

Maillages

- **Géométrie**
- Texture UV

Maillage - structure

Maillage: Ensemble de polygones partageant des sommets

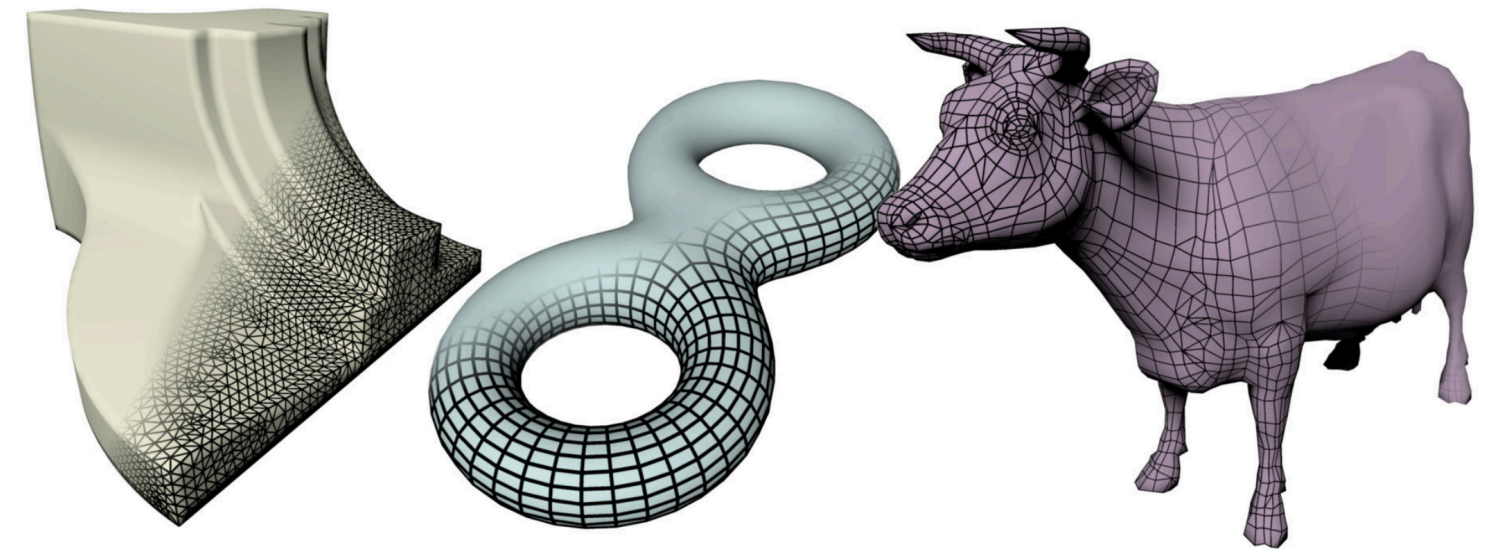
N_f faces, N_v sommets (/vertices), N_e arêtes (/edges)

Triangulation: Toutes les faces sont des triangles

Quad mesh: Toutes les faces sont des quadrangles

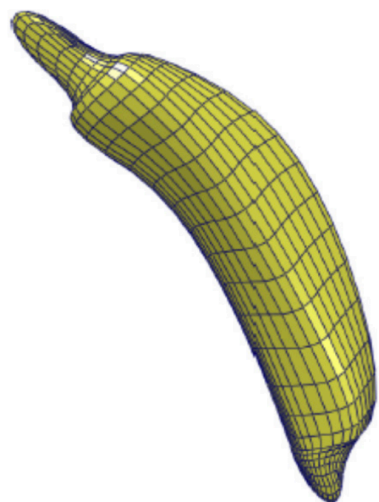
Idéalement quadrangle planaires.

Poly mesh: Les faces sont des polygones quelconques

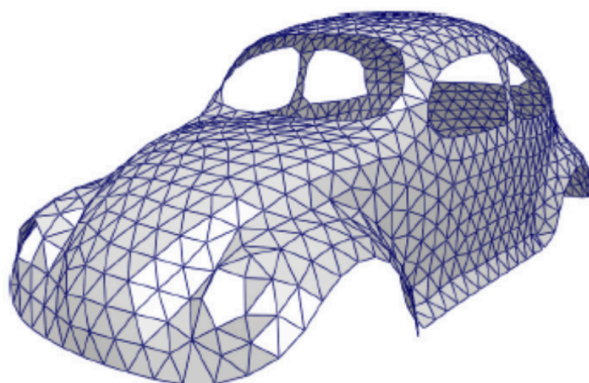


Rem. Maillage représente une variété si chaque arête est partagée par 2 faces au plus.

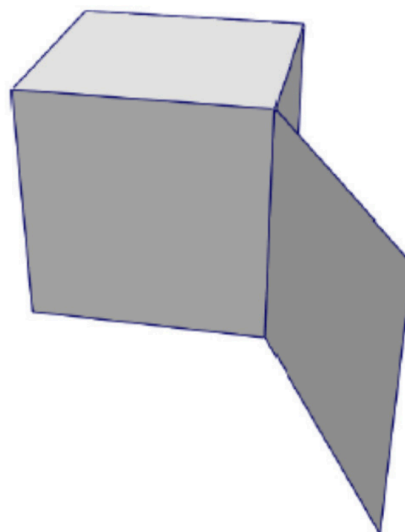
2-manifold



*2-manifold
with boundaries*



not a 2-manifold



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.

Exemples

Tétraèdre

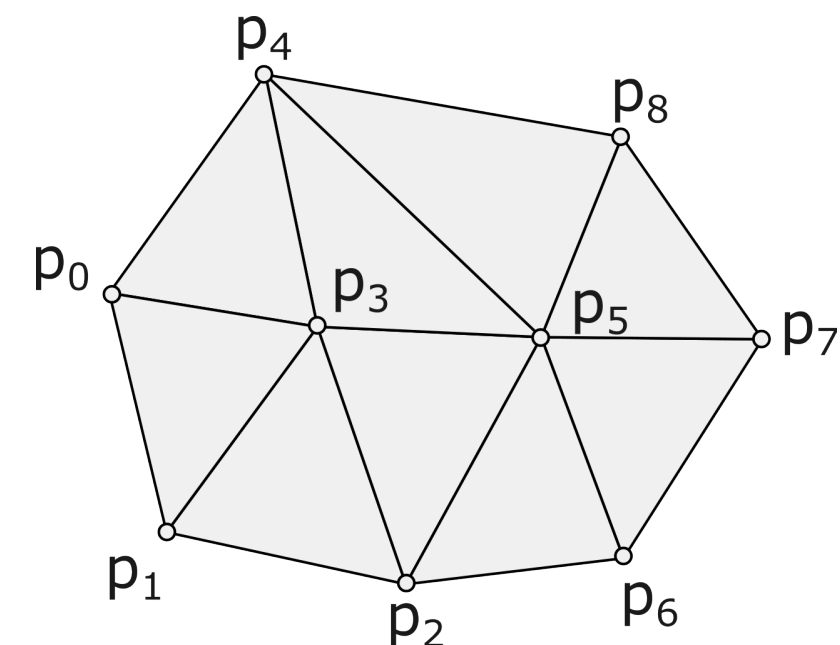
```
std::vector<vec3> position = { {0,0,0}, {1,0,0}, {0,1,0}, {0,0,1} };  
std::vector<int3> connectivity = { {0,1,2}, {0,1,3}, {0,2,3}, {1,2,3} };
```

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 `environment`)
- Appel à `glDraw()`

scene.hpp

```
mesh_drawable drawable  
...
```

scene.cpp

initialize()

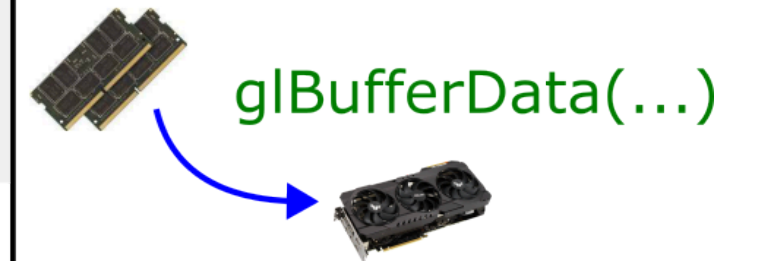
```
mesh m;  
m.position = { ... }  
/or m = mesh_primitive_...(...);  
drawable.initialize_data_on_gpu(m);
```

display_frame()

```
drawable.model.translation = ...  
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} };
```

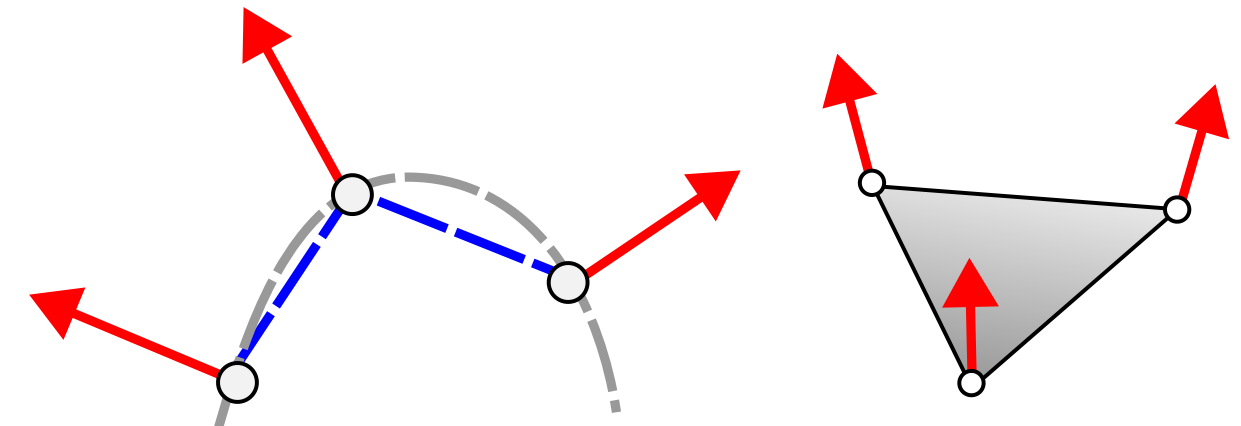
$p_i = (x_i, y_i, z_i)$, $n_i = (n_{xi}, n_{yi}, n_{zi})$ avec $\|n_i\| = 1$, $c_i = (r_i, g_i, b_i)$

