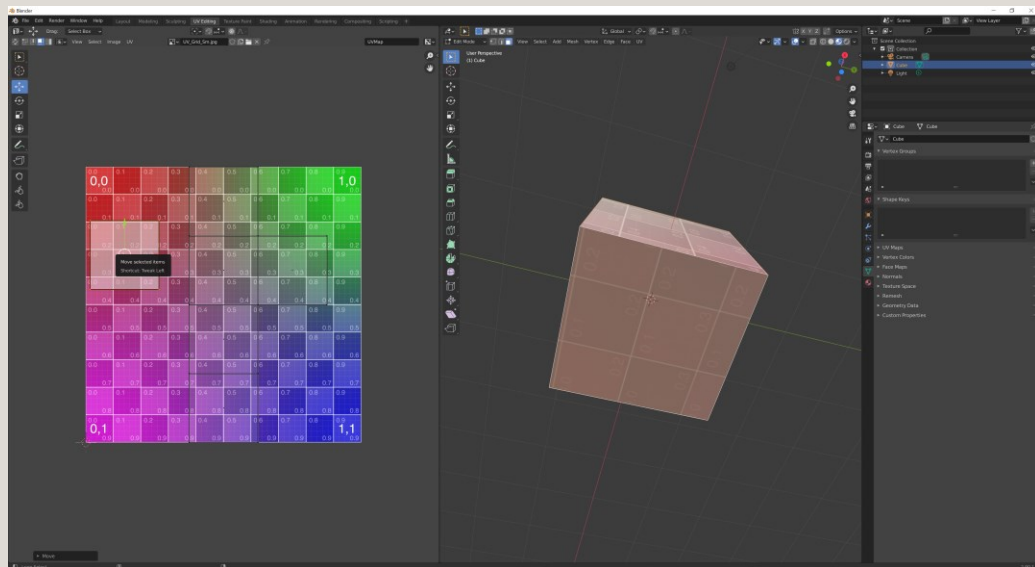
UV UNWRAPPING

- Processus qui consiste à mettre en correspondance les sommets des triangles d'une surface 3D avec les coordonnées d'une texture (U,V).
- Les logiciels de CAO proposent des outils pour réaliser cette opération qui nécessite de diviser l'objet en surfaces à projeter à plat sur une texture. Le découpage des surfaces se fait en fonction de la forme de l'objet et suivant des «coutures» (seams) définies le long des arêtes.



