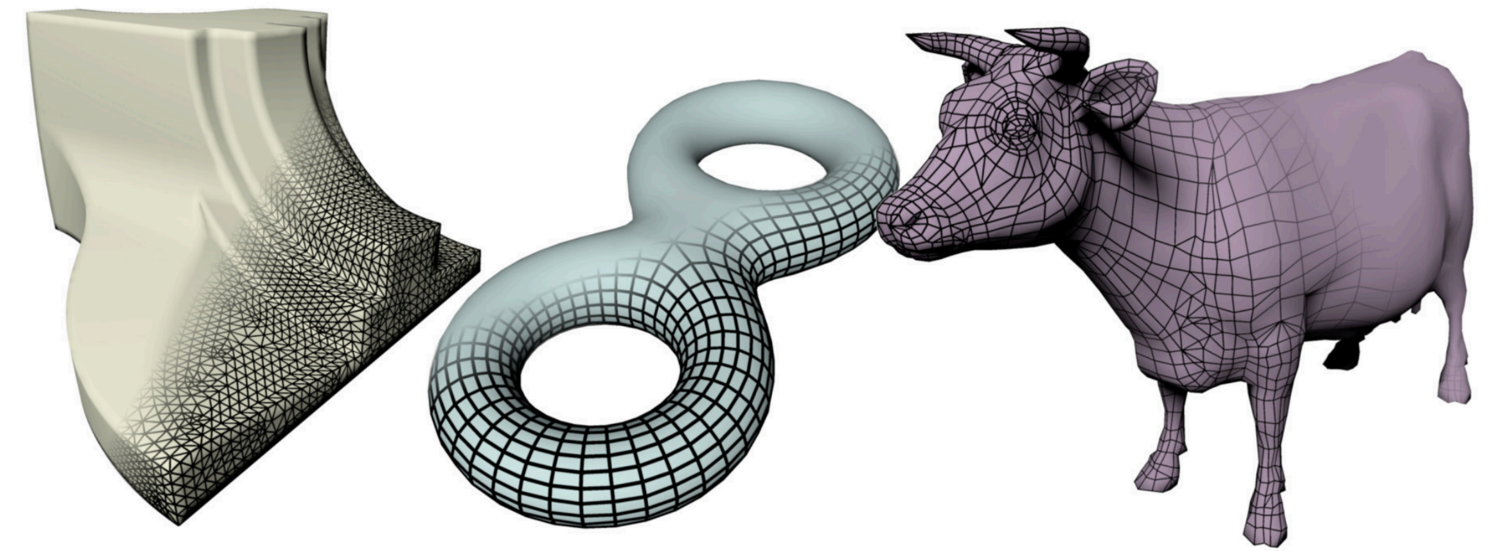
Représentation et modélisation de surfaces

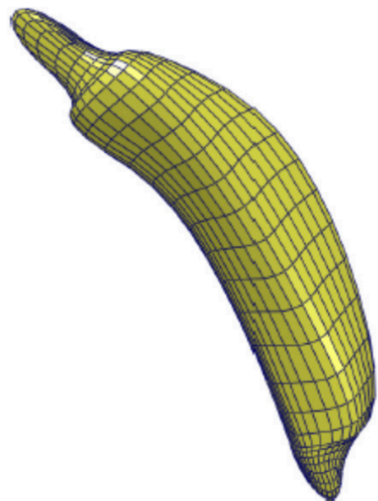
- Théorie des surfaces lisses :
- Courbures
- **Maillages**
- Surfaces Splines
- Surface de subdivision
- Ensemble de points

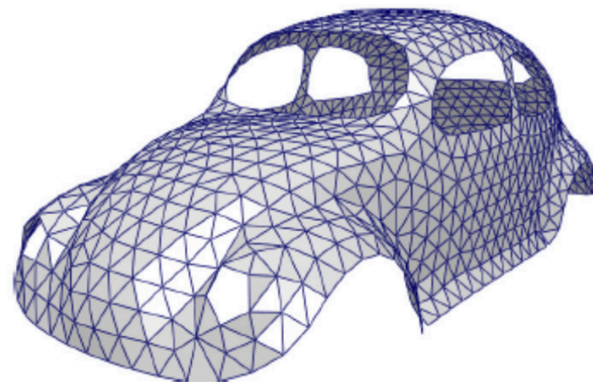
Maillage - structure

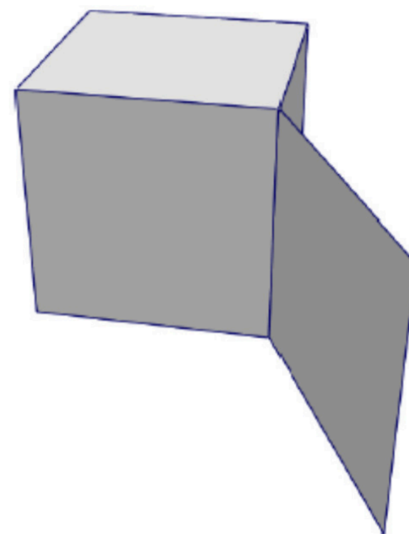
Maillage: Ensemble de polygones partageant des sommets
N_f faces, N_v sommmets (/vertices), N_e arêtes (/edges)
Triangulation: Toutes les faces sont des triangles
Quad mesh: Toutes les faces sont des quadrangles
Idéalement quadrangle planaires.
Poly mesh: Les faces sont des polygones quelconques



Rem. Maillage représente une variété si chaque arête est partagée par 2 faces au plus.







Maillage - Data structure

Encodage typique de maillage triangulaire

```
struct vec3 { float x, y, z;};
struct int3 { int i, j, k; };
std::vector<vec3> position;
std::vector<int3> connectivity
```

`std::vector` assure la contiguité en mémoire.

Examples

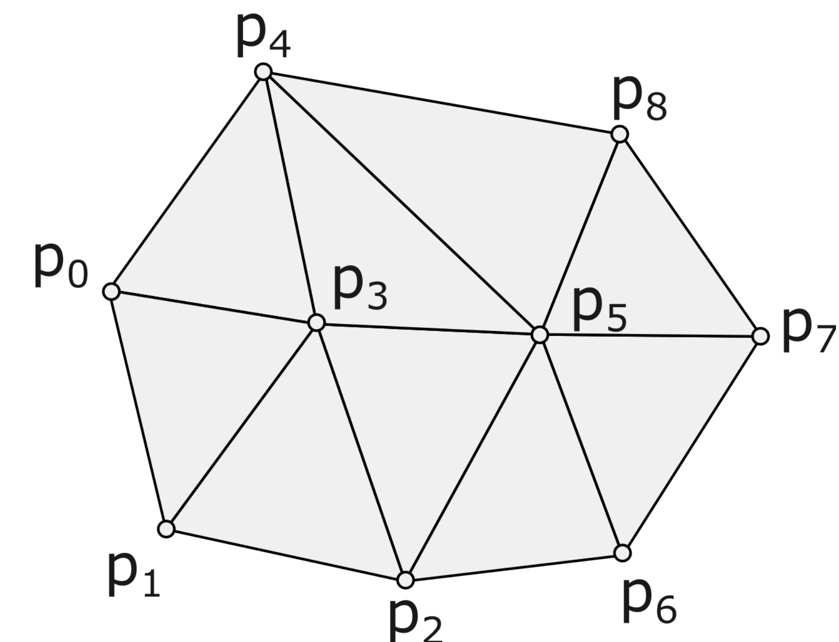
Tétraèdre

```
std::vector<vec3> position = { {0,0,0}, {1,0,0}, {0,1,0}, {0,0,1} };
std::vector<int3> connectivity = { {0,1,2}, {0,1,3}, {0,2,3}, {1,2,3} };
```

Maillage

```
std::vector<vec3> position = { p0,p1,p2,p3,p4,p5,p6,p7,p8 };
std::vector<int3> connectivity = { {0,1,3}, {1,2,3}, {0,3,4}, {3,5,4},
                                   {2,5,3}, {2,6,5}, {5,6,7}, {5,7,8}, {4,5,8} };
```

Rem. $(0,1,3)$ similaire à $(1,3,0)$, similaire à $(3,0,1)$
Mais $(0,1,3)$, orientation inverse de $(3,1,0)$, $(1,0,3)$, $(0,3,1)$



Maillage - cas du code en TP

- Définie `mesh_drawable` variable dans `scene.hpp`
  - *Variable partagée dans plusieurs méthodes de la classe*
- Initialise une structure `mesh` dans `initialize()`
  - *`mesh`: tableaux de données en RAM/CPU*
- Transfère les données de `mesh` vers `mesh_drawable`
  - *Envoie de données sur la mémoire du GPU*
- Dans `display_frame`
  - MAJ paramètres de la forme
    - *écriture sur paramètres envoyés*
      en tant qu'uniforms lors de l'appel à draw();
  - Appel à `draw()`
    - \- Activation du shader (drawable.shader)
    - \- Envoie des paramètres uniformes
      (drawable et environnment)
    - \- Appel à `glDraw()`

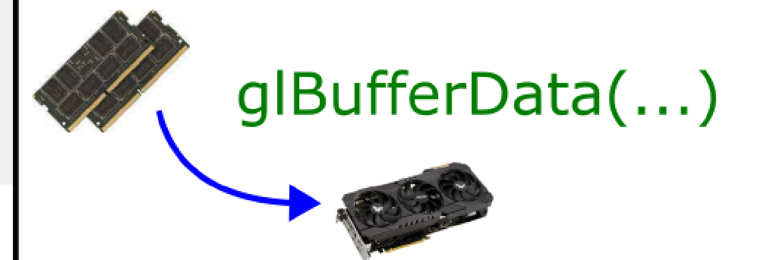
scene.hpp

```
mesh_drawable drawable
...
```

scene.cpp

```
initialize()
    mesh m;
    m.position = { ... }
    /or  m = mesh_primitive_...(...);
    drawable.initialize_data_on_gpu(m);

display_frame()
    drawable.model.position = ...

    draw(drawable, environment);
```

```
struct mesh
    numarray<vec3> position
    numarray<vec3> normal
    numarray<vec3> color
    numarray<vec2> uv

    numarray<int3> connectivity
```



glUseProgram(...)
glUniform(...)
glDraw(...)

Maillage - Attribus de sommets

En pratique, chaque sommet peut être attaché à des attributs supplémentaires

- Normales, coordonnées de textures, couleurs, etc.