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} };
```

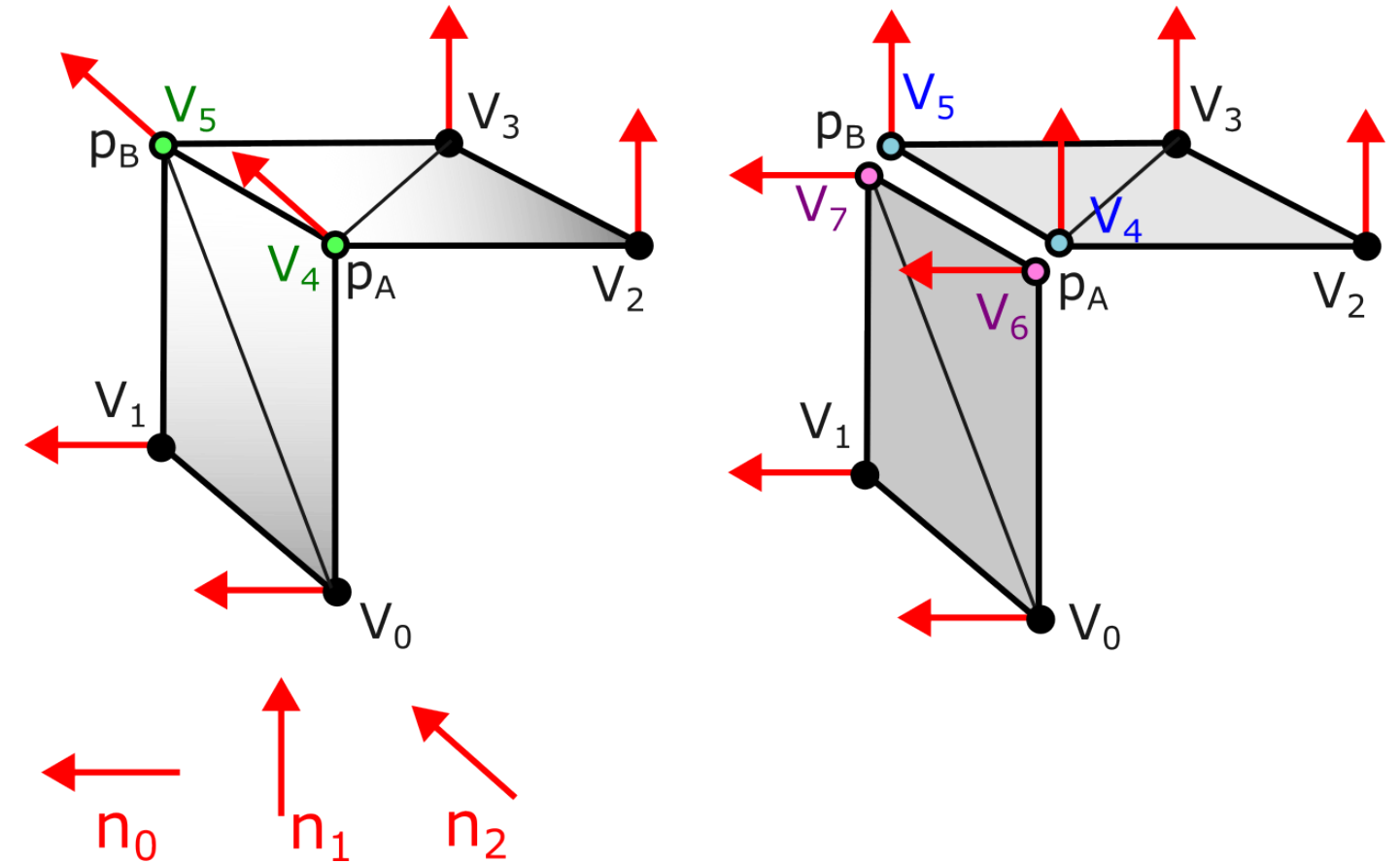
Duplication des sommets:

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

Exemple 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

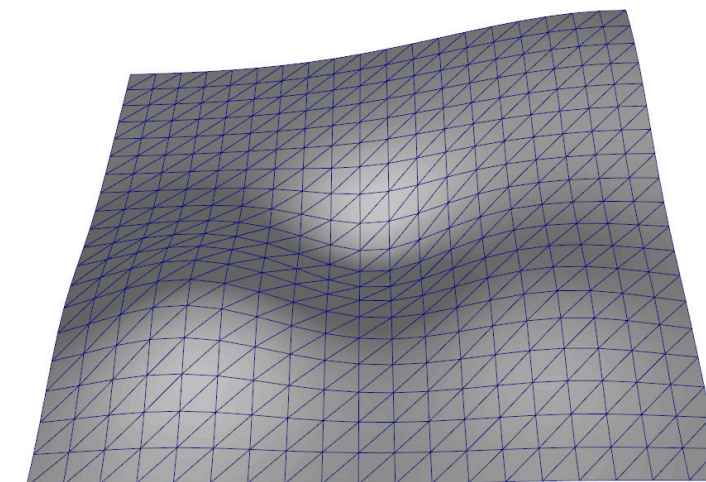
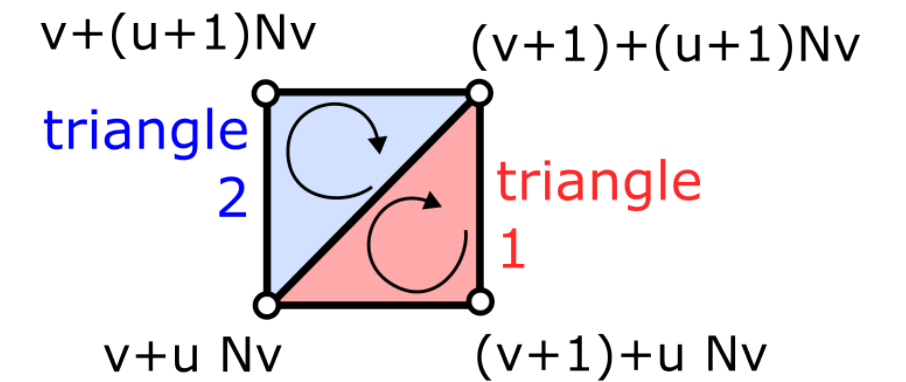
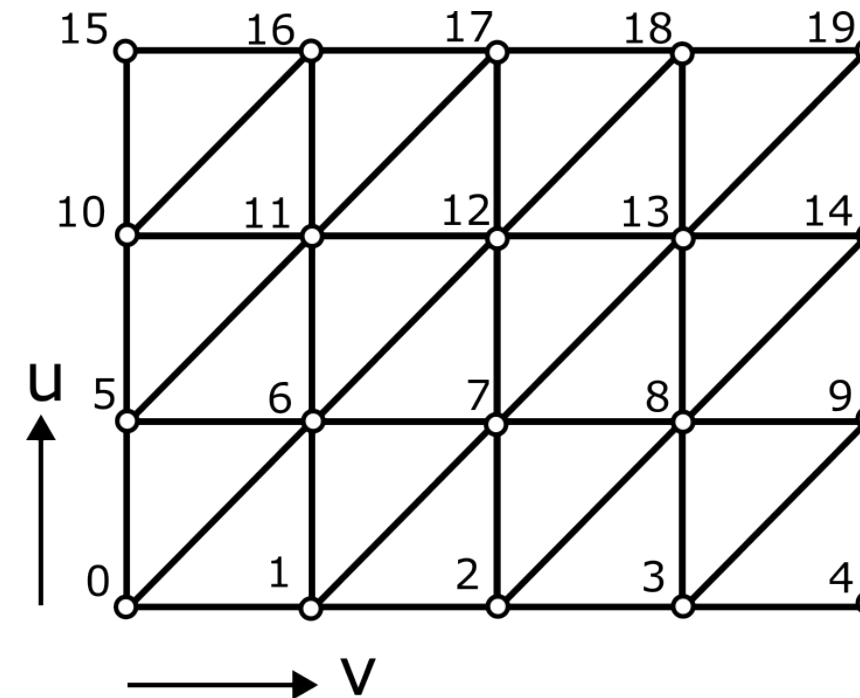
```
// Vertex position
for(int ku=0; ku<Nu; ku++) {
    for(int kv=0; kv<Nv; kv++) {
        // Parametric coordinates (u,v) \in [0,1]
        float u = ku/(Nu-1.0f);
        float v = kv/(Nv-1.0f);

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

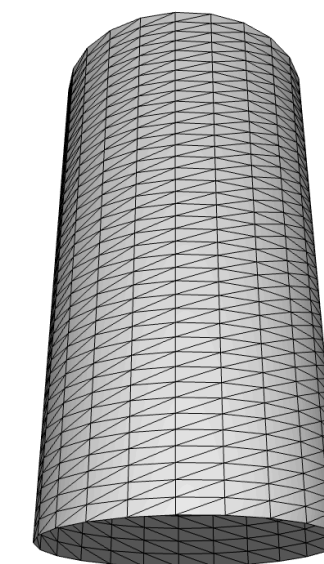
// Connectivity
for(int ku=0; ku<Nu-1; ku++) {
    for(int kv=0; kv<Nv-1; kv++) {
        unsigned int idx = kv + Nv*ku; // offset

        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.

⇒ possibilité d'approximer des normales à partir de la géométrie du maillage.

⇒ utile lors de la déformation de maillages.

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

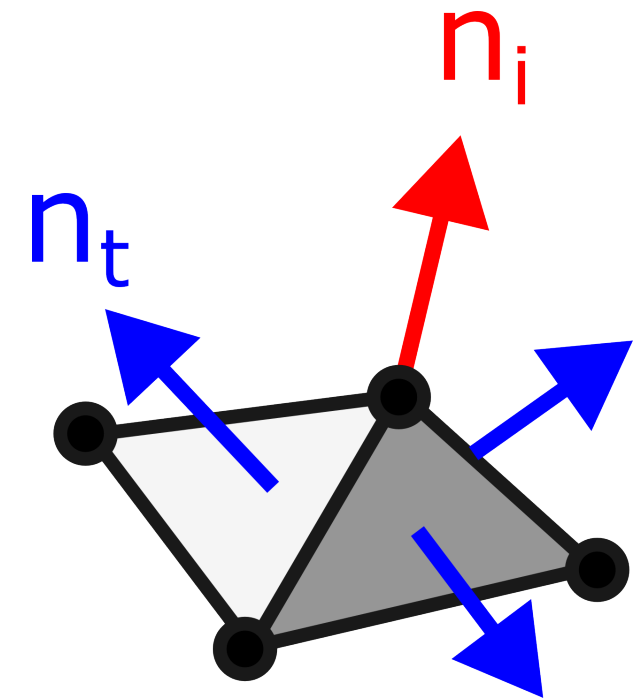
1- Calcul des normales n_t des triangles voisin d'un sommet i , avec $t \in \mathcal{N}_i$