DEMO

● Blender et Keyshot



VIDEO UV WRAPPING

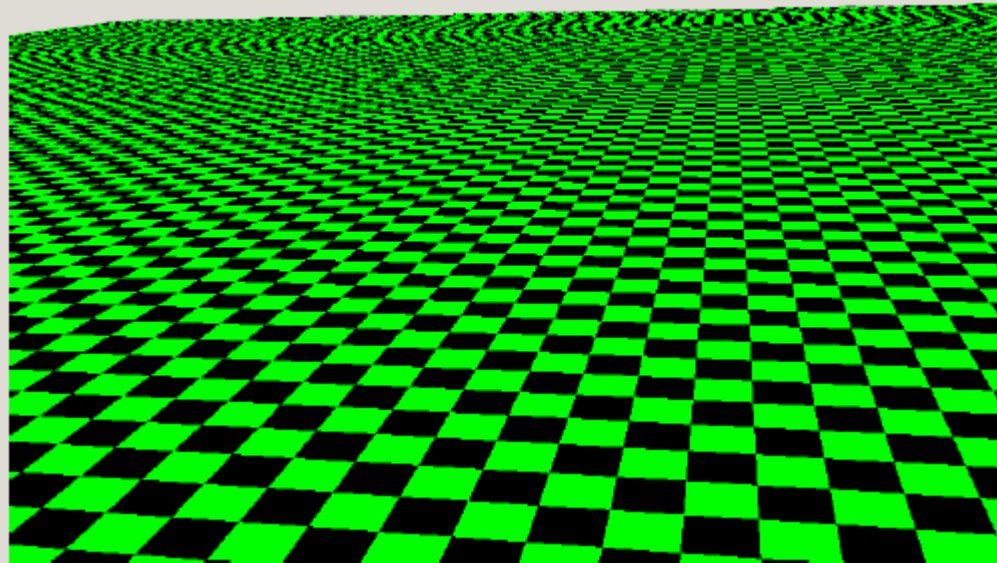


ANTIALIASING

Amélioration du rendu des textures

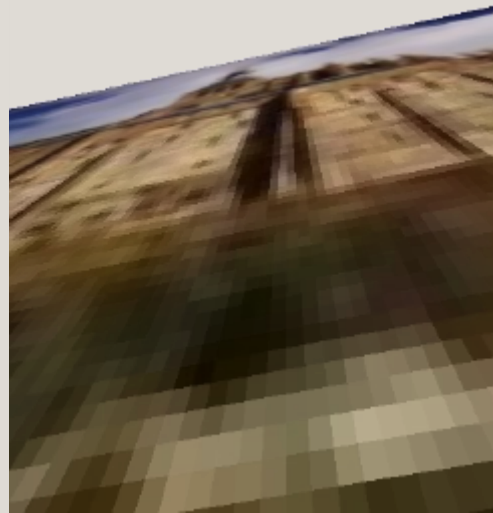
MINIFICATION

- Le passage d'un pixel à l'autre lors de la rasterization peut correspondre à un saut de plusieurs texels.
- 1 Pixel = plusieurs texel (aliassage de réduction, ou minification) = perte d'information



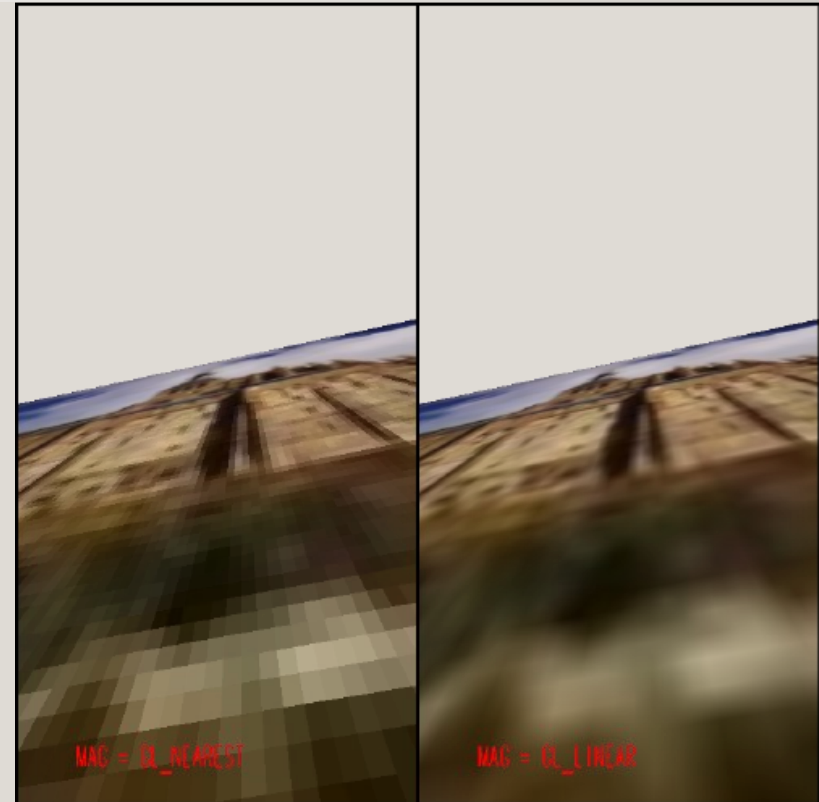
MAGNIFICATION

- Le passage d'un pixel à l'autre lors de la rasterization peut correspondre au même texel.
- 1 Texel = plusieurs pixels (aliassage d'agrandissement, ou magnification) = crénelage