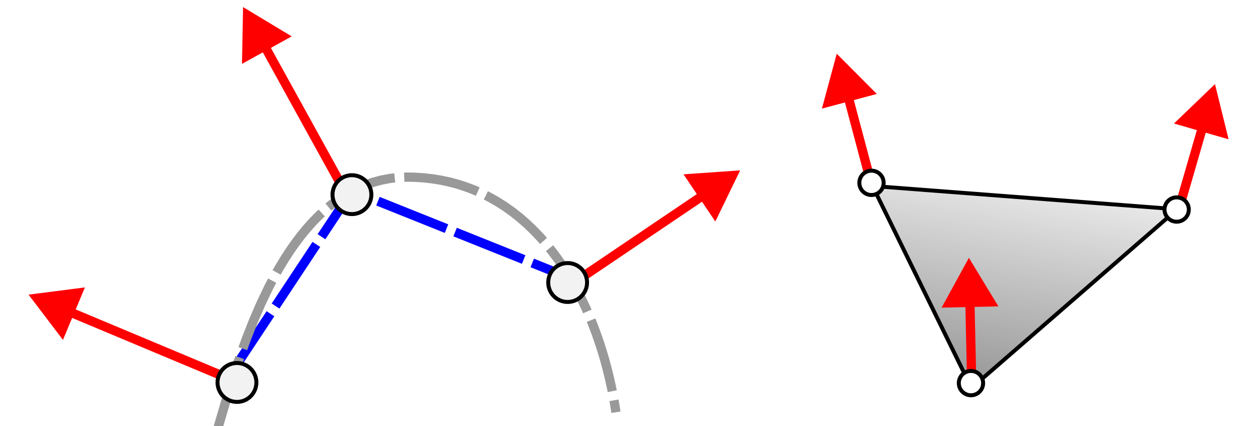
Exemple:

```
std::vector<vec3> position = { p_0, p_1, p_2, p_3 };
std::vector<vec3> normal   = { n_0, n_1, n_2, n_3 };
std::vector<vec3> color    = { c_0, c_1, c_2, c_3 };
std::vector<vec2> uv       = { uv_0, uv_1, uv_2, uv_3 };
std::vector<int3> connectivity = { {0,1,2}, {0,1,3}, {0,2,3}, {1,2,3} };
```

$p_i = (x_i, y_i, z_i), n_i = (n_{xi}, n_{yi}, n_{zi})$ avec $\|n_i\| = 1, c_i = (r_i, g_i, b_i)$

Remarque:

- Les normales sont définies par sommet
  - *Permet une apparence lisse, interpolation d'illumination.*
- Une seule connectivité, partagée par les attributs, pour un affichage GPU.

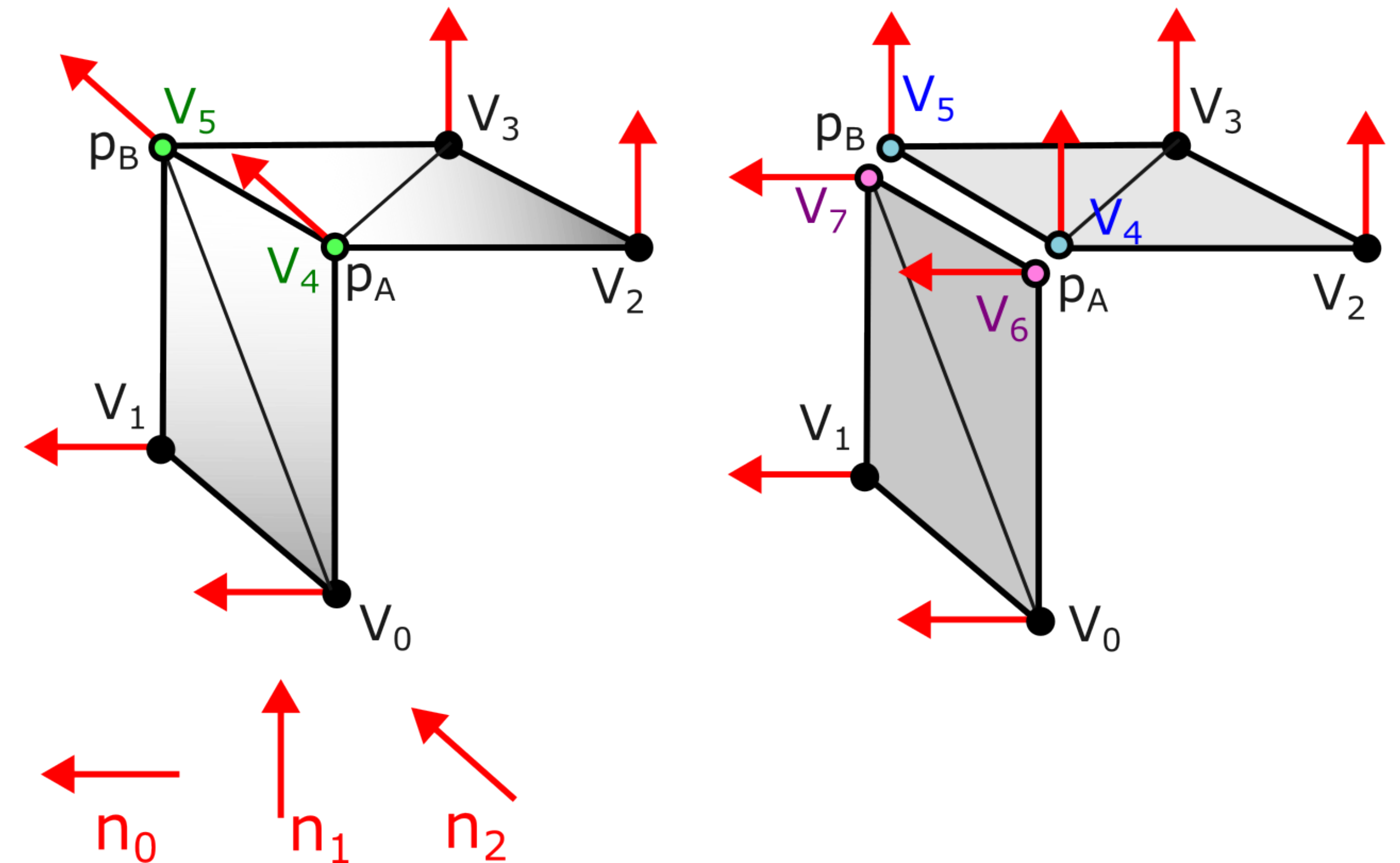


Attribus de sommets - Duplication de sommets

Il peut être nécessaire de "dupliquer" des sommets

Même position, mais attributs différents

Ex. Bords franc (/Sharp corner)



Position p_A avec un seul sommet v_4 sur le coin, et une normale n_1

Position p_B avec un seul sommet v_5 sur le coin, et une normale n_1

```
std::vector<vec3> position = {p0,p1,p2,p3,pA,pB};
std::vector<vec3> normal   = {n0,n0,n1,n1,n2,n2};
std::vector<int3> connectivity = { {0,5,1}, {0,4,5}, {4,2,3}, {4,3,5} };
```

Duplication des sommets:

Position p_A est associée à 2 sommets: v_4 et v_6, mais différentes normales n_1, n_0

Position p_B est associée à 2 sommets: v_5 et v_7, mais différentes normales n_1, n_0

```
std::vector<vec3> position = {p0,p1,p2,p3,pA,pB,pA,pB};
std::vector<vec3> normal   = {n0,n0,n1,n1,n1,n1,n0,n0};
std::vector<int3> connectivity = { {0,7,1}, {0,6,7}, {4,2,3}, {4,3,5} };
```

Rem. Pour une cube: on duplique chaque sommet 3 fois.

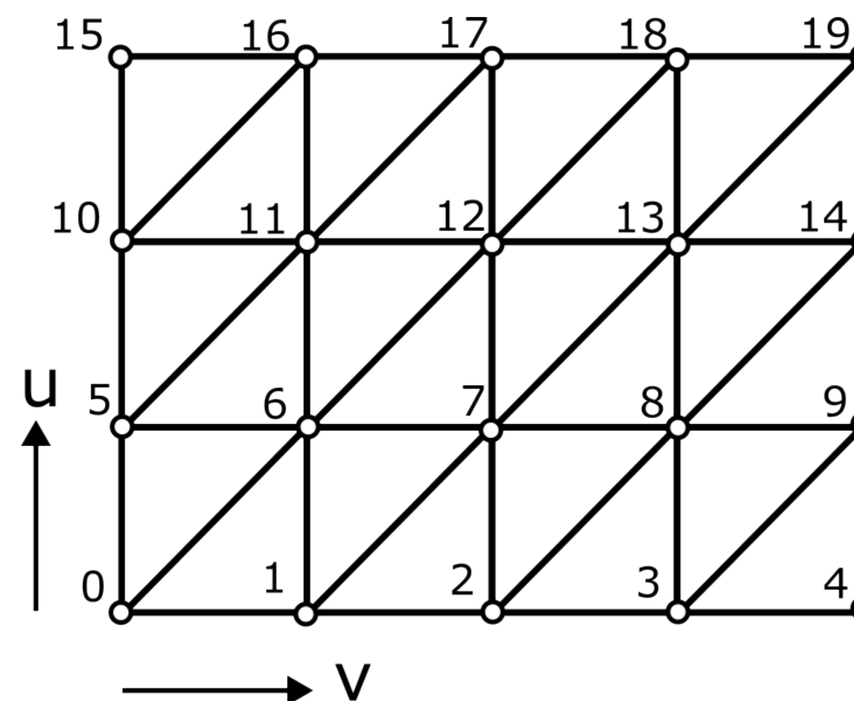
Structure en grille

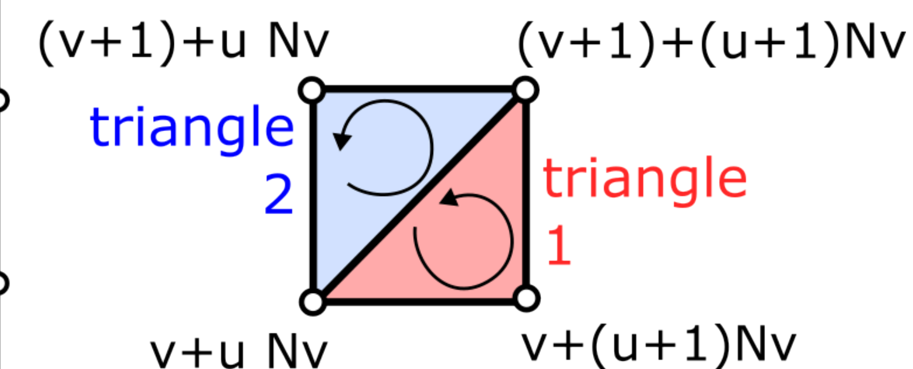
Example typique: Surface paramétrique $S(u,v) = (S_x(u,v), S_y(u,v), S_z(u,v))$