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

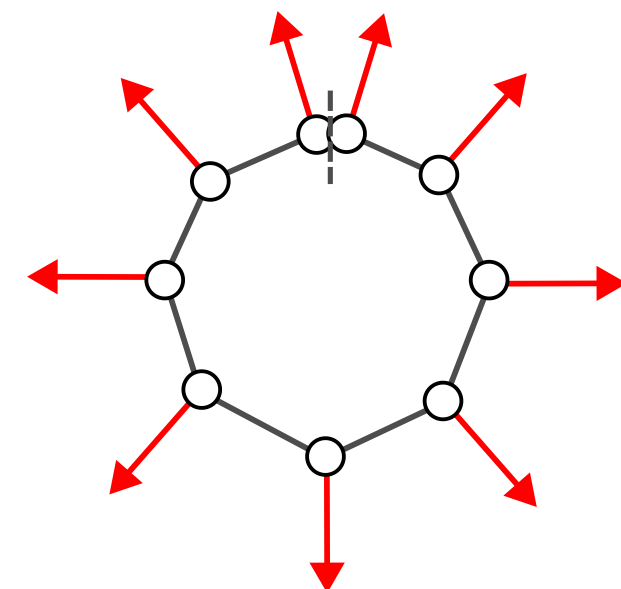
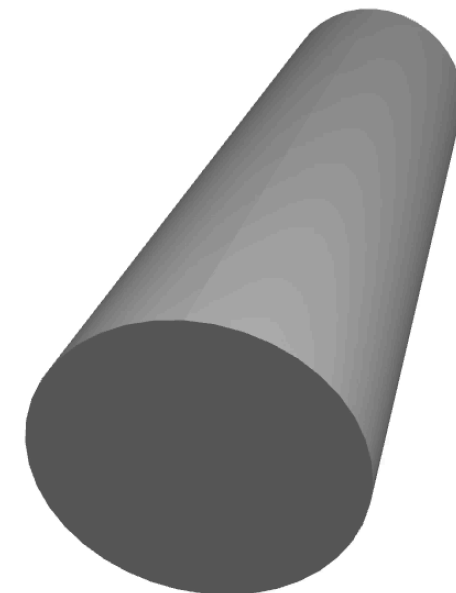
2- Normale du sommet n_i = moyenne normalisée des triangles voisins

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

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



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



Normales des maillages

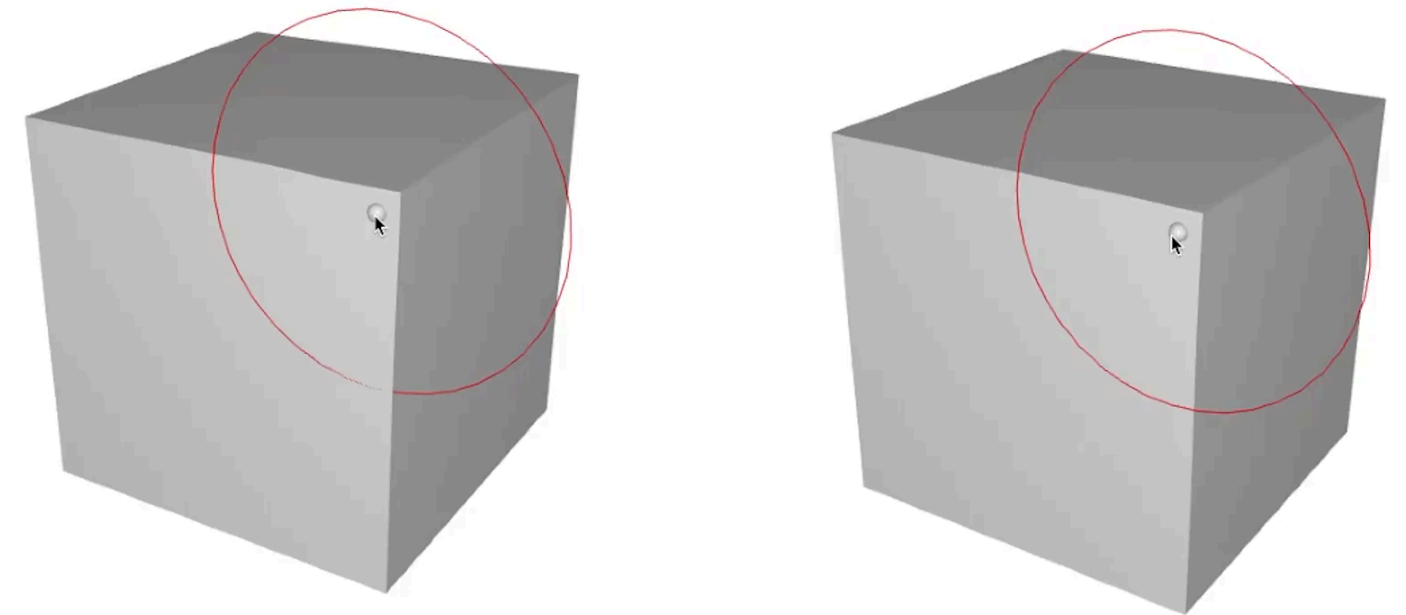
```
std::vector<vec3> normal = position.size();

// 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

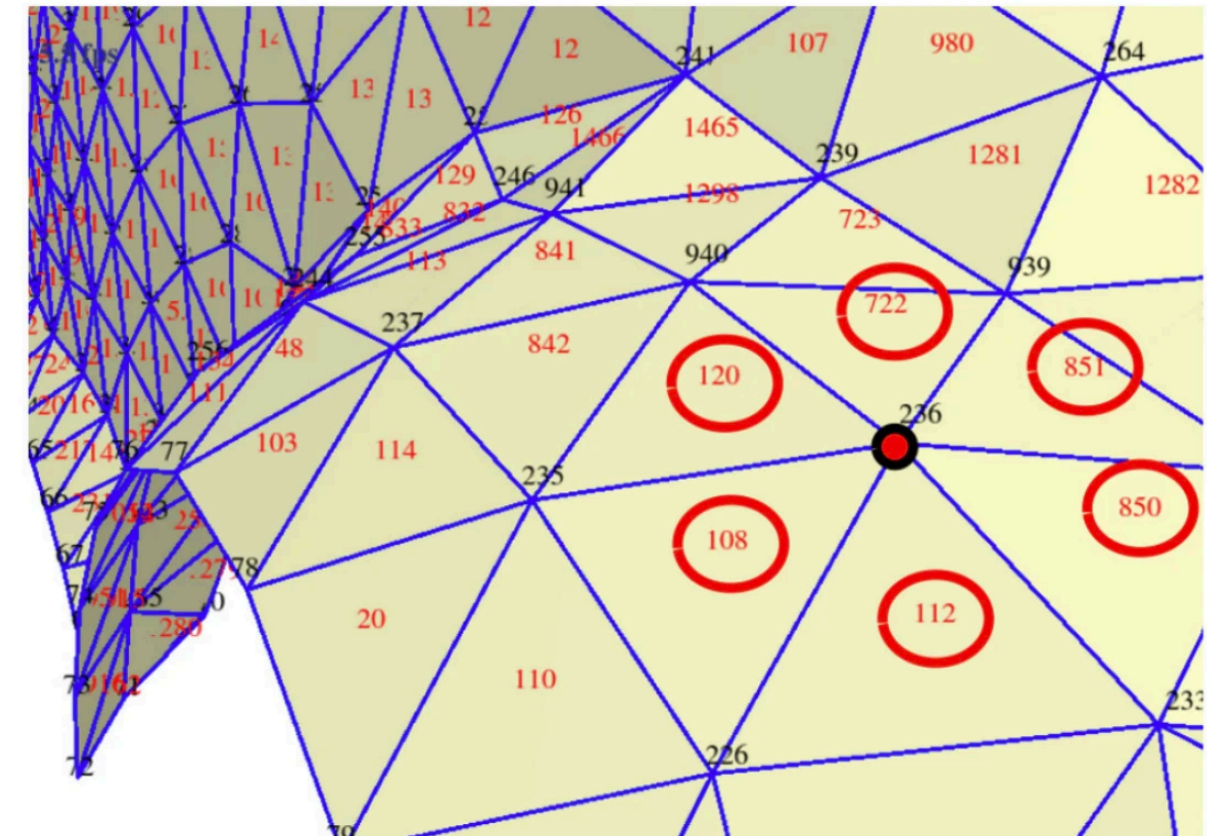
1-voisinage, 1-ring

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

Intérêt:

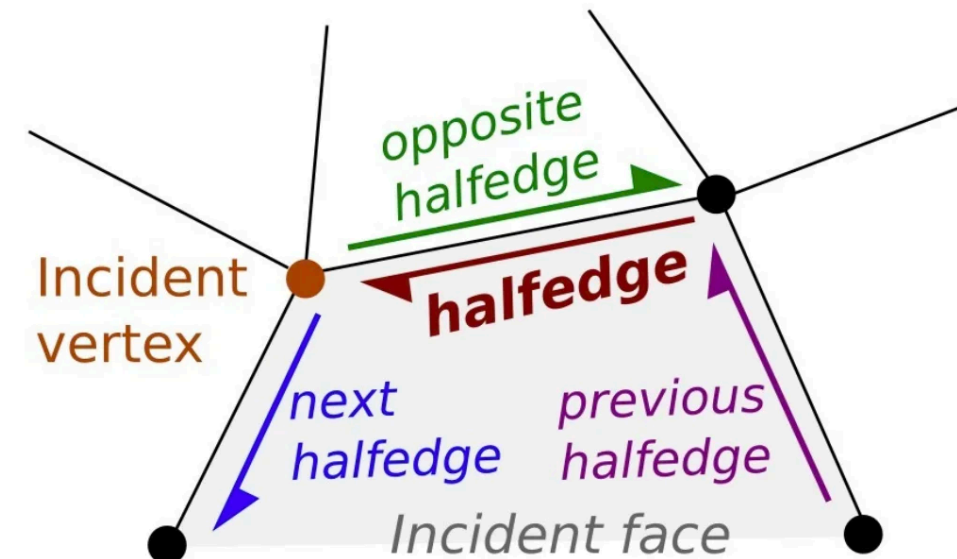
Ajout/ suppression en $O(1)$ (split, collapse)

Calculs d'angles de coins (Cotangent weight)

Limites:

Limité aux 2-variétés

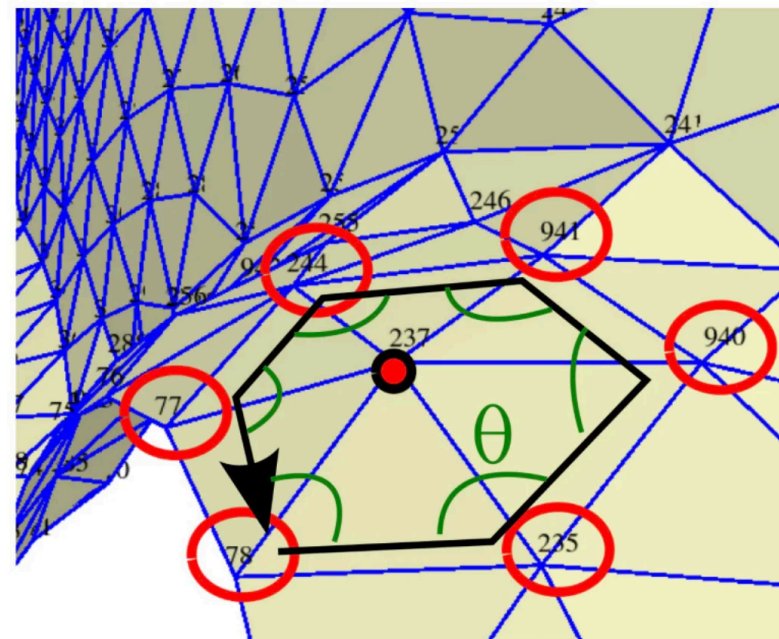
Non-contigues en mémoire



Halfedge

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

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