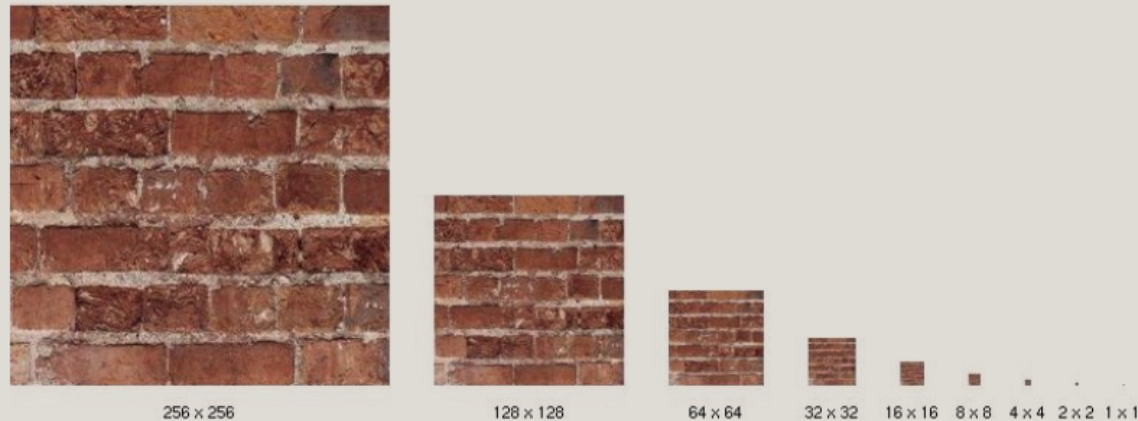
FILTRAGE BILINÉAIRE

- Pour nuancer ces effets, on peut effectuer la moyenne (pondérée) entre les 4 texels qui sont les plus proches des coordonnées (UV).
- Le mélange des couleurs obtenu donne une perception légèrement floue => permet de «lisser» la perte d'information et les marches d'escalier.



MIP-MAPPING

- Stocker plusieurs résolutions d'une même texture.
- Changer de résolution de texture selon le nombre de texels qui correspondent à un pixel (déterminé par les incréments faits sur U et V).
- Permet de nuancer les pertes d'informations (filtre d'aliassage) si les différentes résolutions sont construites par moyennage.



FILTRAGE BILINÉAIRE



BILINÉAIRE AVEC MIPMAP



TRILINÉAIRE



FILTRAGE TRILINÉAIRE

- Considérer les deux niveaux de mipmap les plus proches
- Faire un filtre bilinéaire pour chacun des deux niveaux en U,V
- Faire une moyenne pondérée entre les deux couleurs obtenus en fonction de la densité de pixels

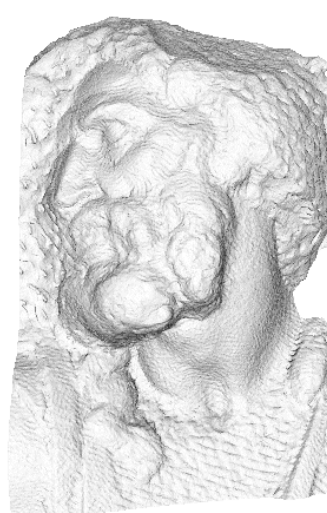


TYPES DE TEXTURES

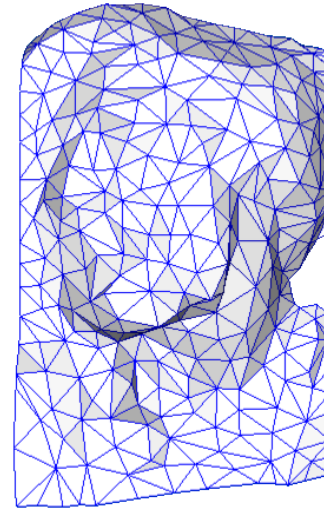
Normalmap, bumpmap, lightmap ...

NORMAL MAPPING

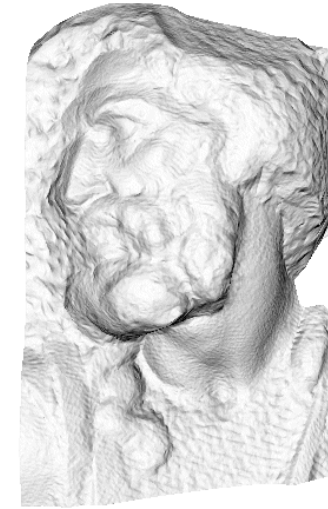
- Le normal mapping est une technique utilisée pour feindre le relief d'une texture
- Chaque texel de la texture définit la valeur la normal au niveau de chaque point d'une triangle grâce aux composantes R,G,B qui représentent les composantes X,Y,Z du vecteur normal (la normalmap à droite représente une hémisphère, un cône, une pyramide)
- La réflexion de la lumière sur la surface reproduit alors un effet similaire à l'usage de petits triangles