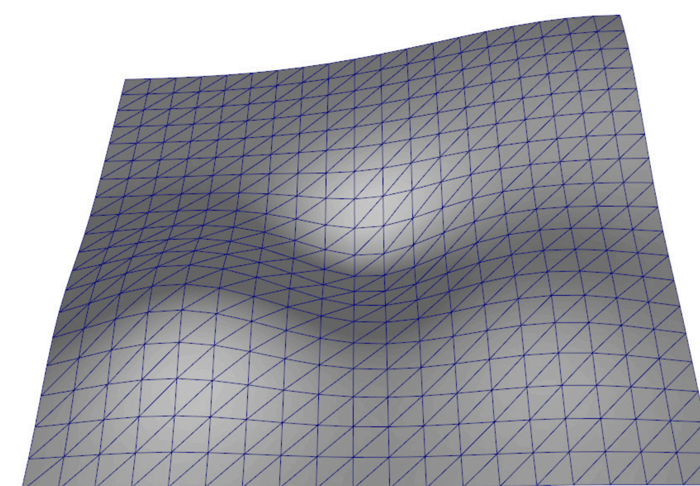
$(u,v) \in [0,1]^2$ (ou tout autre domaine rectangulaire)

Echantillonnée uniformément suivant $N_u \times N_v$ points

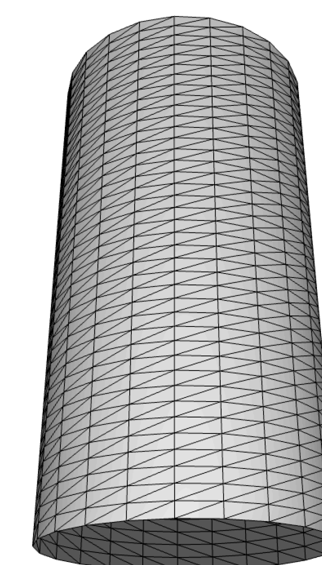
```
// Vertex position
for(int ku=0; ku<Nu; ku++) {
    for(int kv=0; kv<Nv; kv++) {
        // Parametric coordinates (u,v) \in [0,1]
        float u = ku/(Nu-1.0f);
        float v = kv/(Nv-1.0f);

        S.position[kv+Nv*ku] = {Sx(u,v),Sy(u,v),Sz(u,v)};
    }
}

// Connectivity
for(int ku=0; ku<Nu-1; ku++) {
    for(int kv=0; kv<Nv-1; kv++) {
        unsigned int idx = kv + Nv*ku; // offset

        uint3 triangle_1 = {idx, idx+1+Nv, idx+1};
        uint3 triangle_2 = {idx, idx+N, idx+1+Nv};

        S.connectivity.push_back(triangle_1);
        S.connectivity.push_back(triangle_2);
    }
}
```







Champ de hauteur/height field $(u, v, z(u,v))$



Cylindre paramétrique

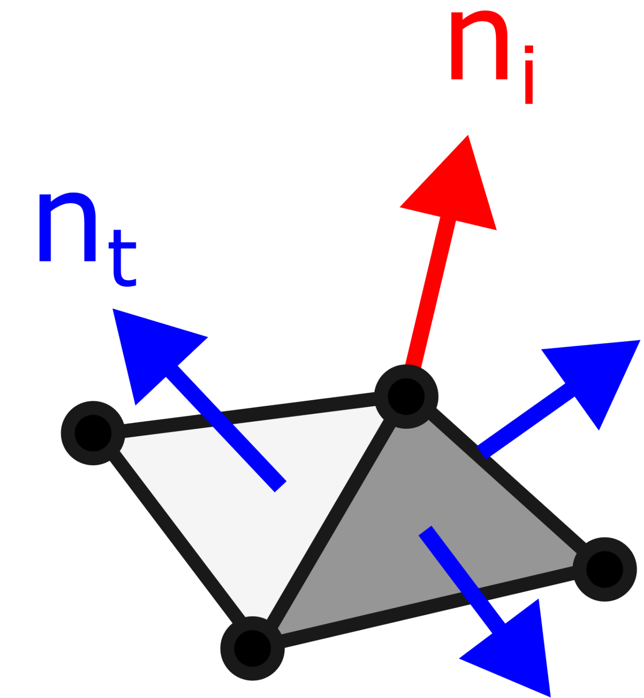
Normales des maillages

Les normales ne sont pas toujours données (ex. lecture de fichiers, maillages procéduraux sans normales, etc.)

Parfois seul les positions sont fournies en entrées.

$\Rightarrow$ possibilité d'approximer des normales à partir de la géométrie du maillage.

$\Rightarrow$ utile lors de la déformation de maillages.



Ex. Moyenne (non pondérée) des normales des triangles voisins d'un sommet:

1- Calcul des normales n_t des triangles voisin d'un sommet i, avec $t \in \mathcal{N}_i$

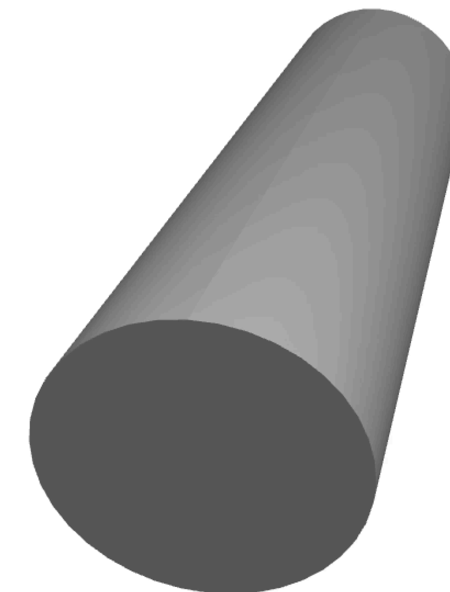
$\mathcal{N}_i$ est le 1-voisinage (1-ring) du sommet i.

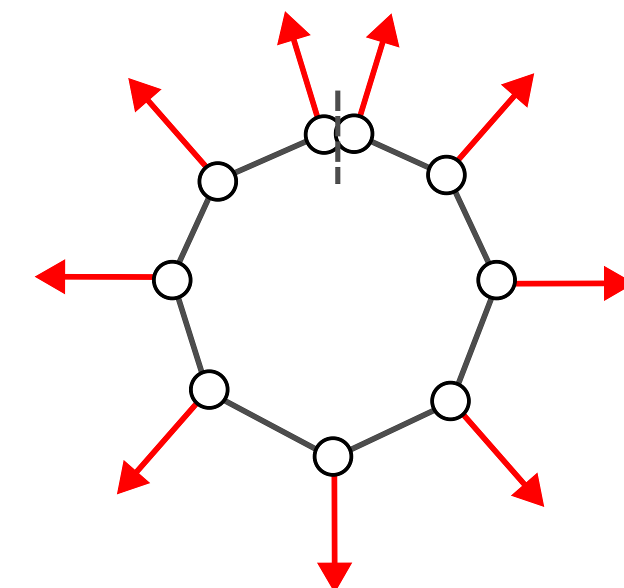
2- Normale du sommet n_i = moyenne normalisée des triangles voisins

$$n_i = \sum_{t \in \mathcal{N}_i} n_t$$

$$n_i \leftarrow n_i / \|n_i\|$$

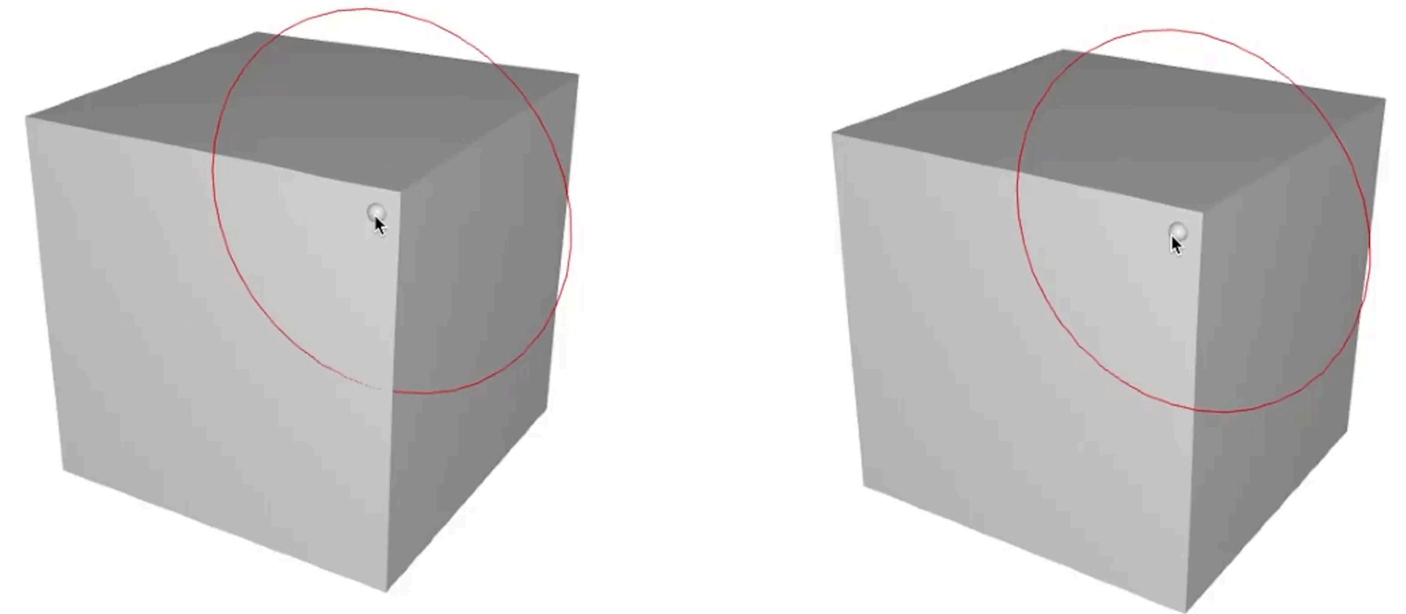
Rem. si les sommets sont dupliqués, les normales peuvent être différentes en fonction de leur connectivité.