```
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

- Géométrie
- **Texture UV**

Textures

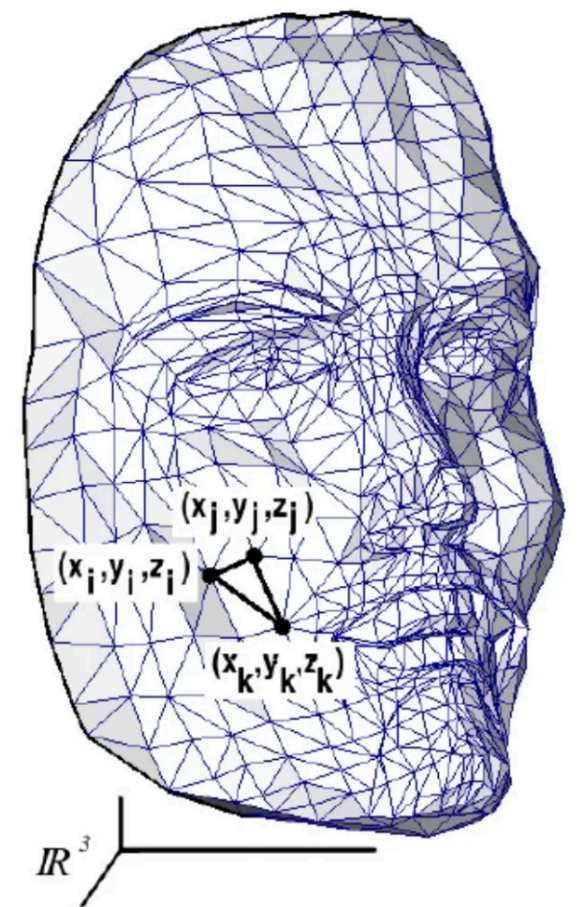
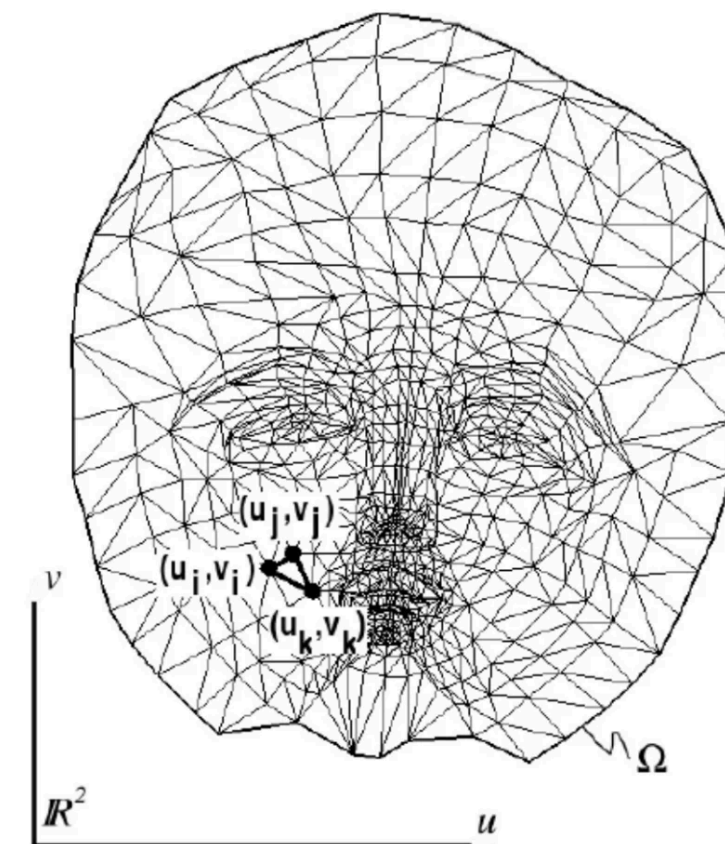
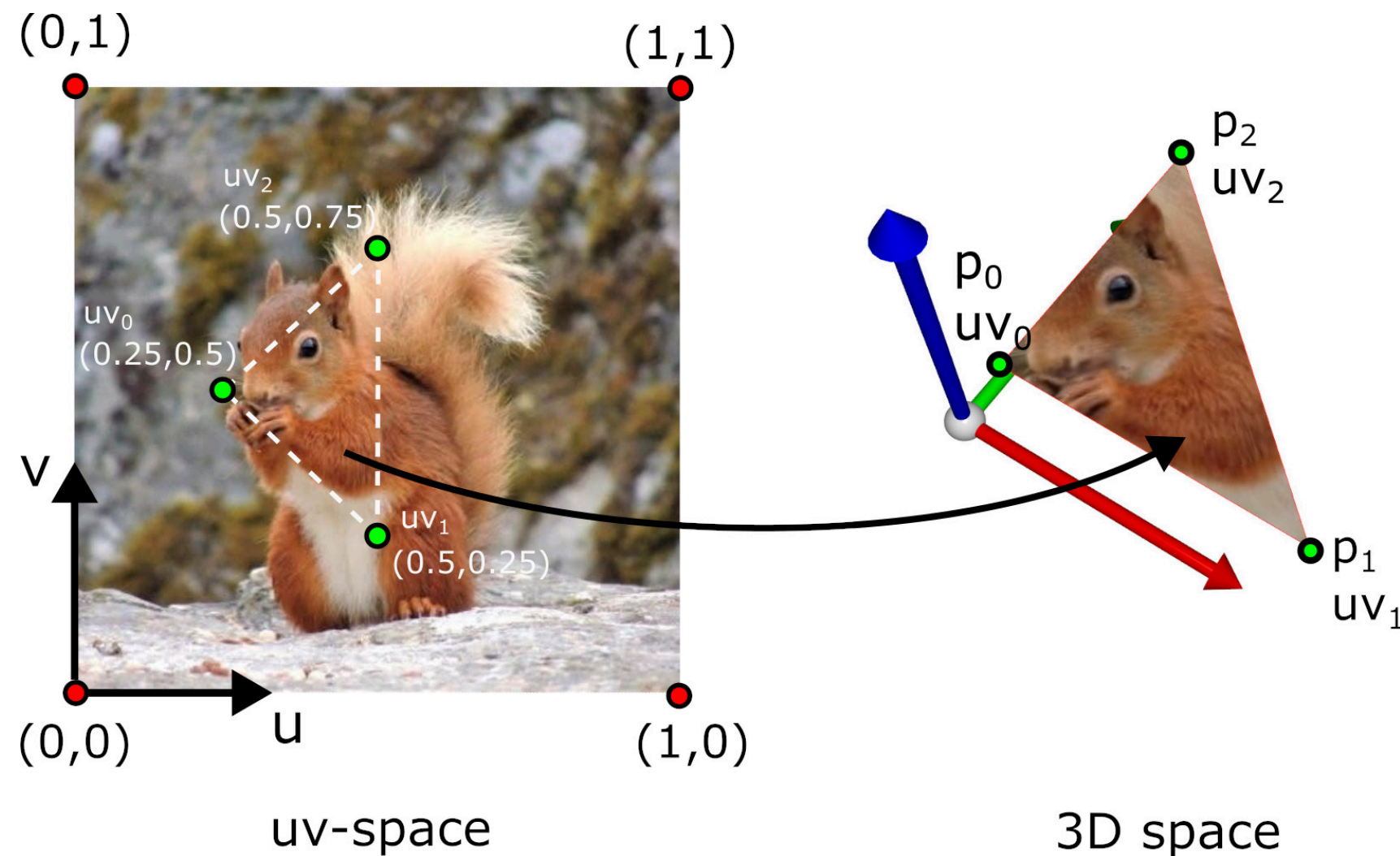
Objectif: Niveau de détail (couleur) plus petit que la géométrie / triangle