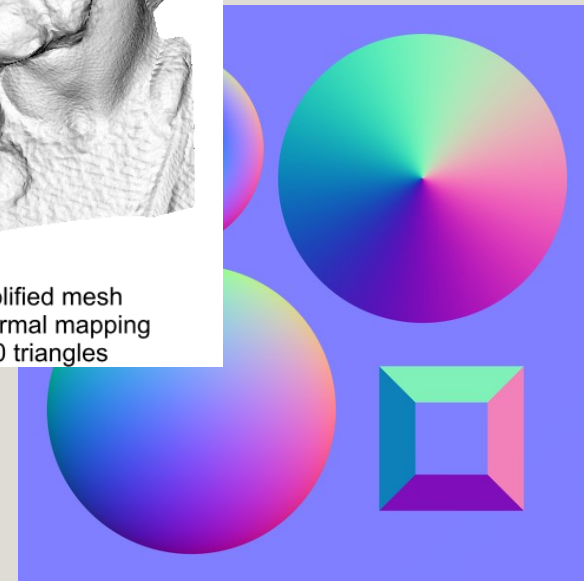
original mesh
4M triangles



simplified mesh
500 triangles

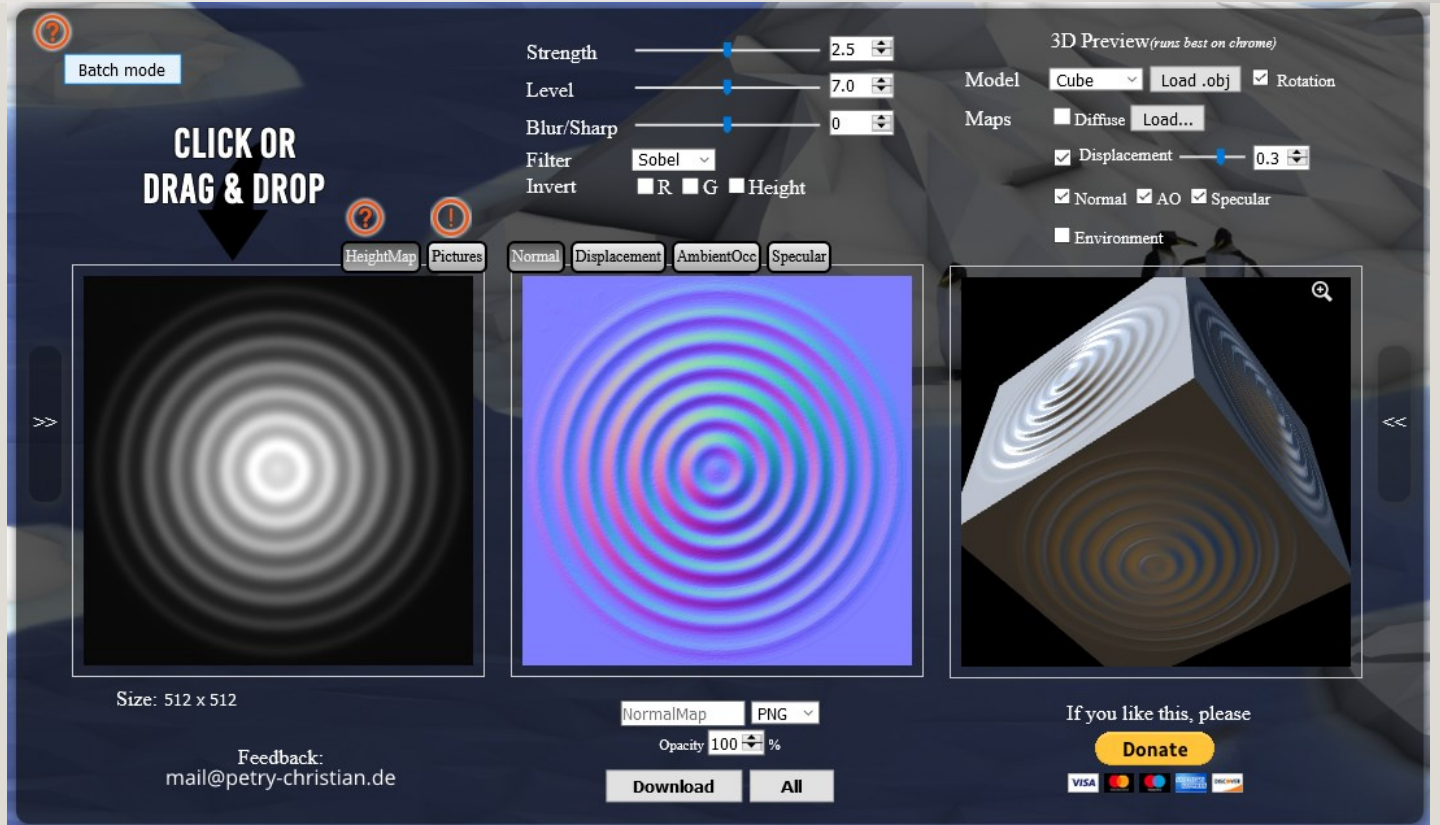


simplified mesh
and normal mapping
500 triangles



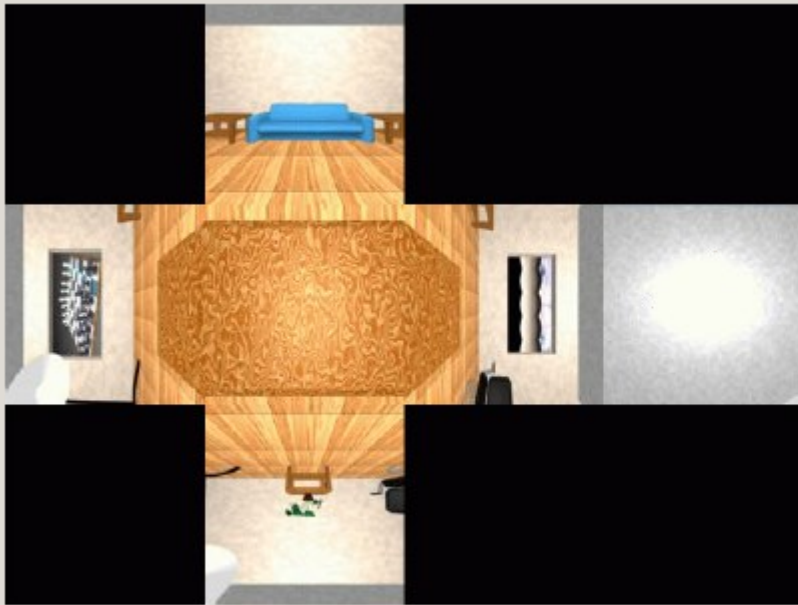
NORMAL MAPPING DEMO

- <https://cpetry.github.io/NormalMap-Online/>



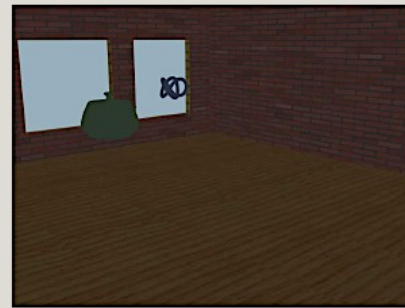