Normales des maillages

```
std::vector<vec3> normal = position.size();

// Summation per triangle
for(int t=0; t<connectivity.size(); t++) {
    vec3 a = connectivity[t][0];
    vec3 b = connectivity[t][1];
    vec3 c = connectivity[t][2];

    vec3 n = cross(position[b]-position[a], position[c]-position[a]);
    n = normalize(n); // beware of degenerate triangles

    normal[connectivity[t][a]] += n;
    normal[connectivity[t][b]] += n;
    normal[connectivity[t][c]] += n;
}

// Final normalization
for(int k=0; k<normal.size(); k++) {
    normal[k] = normalize(normal[k]);
}
```



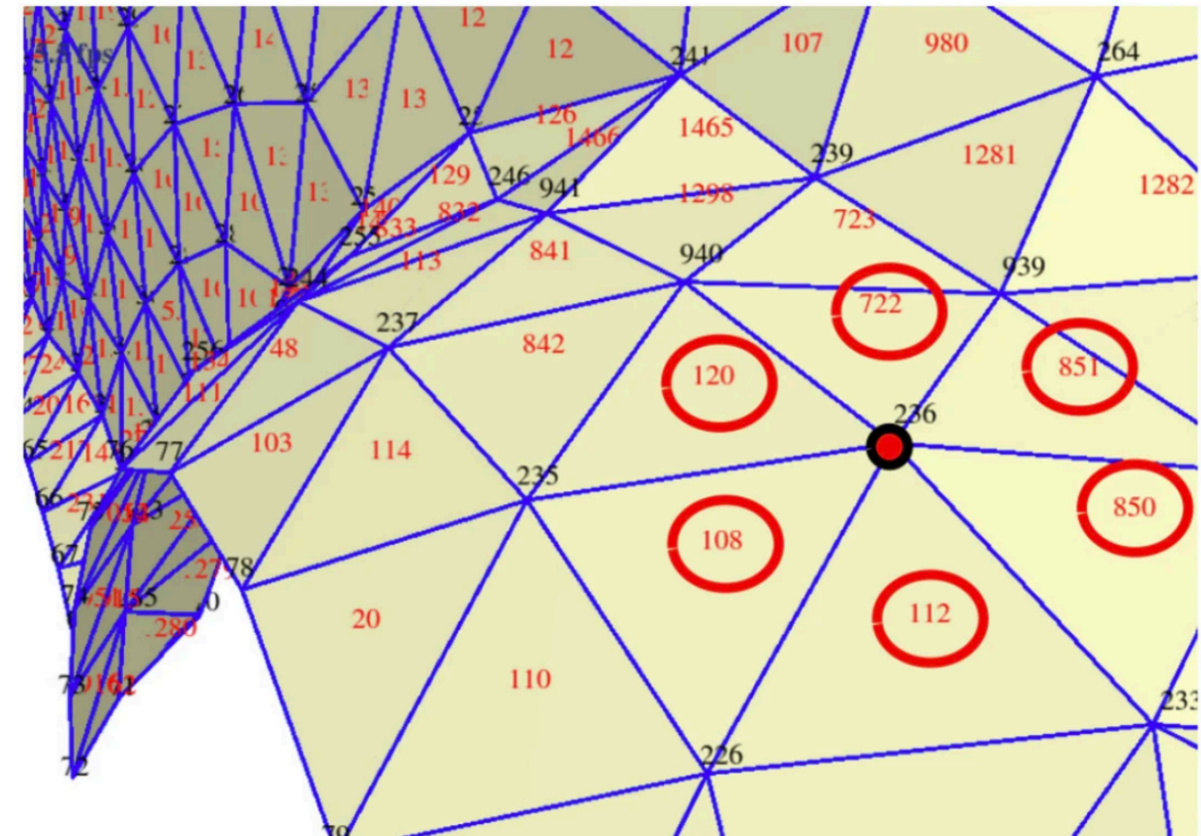
- *Left: Initial normals*
- *Right: Normal updated from the geometry*

1-voisinage, 1-ring

Structure de 1-voisinage, peut être utile à stocker

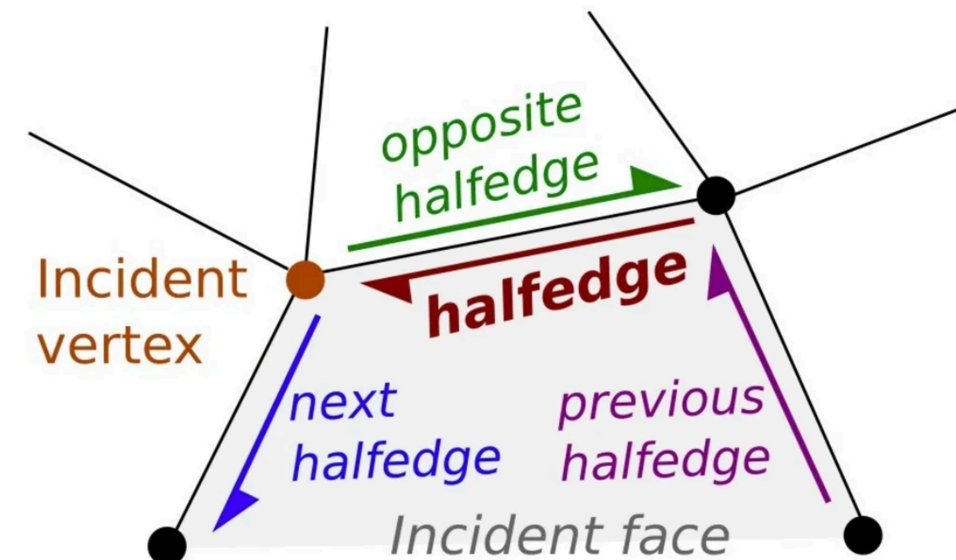
```
std::vector<std::vector<int> > one_ring;
one_ring.resize(position.size());
for(int k_tri=0; k_tri<connectivity.size(); k_tri++) {
    for(int k=0; k<3; k++) {
        one_ring[connectivity[k_tri][k]].push_back(k_tri);
    }
}
```

```
one_ring[235] = [842,120,108,110,20,114]
one_ring[236] = [108,112,851,850,120,722]
```



Structure en demi-arête (half-edge)

- Encode les arêtes successives
  - Arêtes sont orientées, chaque arête a un opposé
- Faces se retrouvent comme une boucle le long d'arêtes



Halfedge

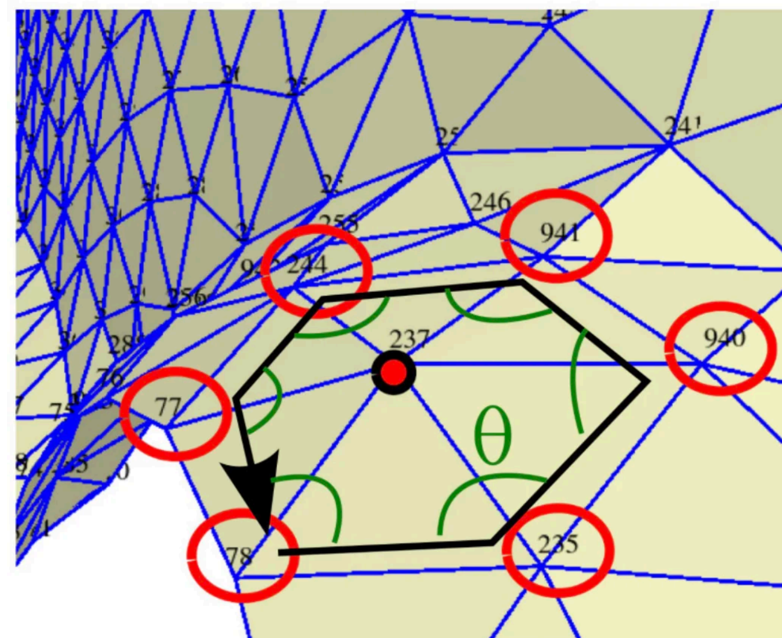
```
Halfedge opposite();
Halfedge next();
Halfedge prev();

Vertex vertex();
Face face();
```

Intérêt:

- Ajout/suppression en O(1) (split, collapse)
- Calculs d'angles de coins (Cotangent weight)

Limites:

- Limité aux 2-variétés
- Non-contigues en mémoire




```
typedef CGAL::Cartesian<double> Kernel;
typedef CGAL::Polyhedron_3<Kernel> Polyhedron;

int main() {
    Polyhedron mesh;
    std::ifstream stream("mesh.off");
    stream >> mesh;

    auto it_face = mesh.facets_begin();
    auto it_face_end = mesh.facets_end();

    int face_number=0;
    for(;it_face!=it_face_end;++it_face){
        std::cout << "Face " << face_number << std::endl;
        auto halfedge = it_face->halfedge();
        auto const halfedge_end = halfedge;
        do{
            const auto p = halfedge->vertex()->point();
            std::cout << p << std::endl;
            halfedge = halfedge->next();
        }while(halfedge != halfedge_end);

        face_number++;
    }
}
```