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

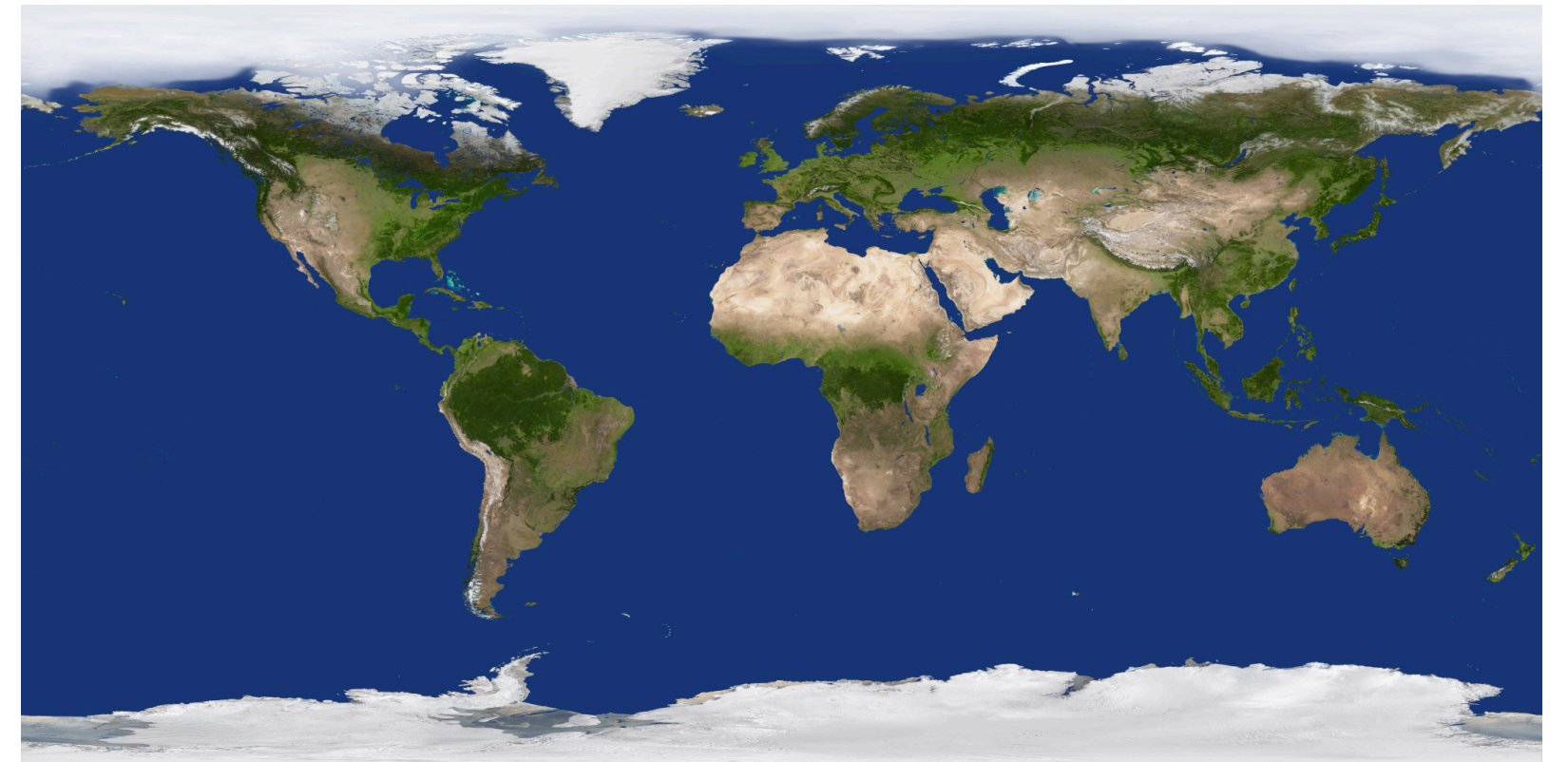
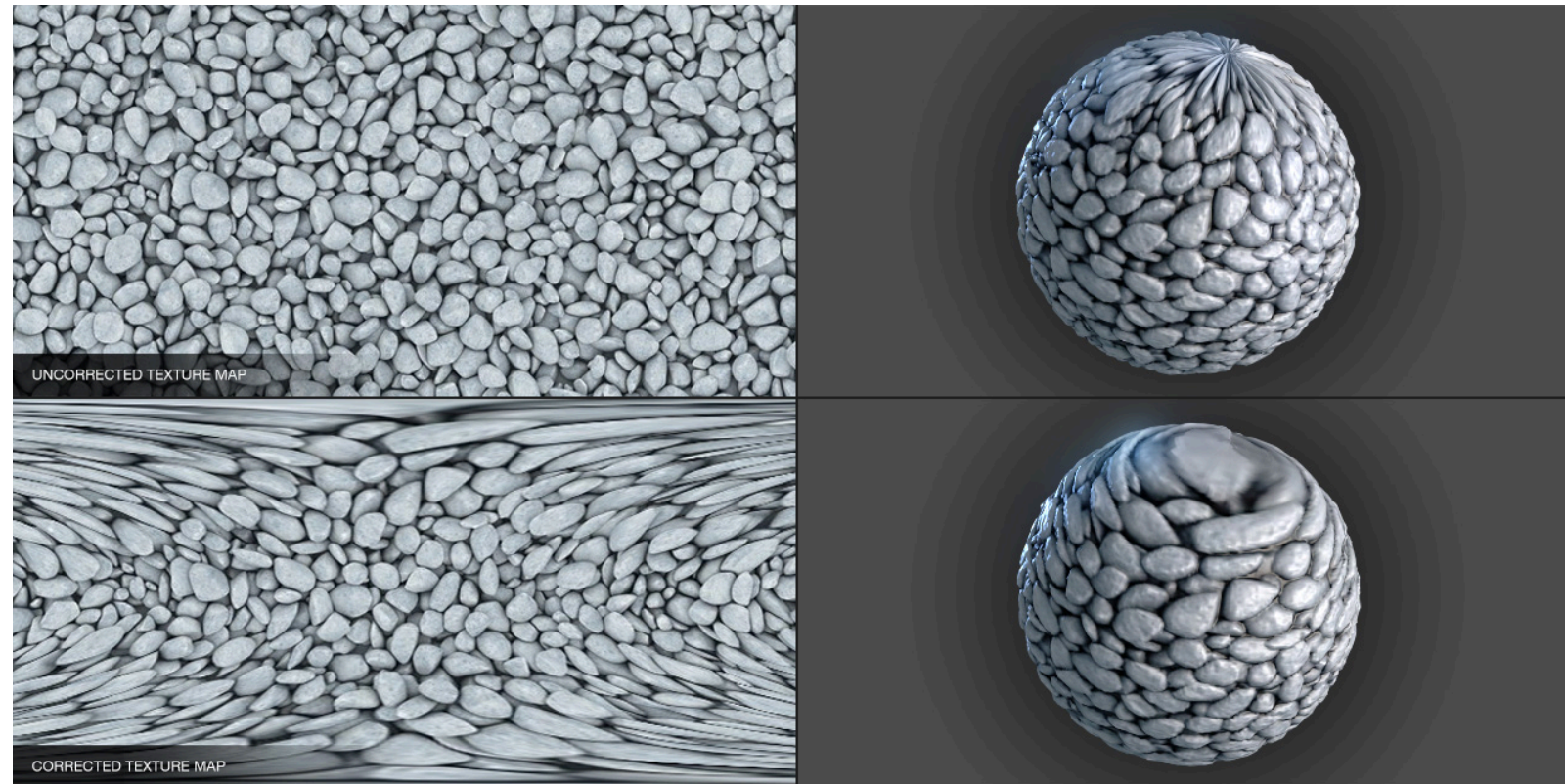
Au rendu (fragment shader): Affiche la couleur de l'image à la coordonnée interpolée



Textures - uv-mapping

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

Généralement associé à des déformation de longueurs et d'angles

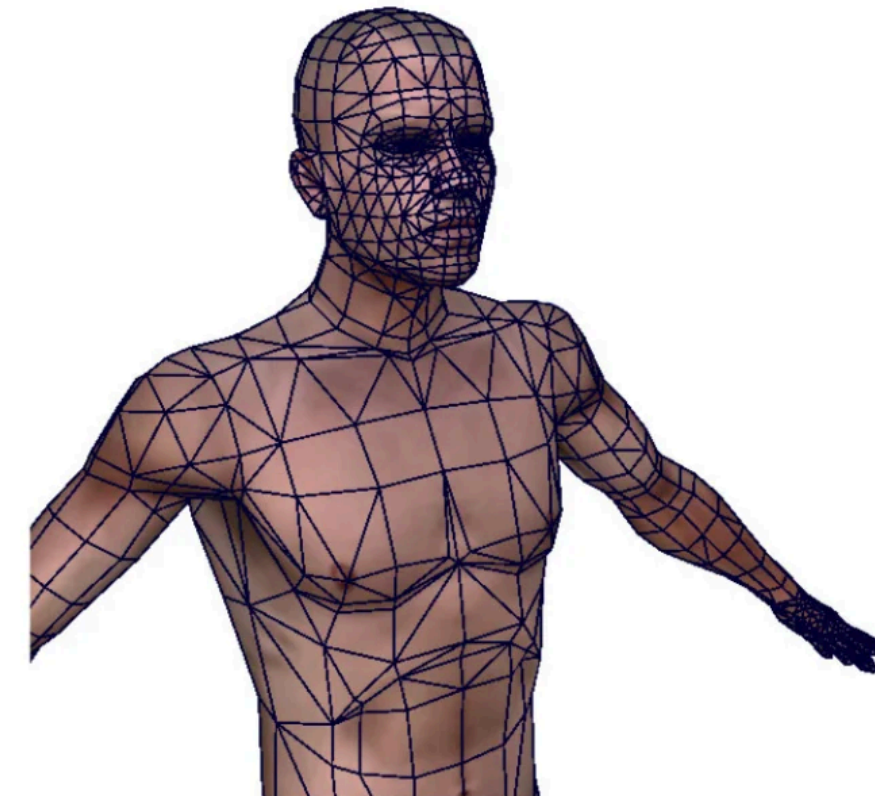
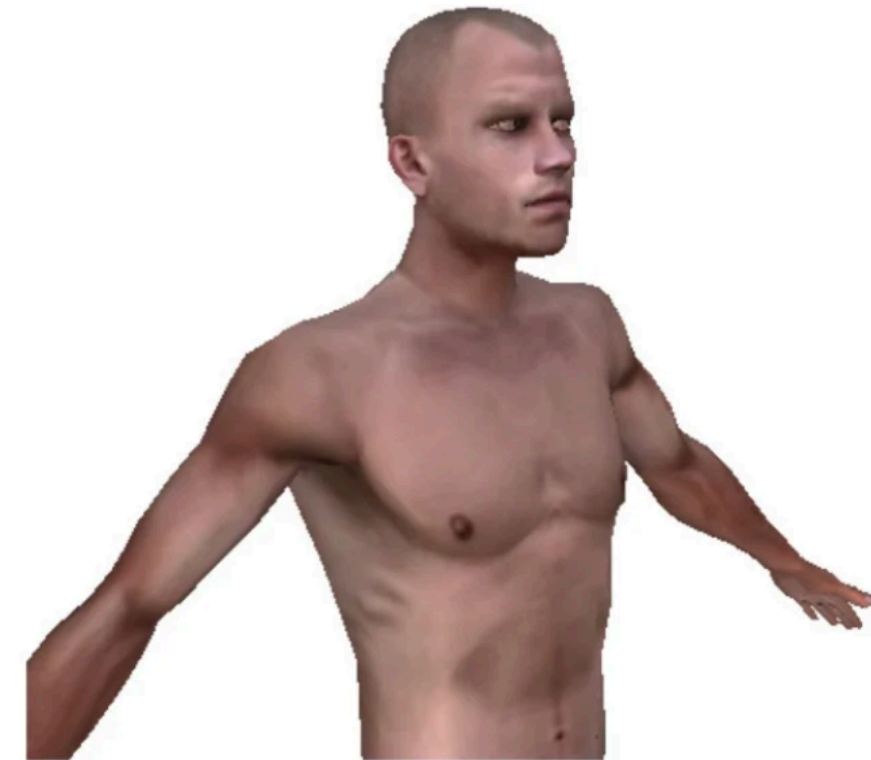
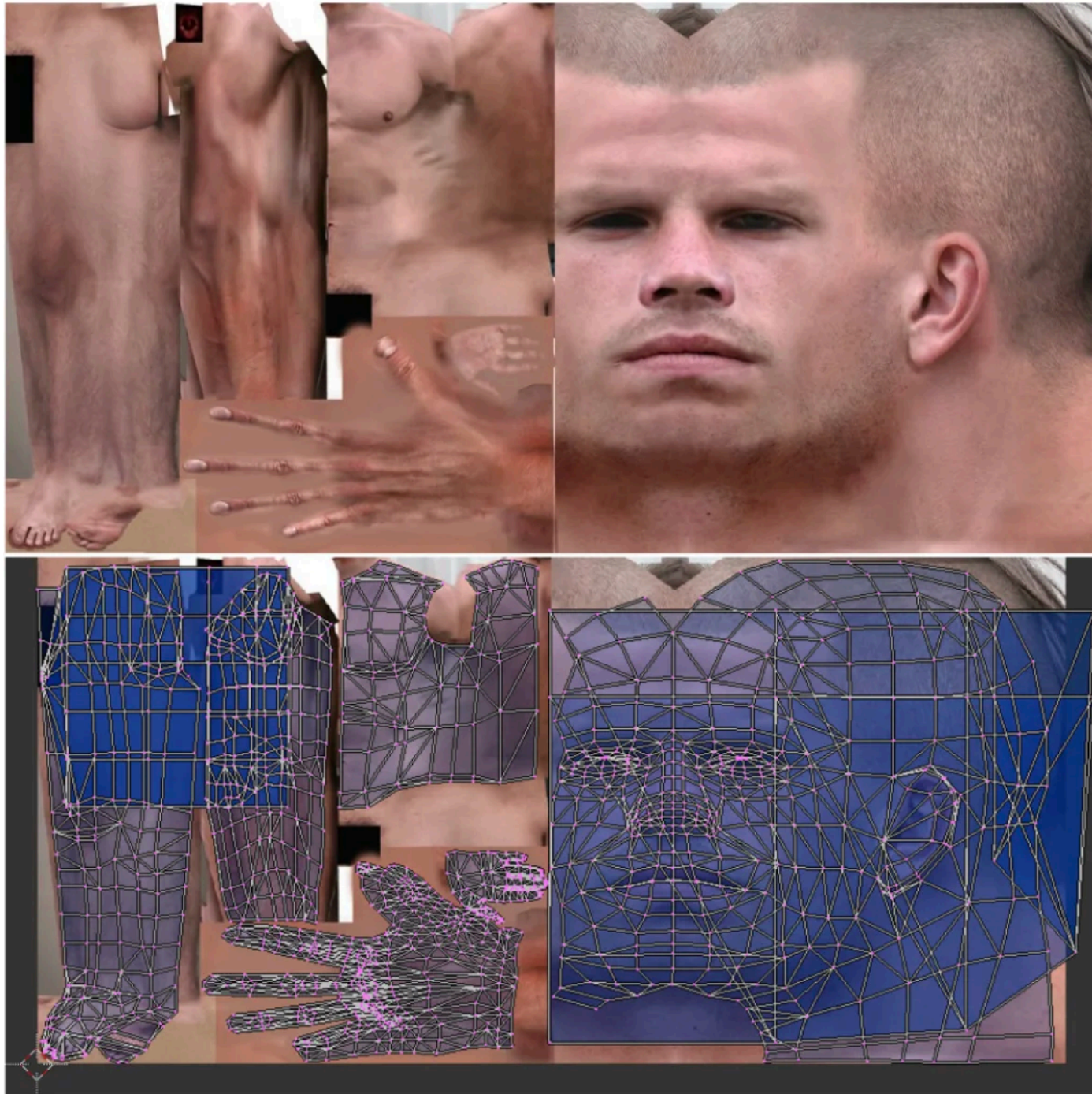