ENVIRONMENT MAPPING

- Environment Cube Mapping : calcul de réflexion basé sur un cube



LIGHTMAP

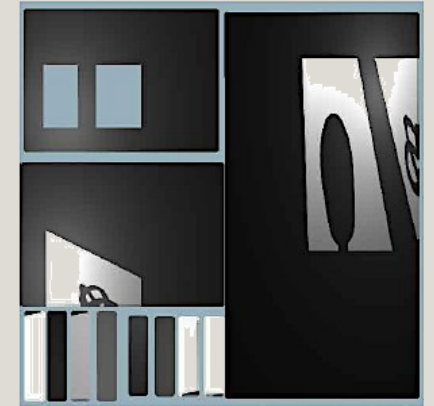
- Une source lumineuse peut-être simulée par une texture (projection d'un spot, effet spéculaire avec l'environnement mapping)



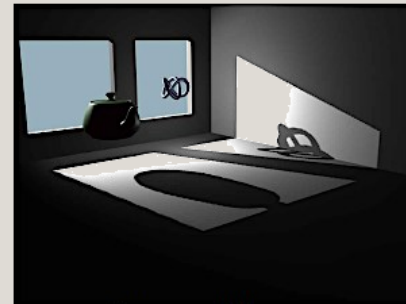
Scène



Eclairage gl



Texture lightmap (précalculée)