Représentation et modélisation de surfaces

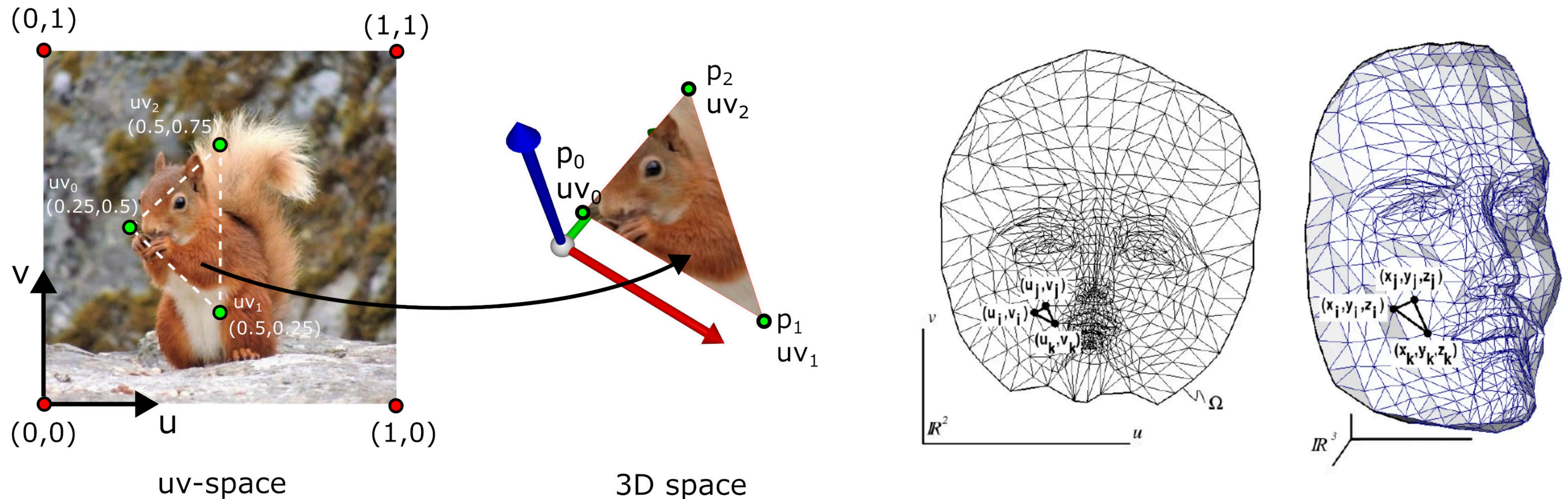
- Théorie des surfaces lisses :
- Courbures
- Maillages
- **Textures uv**
- Surfaces Splines
- Surface de subdivision
- Ensemble de points

Textures

Objectif: Niveau de détail (couleur) plus petit que la géométrie/triangle

Cas classique: **UV-mapping** = Image 2D plaquée sur une surface 3D

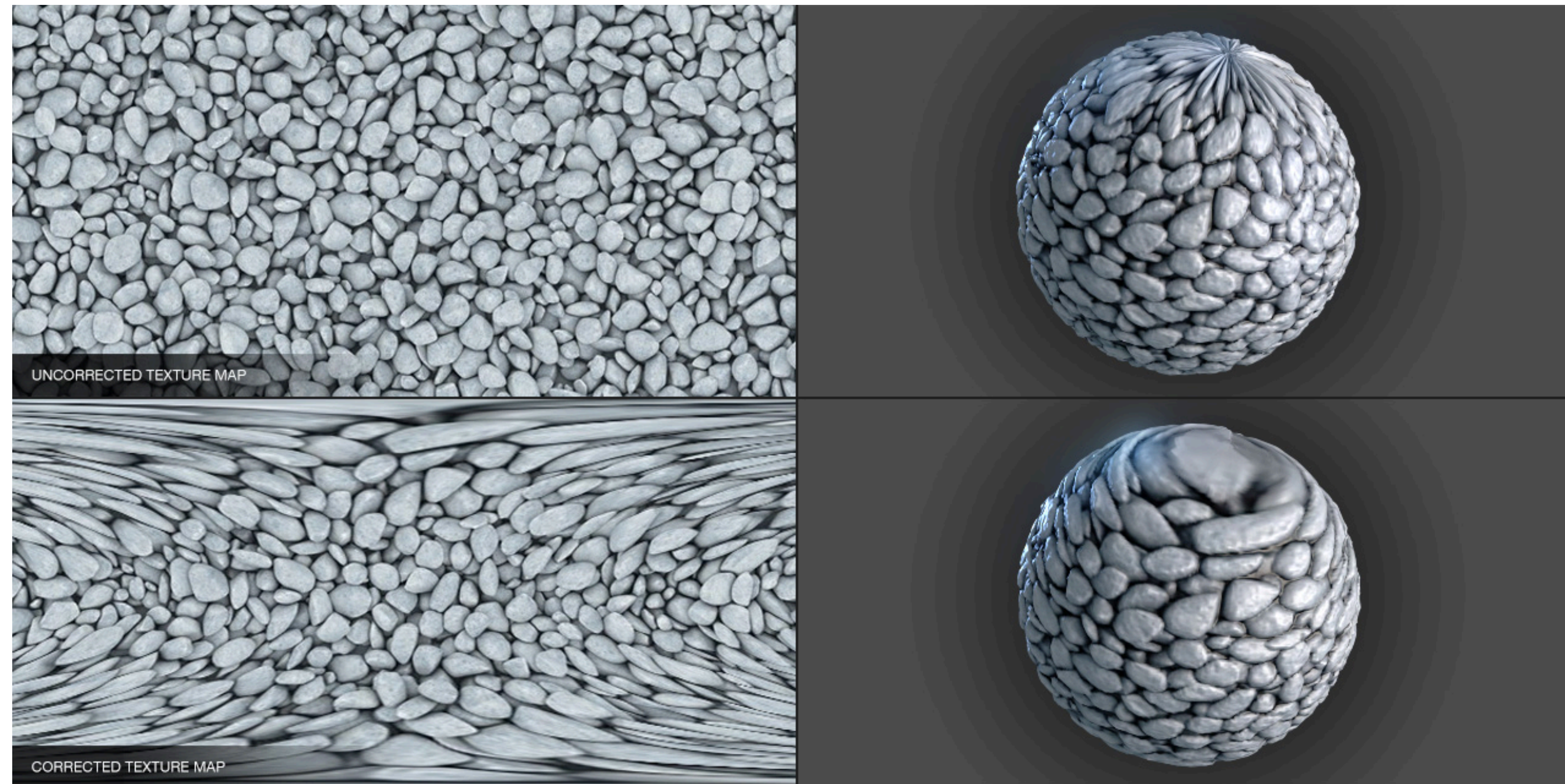
- Texture: L'image 2D.
- Coordonnées de texture (uv/rs): Coordonnées 2D d'un sommet dans l'image.
  - Associe à chaque sommet v_i une coordonnées uv_i
  - Au rendu (fragment shader): Affiche la couleur de l'image à la coordonnée interpolée