Rem. Impossible de plaquer une image plane sur une sphère sans déformation de longueurs

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

Textures - cas réel

Définition par morceaux (patches), pouvant se recouvrir



Effets: Skybox

Boite texturée d'un environnement

Toujours centré sur la position de la caméra.

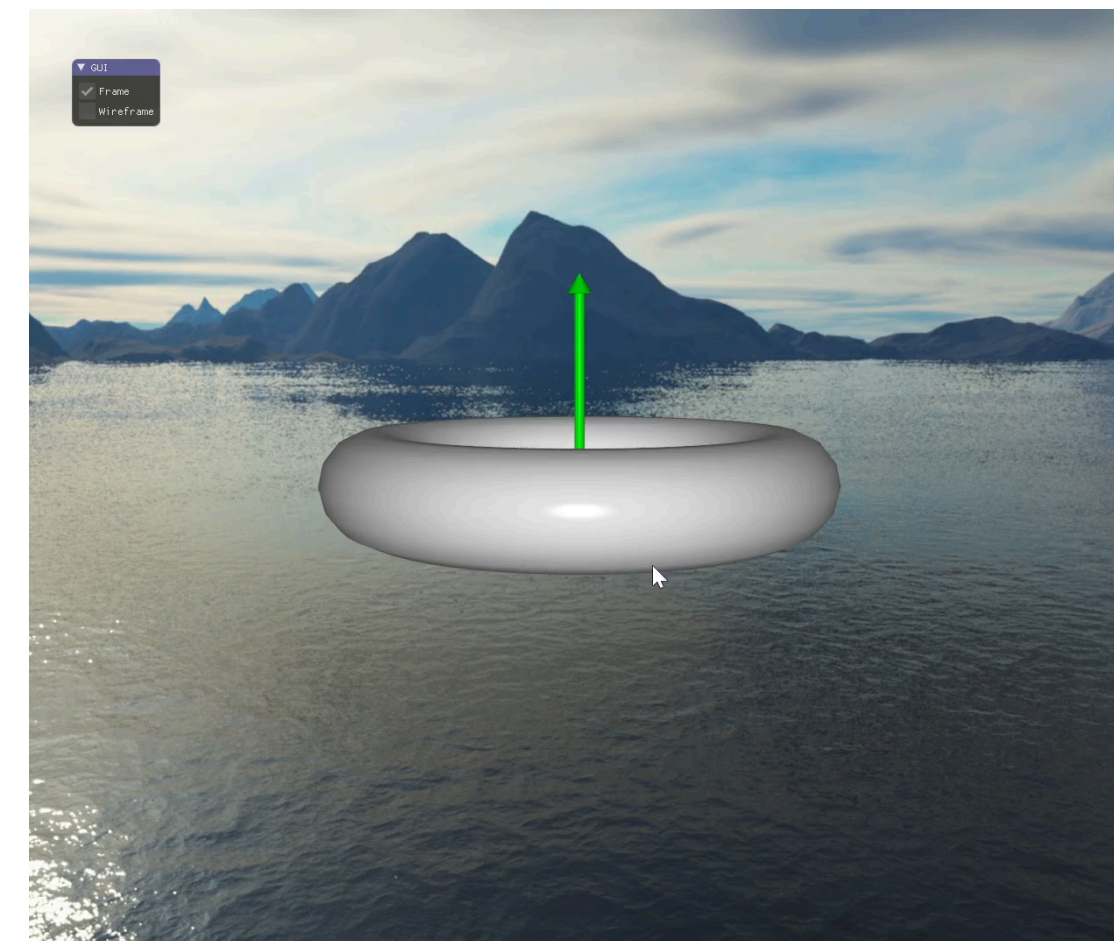
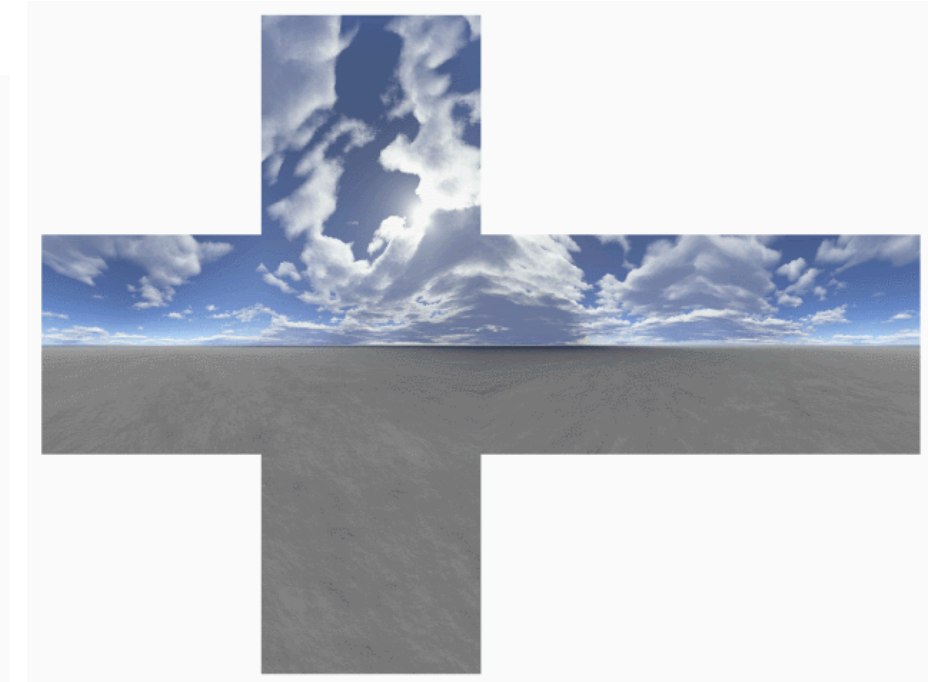
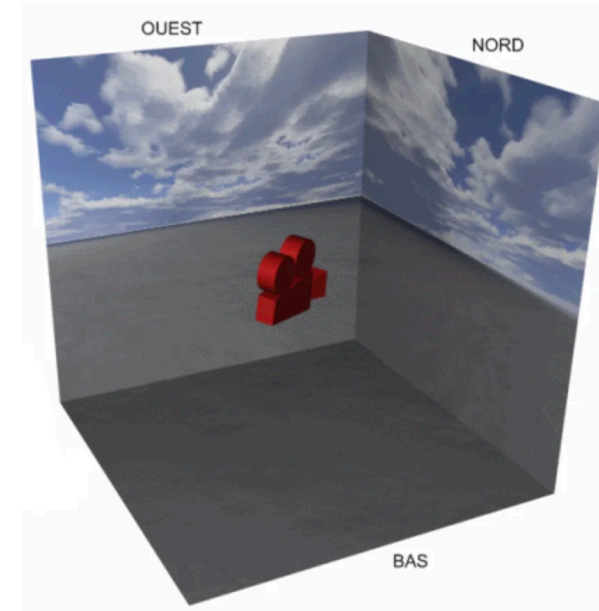
Impression de paysage à l'infini.