Placage lightmap

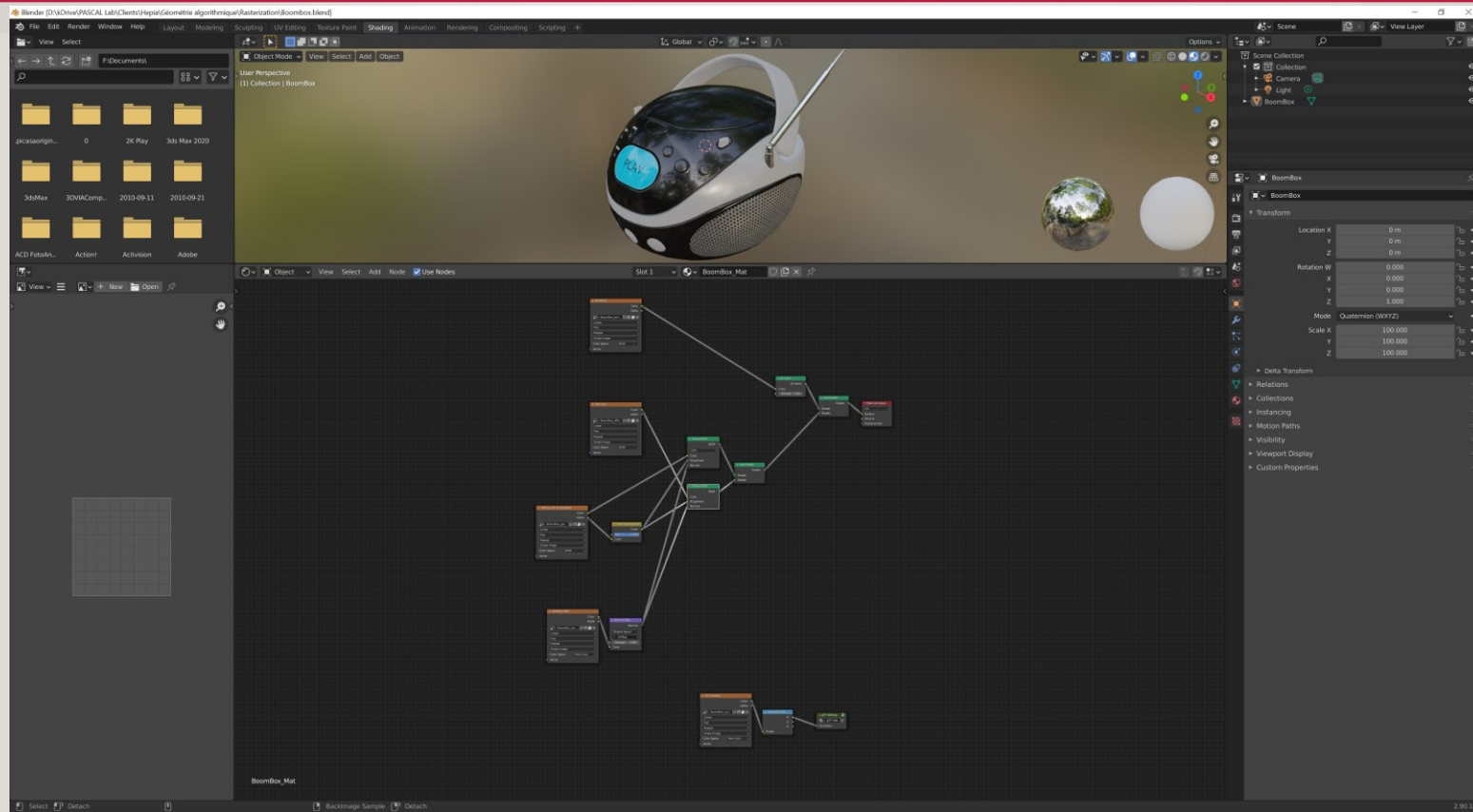


résultat final

PHYSICALLY BASED RENDERING

- Combiner plusieurs types de textures pour créer un rendu proche de la réalité en s'appuyant sur les caractéristiques physiques des matériaux :
 - Réflectivité du matériau
 - Relief avec Normal Mapping
 - Ambient Occlusion (zones peu éclairées)
 - Brillance du matériau
 - Emission de lumière