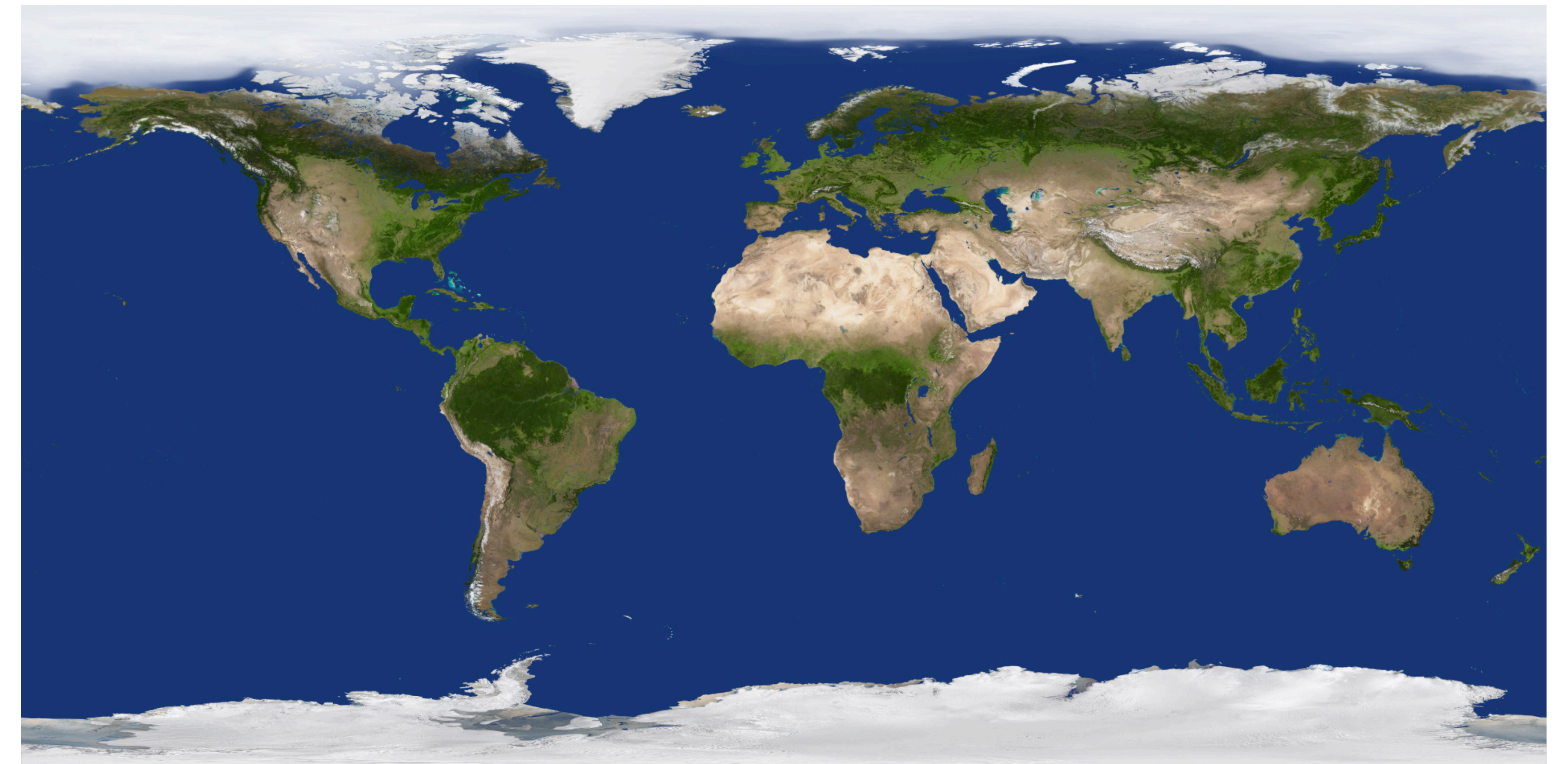


Textures - uv-mapping

UV-mapping: Meilleur placage d'une image sur une surface 3D

- Généralement associé à des déformation de longueurs et d'angles



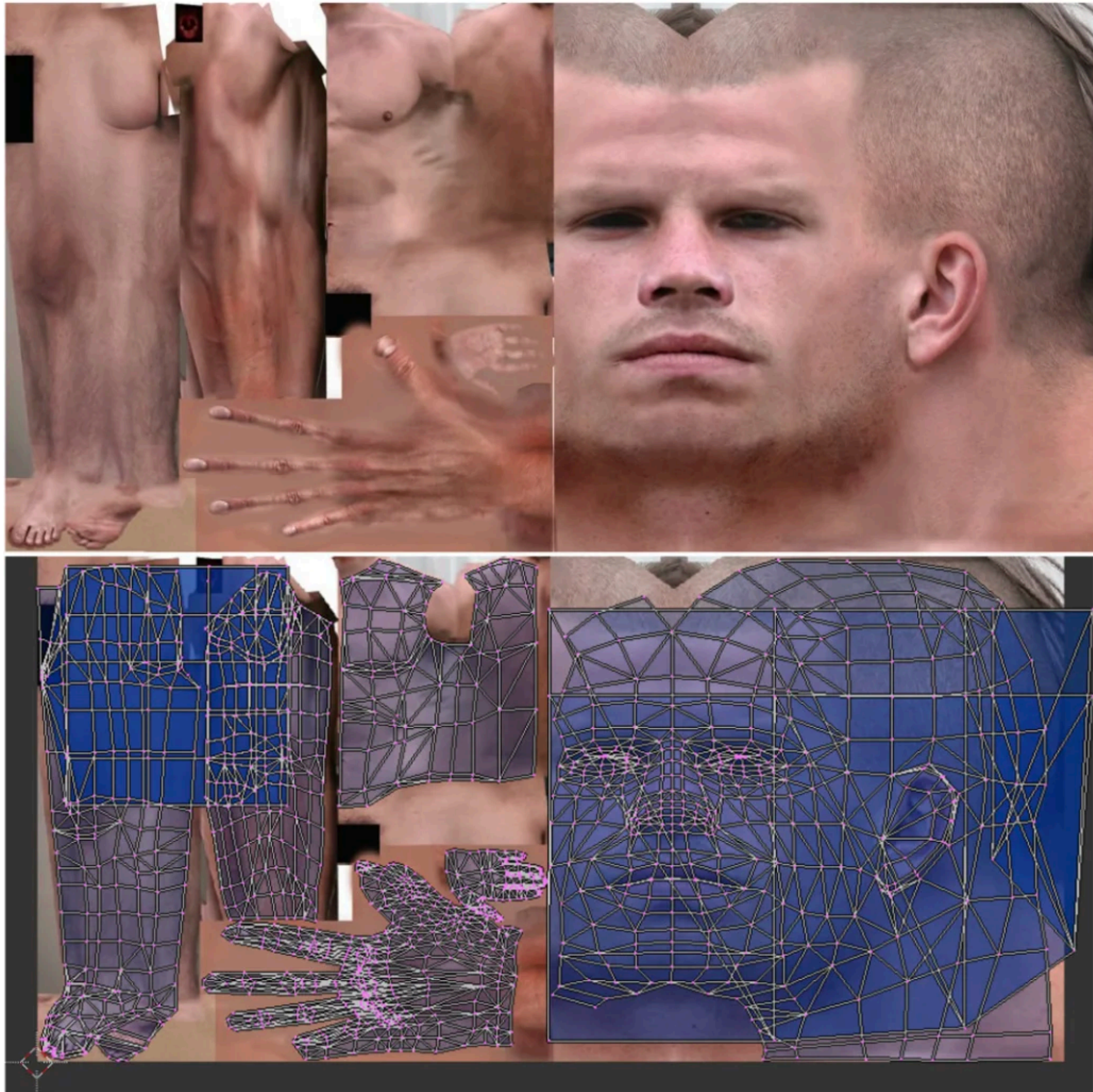


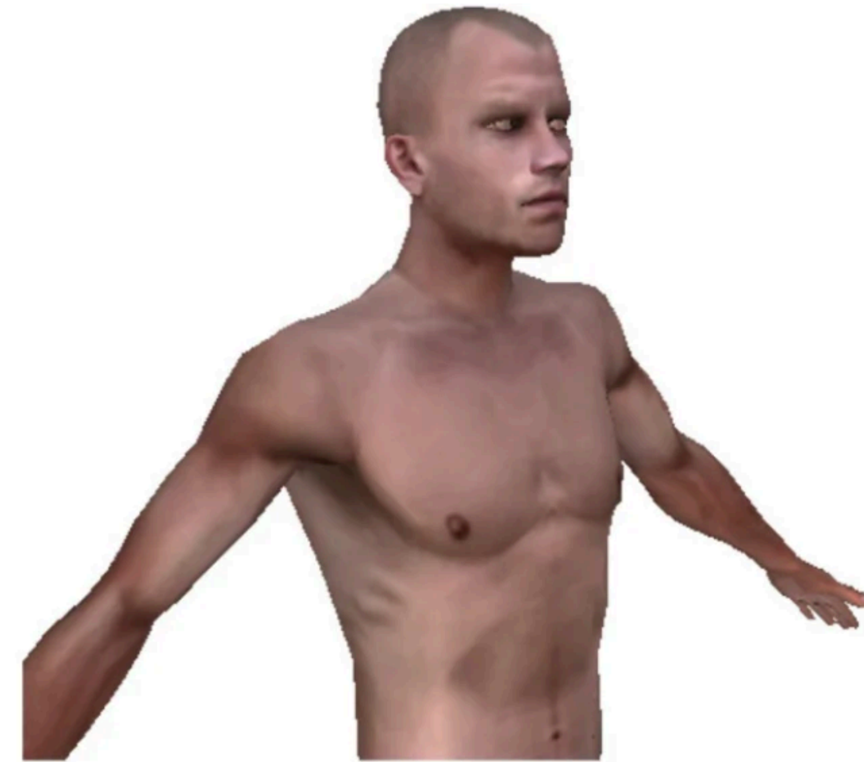
Rem. Impossible de plaquer une image plane sur une sphère sans déformation de longueurs

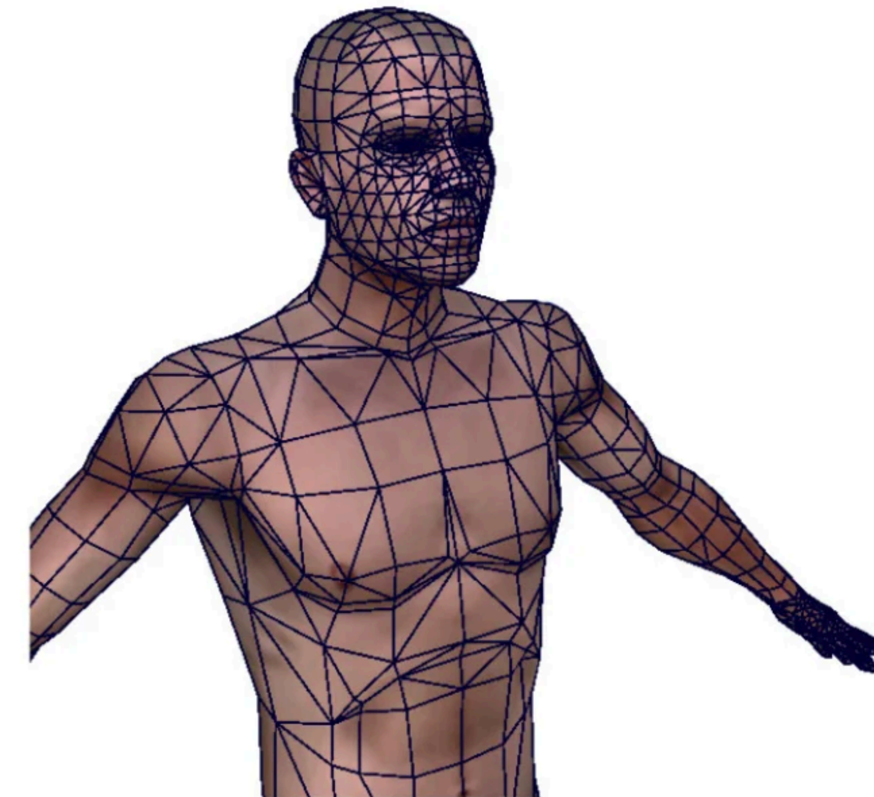
Uniquement possible avec surfaces développables (courbure de Gauss nulle).

Textures - cas réel

Définition par morceaux (patchs), pouvant se recouvrir





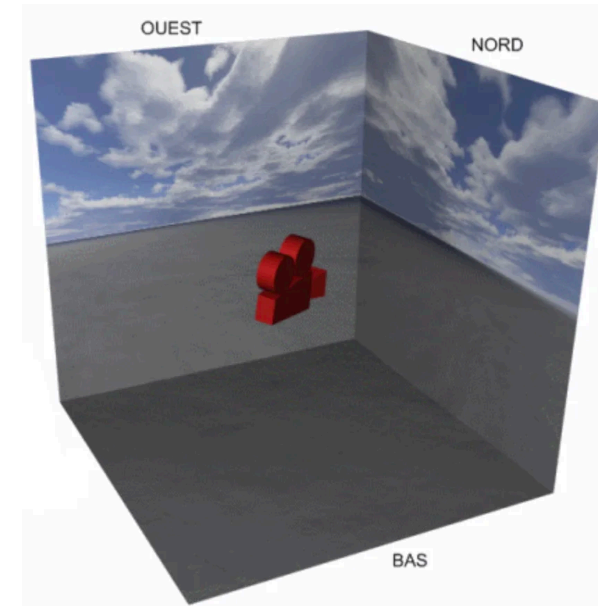


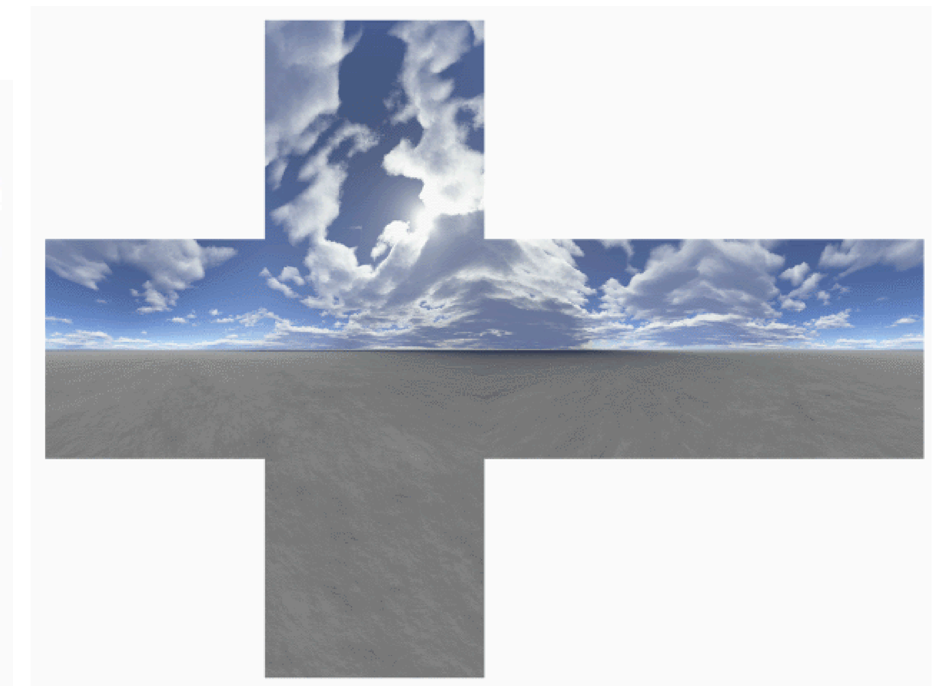
Effets: Skybox

Boite texturée d'un environement

Toujours centré sur la position de la caméra.