Principe

Afficher la skybox en premier

Afficher tous les autres objets ensuite, toujours au dessus

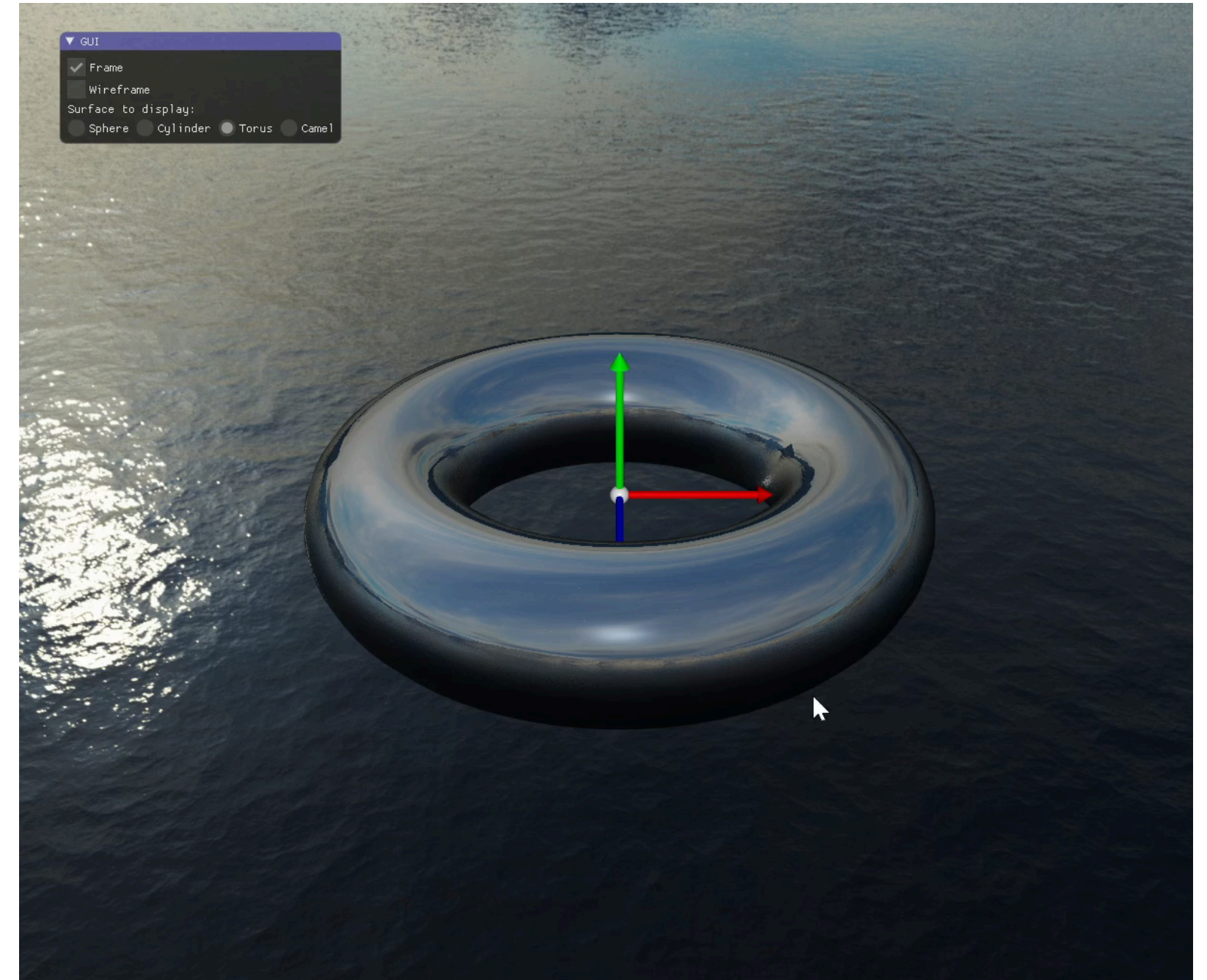
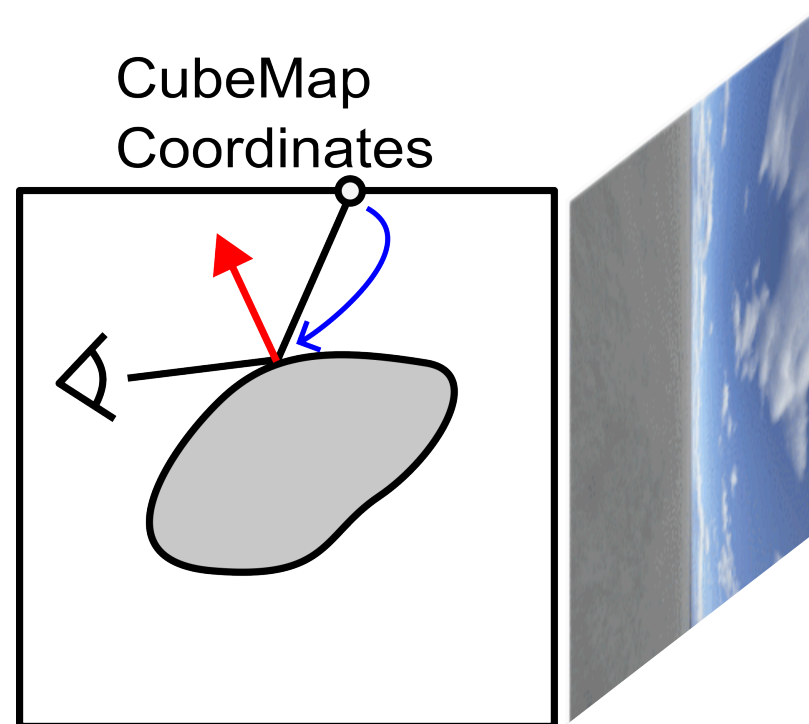
```
display(){  
    glDepthMask( GL_FALSE ); // disable depth-buffer writing  
    // draw skybox, ex. draw(skybox, environment);  
  
    glDepthMask( GL_TRUE ); // re-activate depth-buffer writing  
    // draw other objects  
    // ...  
}
```



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

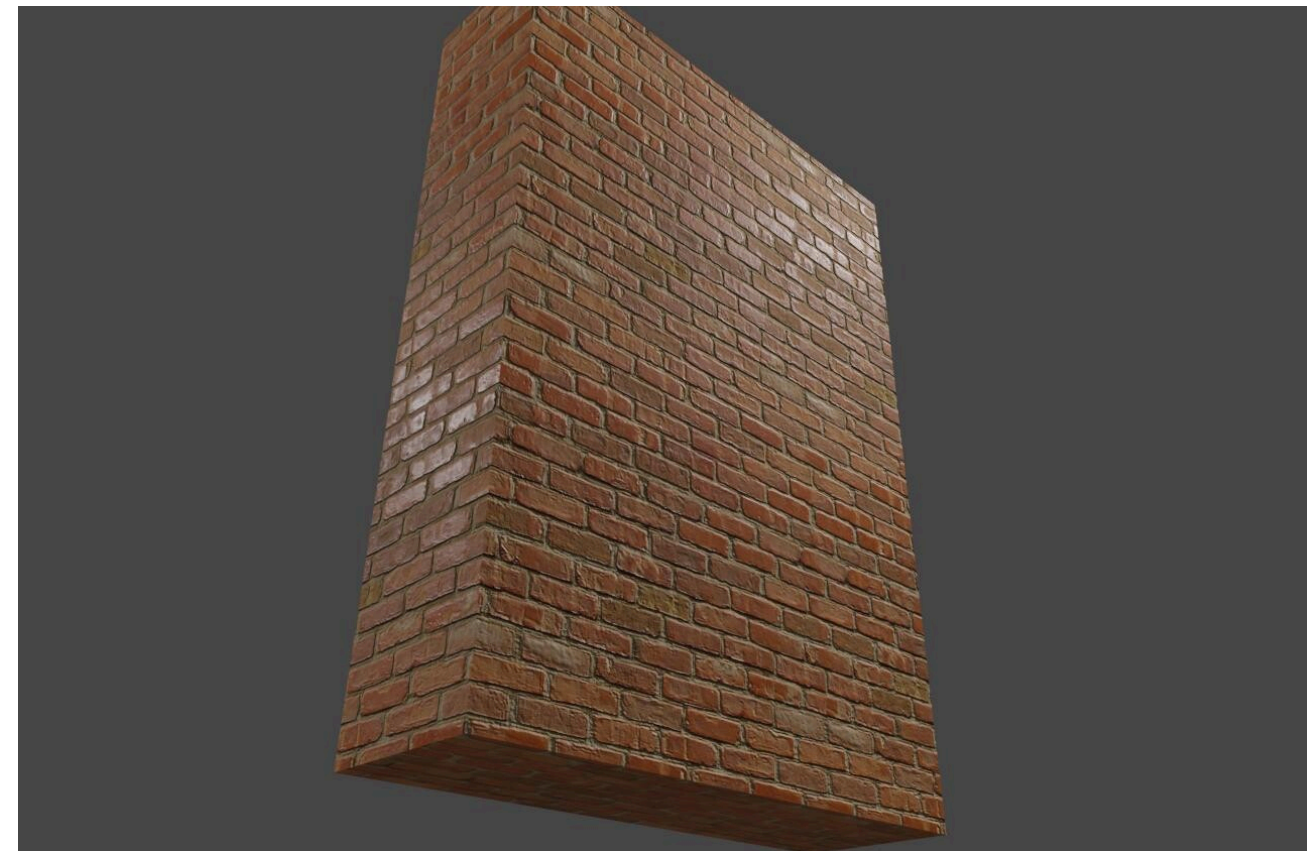
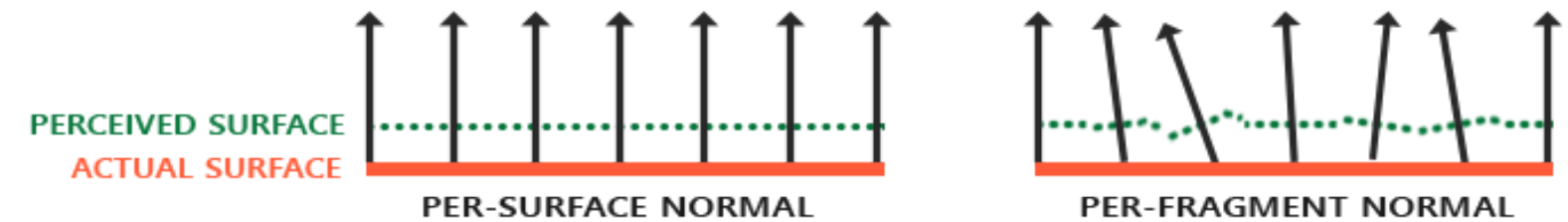
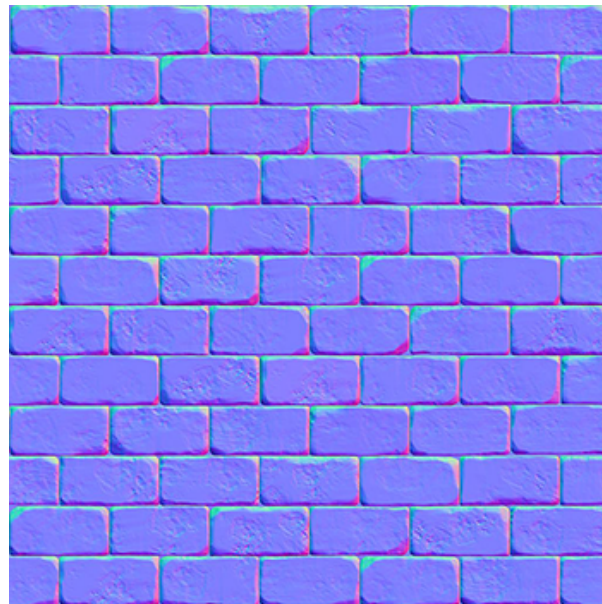
La géométrie des triangles reste inchangée

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) .