PHYSICALLY BASED RENDERING DEMO

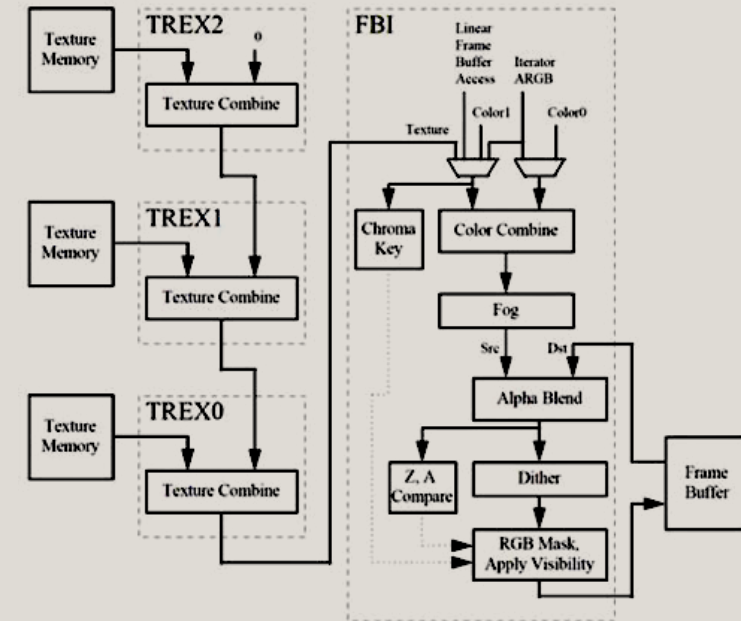


SHADERS

Introduction sur les shaders et GLSL

INTRODUCTION

- Les premières carte graphique utilisaient des puces dédiées pour calculer tous les effets graphiques : illumination, antialiasing, transparence, etc...
- Voodoo 3dFX en 1996 faisait tourner une version très « réaliste » de Tomb Raider
- Le problème c'est que la chaine était figée, les shaders sont apparus pour permettre de rendre le tout plus flexible

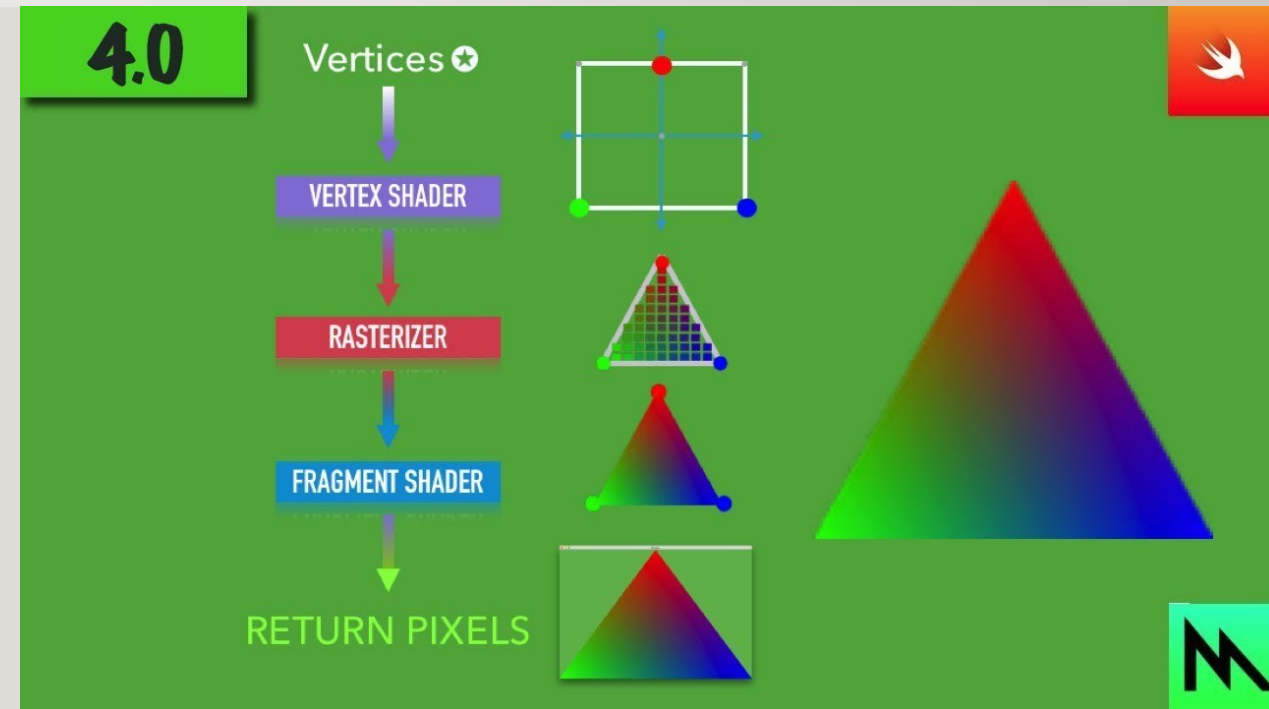


QU'EST-CE QU'UN SHADER

- Un algorithme (ici en GLSL)
- S'applique à tous les éléments d'une étape
- S'exécute sur la carte graphique
- Ne peut pas accéder directement à la mémoire de l'application
- Doit être compilé au préalable
- Algorithme instancié et exécuté une fois pour chaque élément
- Programmation parallèle
 - Indépendance des données
 - Séparation des opérations