  Impression de paysage à l'infini.



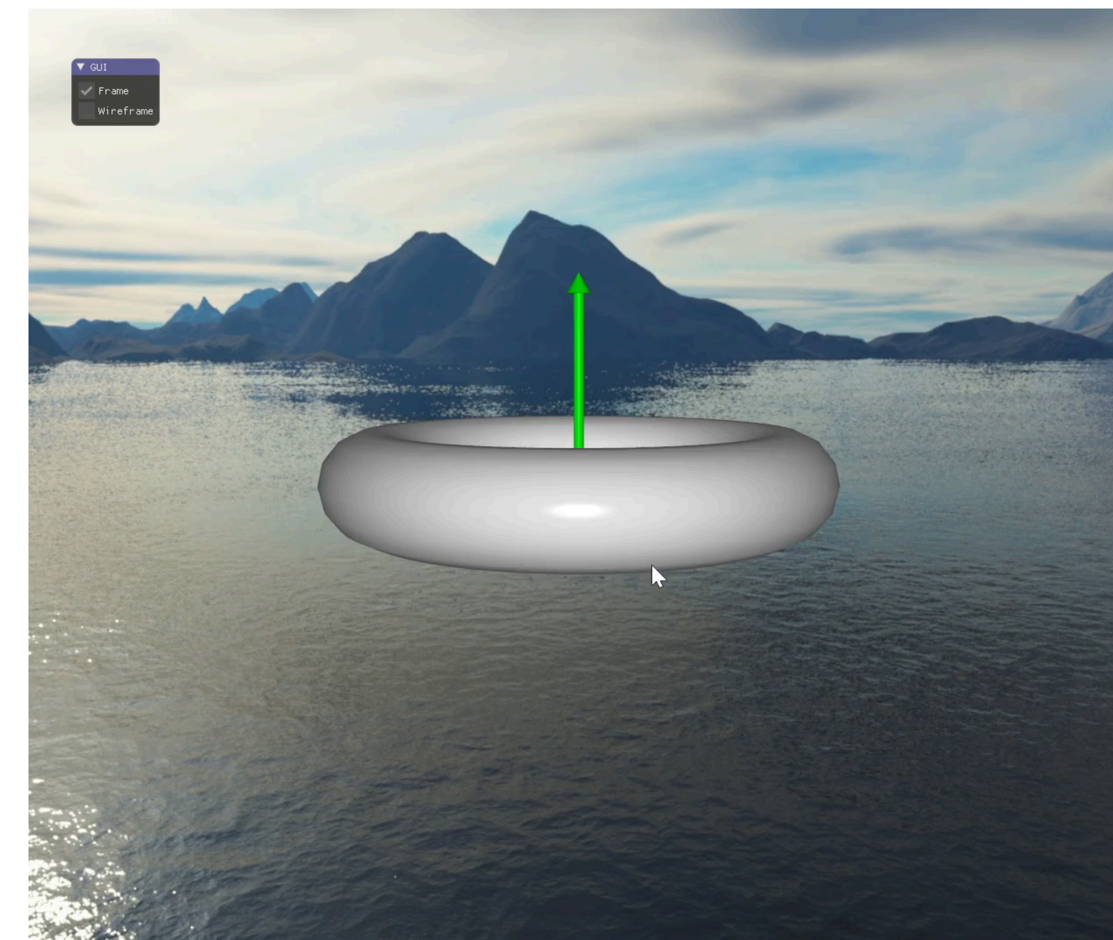


Principe

Afficher la skybox en premier

Afficher tous les autres objets ensuite, toujours au dessus

```
display(){
    glDepthMask( GL_FALSE ); // disable depth-buffer writing
    // draw skybox, ex. draw(skybox, environment);

    glDepthMask( GL_TRUE ); // re-activate depth-buffer writing
    // draw other objects
    // ...
}
```

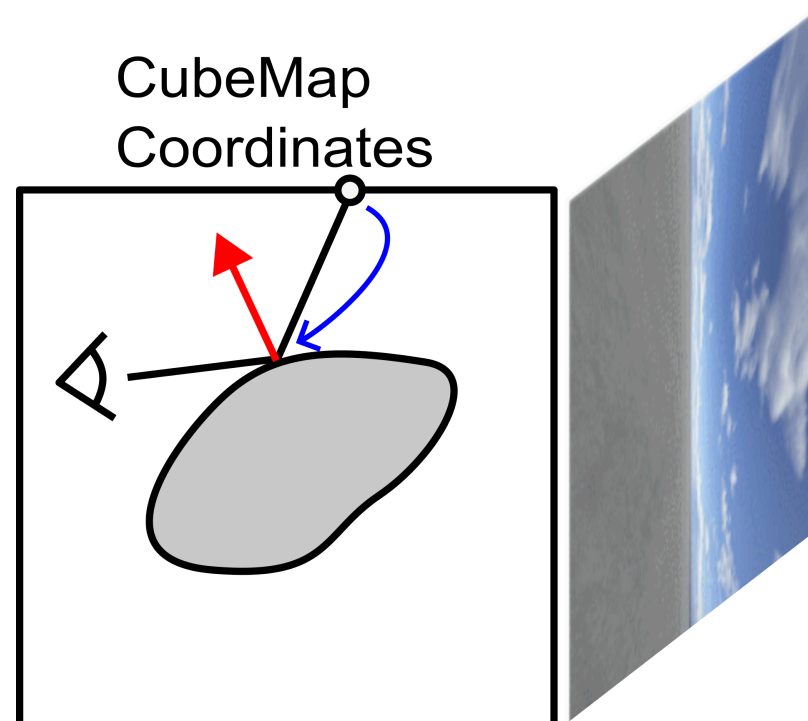


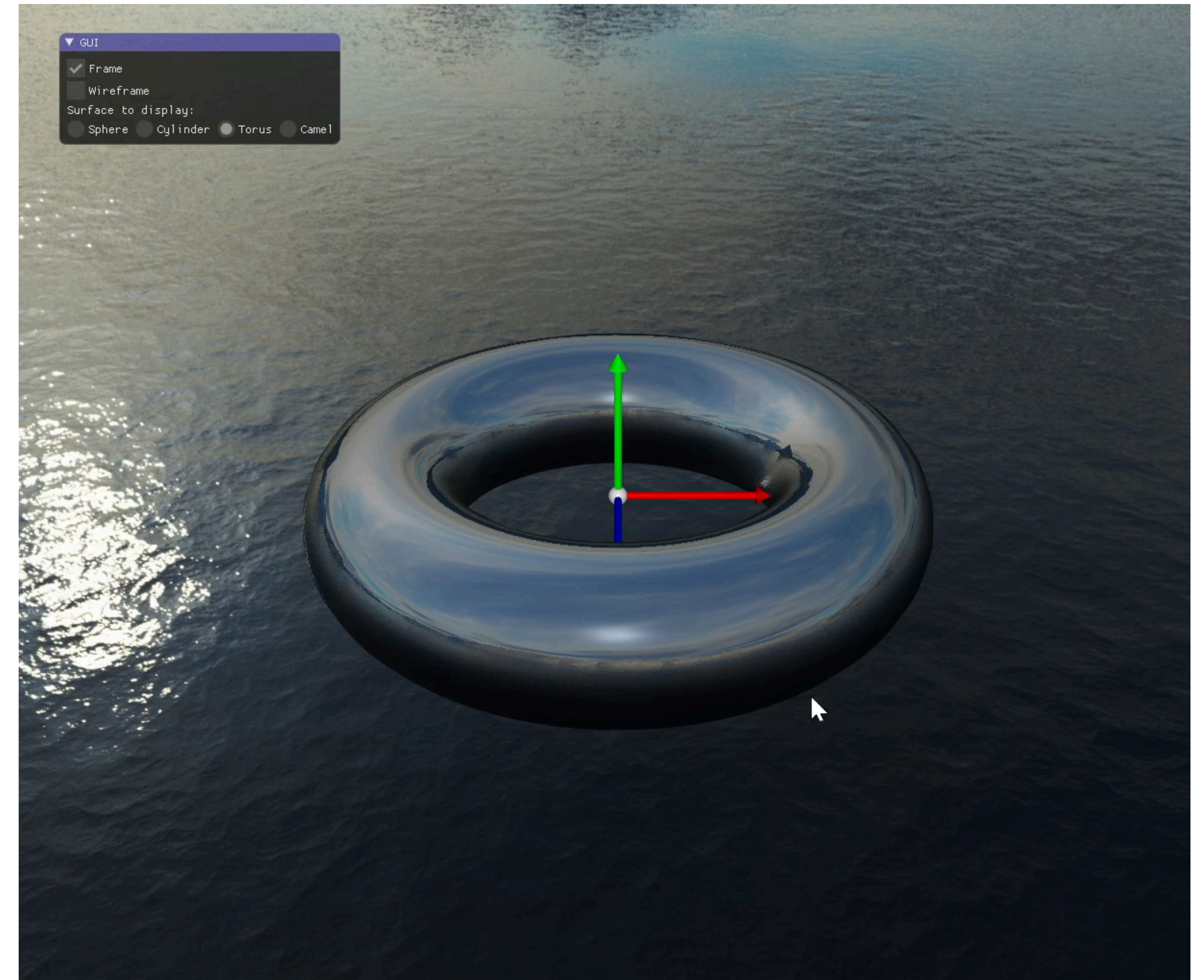
Effets: Environment Mapping

Représenter une reflexion d'une skybox sur une surface

Impression de reflet de l'environnement sur la surface

```
// Get the color of the environment map
vec3 V = normalize(camera_position - fragment.position);
vec3 R_skybox = reflect(-V, N);
vec4 color_environment_map = texture(image_skybox, skybox_rotation);
```



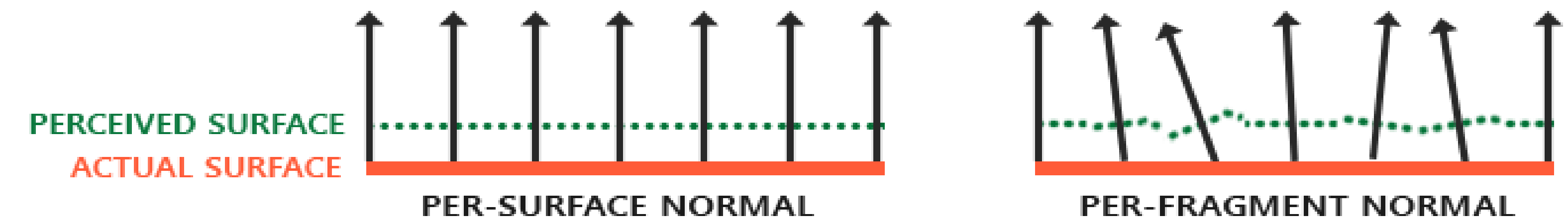


Demo code

Effets: Normal map

Déformer les normales: simule des interactions lumineuses avec des détails plus fins qu'un triangle

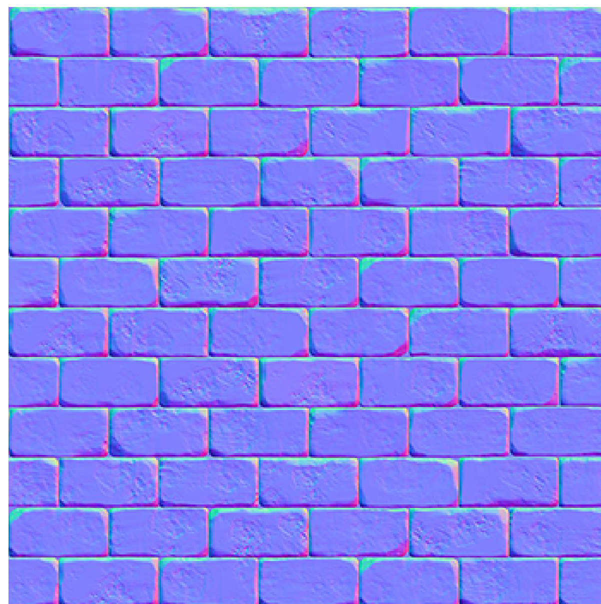
- La géométrie des triangles reste inchangée

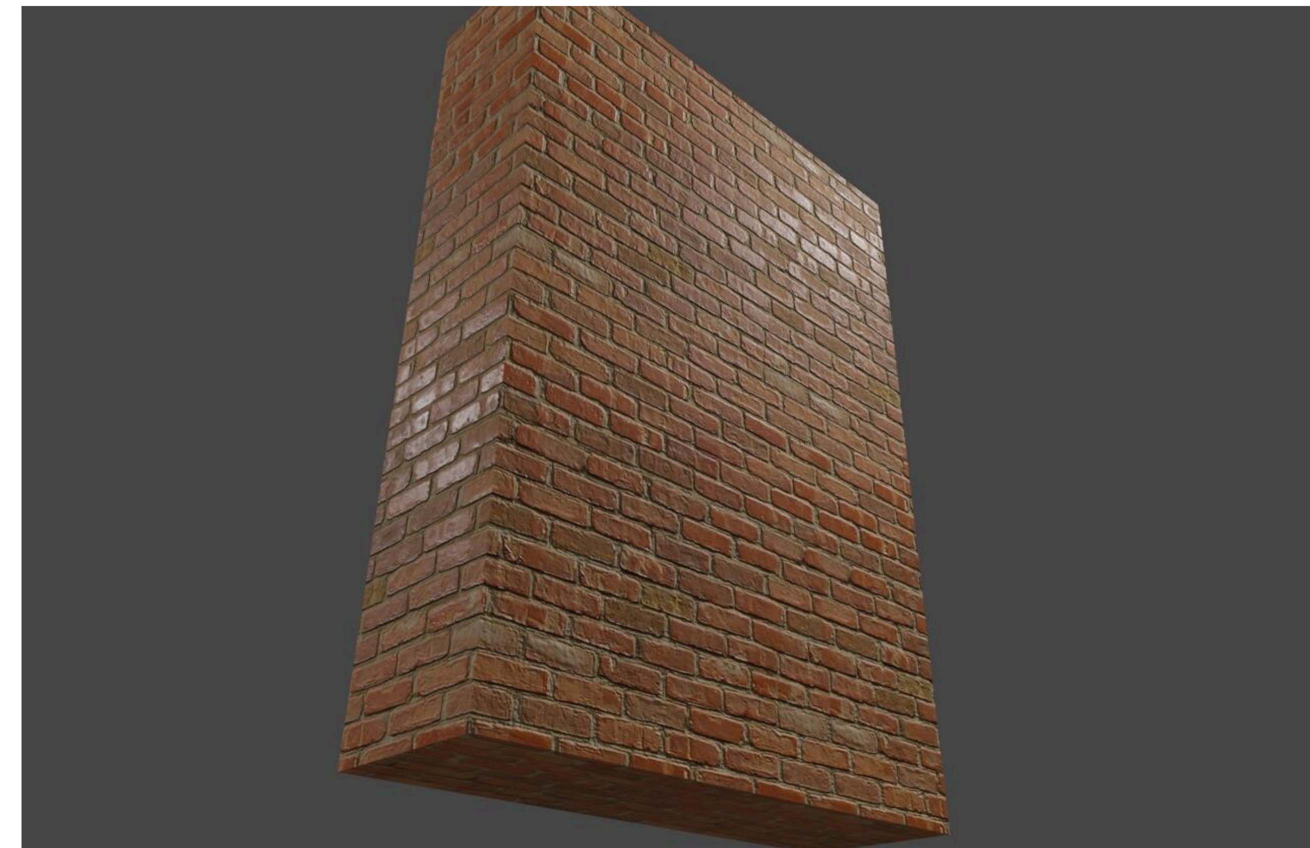


Normale encodée en (r, g, b) dans l'espace tangent

- $n = r\,\mathbf{T} + g\,\mathbf{B} + b\,\mathbf{N}$
- $(\mathbf{T}, \mathbf{B}, \mathbf{N})$: Repère: Tangente, Binormale, Normale définie par les coordonnées (u, v).

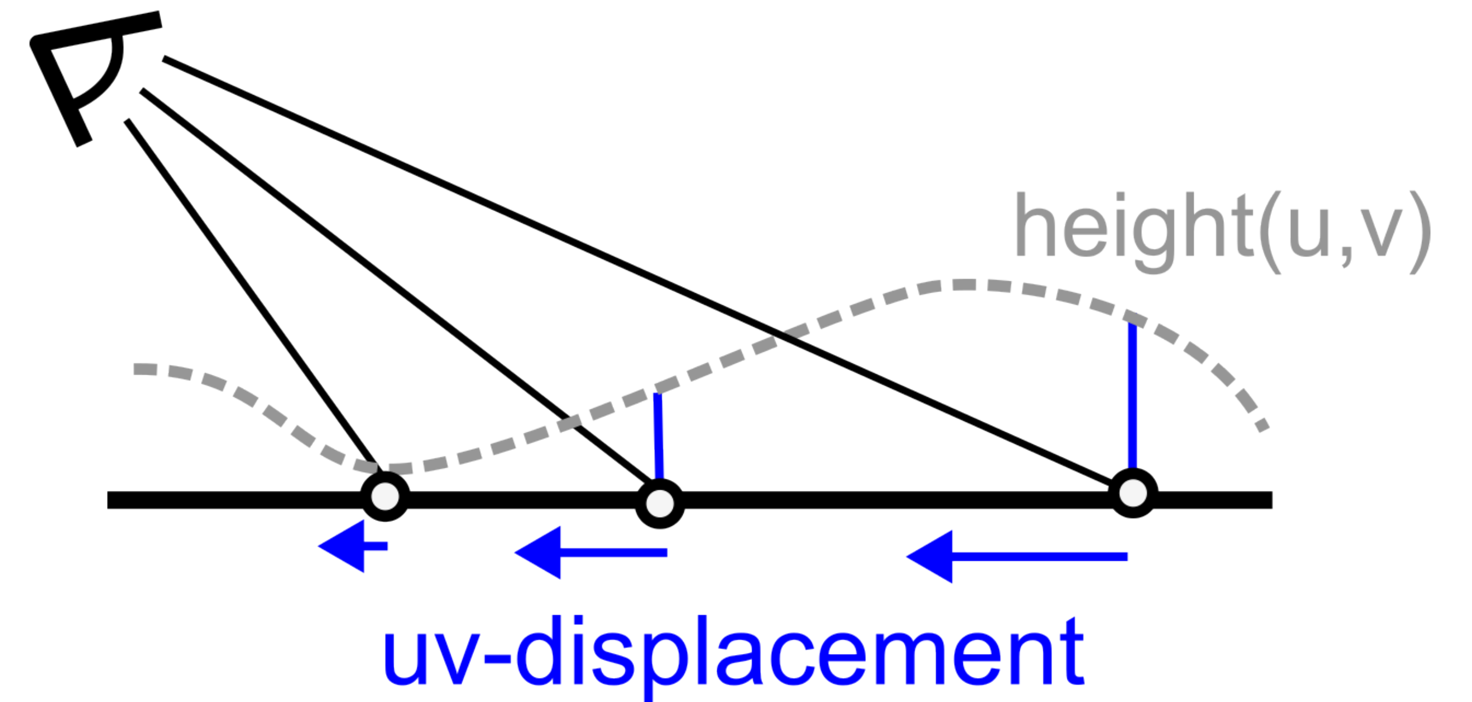
$$\begin{cases} p_0 - p_1 = (u_0 - u_1)\mathbf{T} + (v_0 - v_1)\mathbf{B} \\ p_2 - p_1 = (u_2 - u_1)\mathbf{T} + (v_2 - v_1)\mathbf{B} \\ \mathbf{N} = \mathbf{T} \times \mathbf{B} \end{cases}$$





Effets: Parallax Mapping

- Déformer les coordonnées de normales: impression de relief
  - Application aux champs de hauteurs / height-field
  - Déplacement proportionnel à la hauteur



Your graphics card does not seem to support WebGL.
Find out how to get it here.

[Demo code] [Source github]