TYPES DE SHADER

- Il y a principalement deux types de shader, le vertex shader et le fragment shader.
- Le vertex shader transforme les sommets pour les préparer pour la rasterization
- La rasterization calcul les pixels qui nécessitent d'être représentés
- Le fragment shader renvoi la couleur du pixel rasterisé
- La carte graphique dessine le pixel dans le framebuffer



GLSL

- Les shaders sont des petits programmes utilisant la syntaxe et la structure du C dans un langage appelé GLSL.

```
attribute vec3 position;
attribute vec3 normal;
uniform mat4 projection, modelview, normalMat;
varying vec3 normalInterp;
varying vec3 vertPos;

void main(){
    vec4 vertPos4 = modelview * vec4(position, 1.0);
    vertPos = vec3(vertPos4) / vertPos4.w;
    normalInterp = vec3(normalMat * vec4(normal, 0.0));
    gl_Position = projection * vertPos4;
}
```

```
precision mediump float;
varying vec3 normalInterp; // Surface normal
varying vec3 vertPos;      // Vertex position
uniform int mode;          // Rendering mode
uniform float Ka;           // Ambient reflection coefficient
uniform float Kd;           // Diffuse reflection coefficient
uniform float Ks;           // Specular reflection coefficient
uniform float shininessVal; // Shininess

// Material color
uniform vec3 ambientColor;
uniform vec3 diffuseColor;
uniform vec3 specularColor;
uniform vec3 lightPos;     // Light position

void main() {
    vec3 N = normalize(normalInterp);
    vec3 L = normalize(lightPos - vertPos);

    // Lambert's cosine law
    float lambertian = max(dot(N, L), 0.0);
    float specular = 0.0;
    if(lambertian > 0.0) {
        vec3 R = reflect(-L, N); // Reflected light vector
        vec3 V = normalize(-vertPos); // Vector to viewer
        // Compute the specular term
        float specAngle = max(dot(R, V), 0.0);
        specular = pow(specAngle, shininessVal);
    }
    gl_FragColor = vec4(Ka * ambientColor +
                        Kd * lambertian * diffuseColor +
                        Ks * specular * specularColor, 1.0);
}
```


VARIABLES DES SHADER

- Chaque variable (globale) correspond à une donnée.
- Chaque donnée possède (en plus de son type) un qualificateur précisant sa nature
- Des données qui traversent le pipeline
 - Des données qui entrent dans le pipeline (qualificateur attribute)
 - Des données qui transitent (entre les shaders) (qualificateur varying)
- Des données qui ne traversent pas le pipeline mais servent à en contrôler ou modifier l'exécution (qualificateur uniform)

DEMO

- ShaderToy : <https://www.shadertoy.com/>
- SHDR : <https://pascal-lab.ch/shdr/>

VERTEX SHADER

Utiliser les informations des sommets

STRUCTURE

- En entrée du shader on trouve les informations indiquées dans le model 3D : position, normal, etc... Elles concernent les triangles en cours de projection.
- On trouve aussi des paramètres indépendants de l'index du sommet en cours de traitement. Typiquement le produit des matrices monde x vue, la matrice de projection, une matrice normale.
- Finalement on a les paramètres de sortie interpolés bilinéairement (varying) qui sont transmis ensuite au fragment shader.
- Il n'y a pas de return mais le résultat est affecté à la variable globale : **gl_Position**

```
precision highp float;
attribute vec3 position;
attribute vec3 normal;
uniform mat3 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
varying vec3 fNormal;
varying vec3 fPosition;

void main()
{
    fNormal = normalize(normalMatrix * normal);
    vec4 pos = modelViewMatrix * vec4(position.xyz, 1.0);
    fPosition = pos.xyz;
    gl_Position = projectionMatrix * pos;
}
```


FRAGMENT SHADER

Utiliser les informations des pixels

STRUCTURE

- En entrée du shader on trouve les variables définies par le vertex shader : fPosition et fNormal.
- On a également des paramètres indépendants du pixel en cours de traitement : time et resolution.
- En sortie la variable global gl_FragColor contient un vecteur R, G, B, A à écrire dans le framebuffer (mémoire stockant les pixels dessinés à l'écran).

```
precision highp float;
uniform float time;
uniform vec2 resolution;
varying vec3 fPosition;
varying vec3 fNormal;

void main()
{
    gl_FragColor = vec4(fNormal, 1.0);
}
```