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



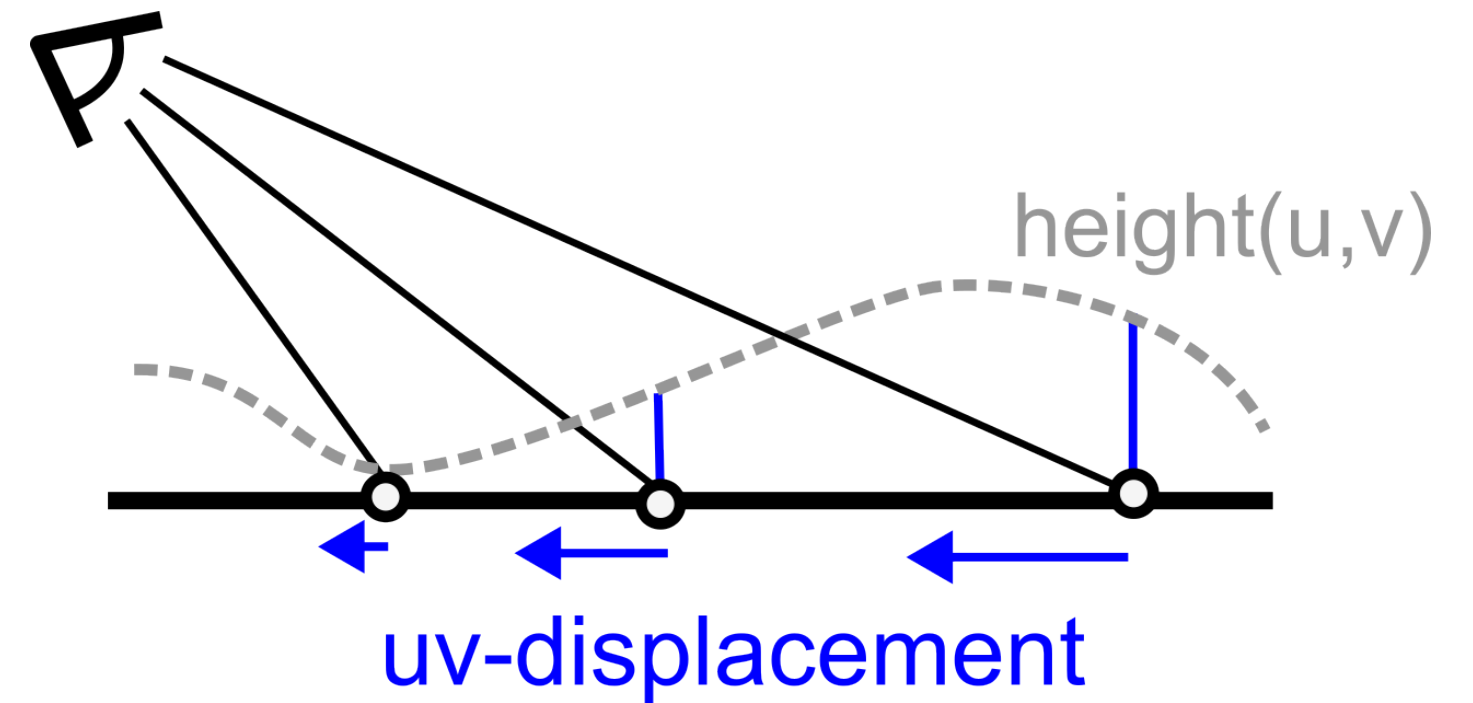
Effets: Parallax Mapping

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](#)]

Particules, imposteurs / billboard

Exemple de système de particules

Chute libre de particules

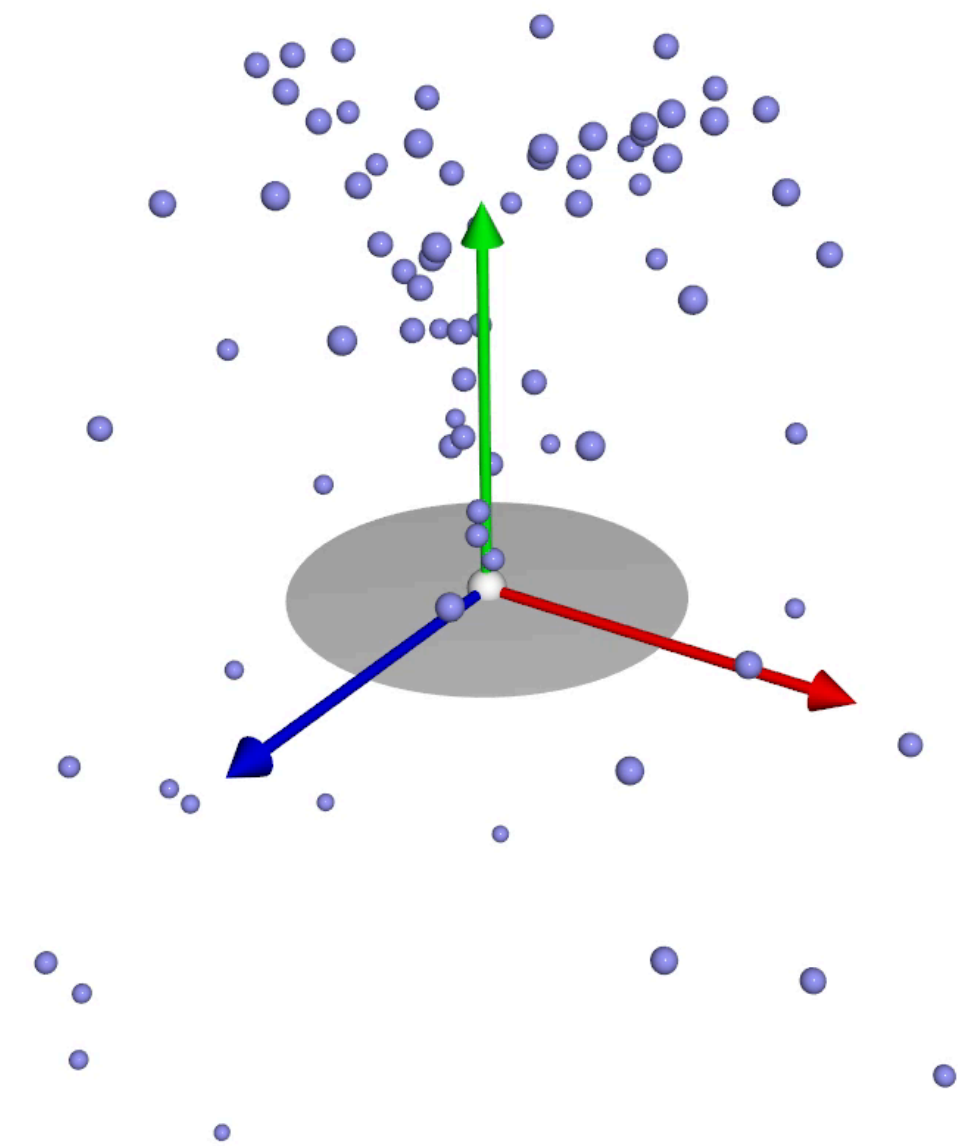
- Représentation géométrique des particules: sphere
- Equation du mouvement $p(t) = \frac{1}{2}gt^2 + v_0t + p_0$
- Position initiale et vitesse peuvent avoir des composantes aléatoires
- Chaque particule peut avoir une durée de vie limitée

Q. Quels sont les paramètres utilisés pour p_0 et v_0 dans cet exemple ?

$p_0 = \dots$

$v_0 = \dots$

Note: Système de coordonnées $(x, y, z) = (\text{rouge}, \text{vert}, \text{bleu})$.



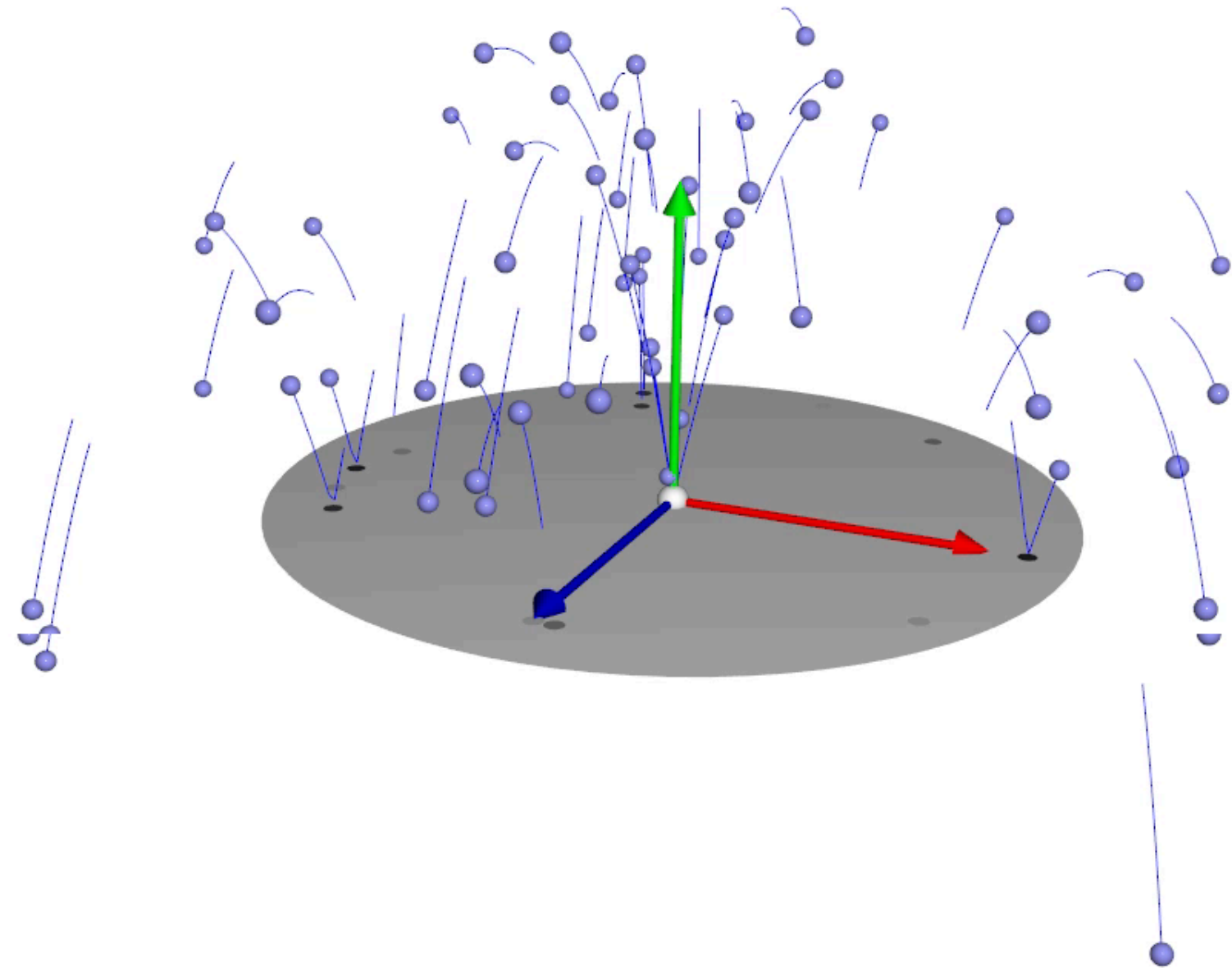
Rebond de sphères

Q. Quelle est l'équation du mouvement ?
(en prenant en compte un rebond)

$$p(t) = \dots$$

Aide:

- Considère une particule émise à $t = 0$
- A quel temps t_i , la particule touche le sol ?
- Quelle est la nouvelle vitesse après impact ?
- Exprimer l'équation du mouvement complète (définition par morceaux)

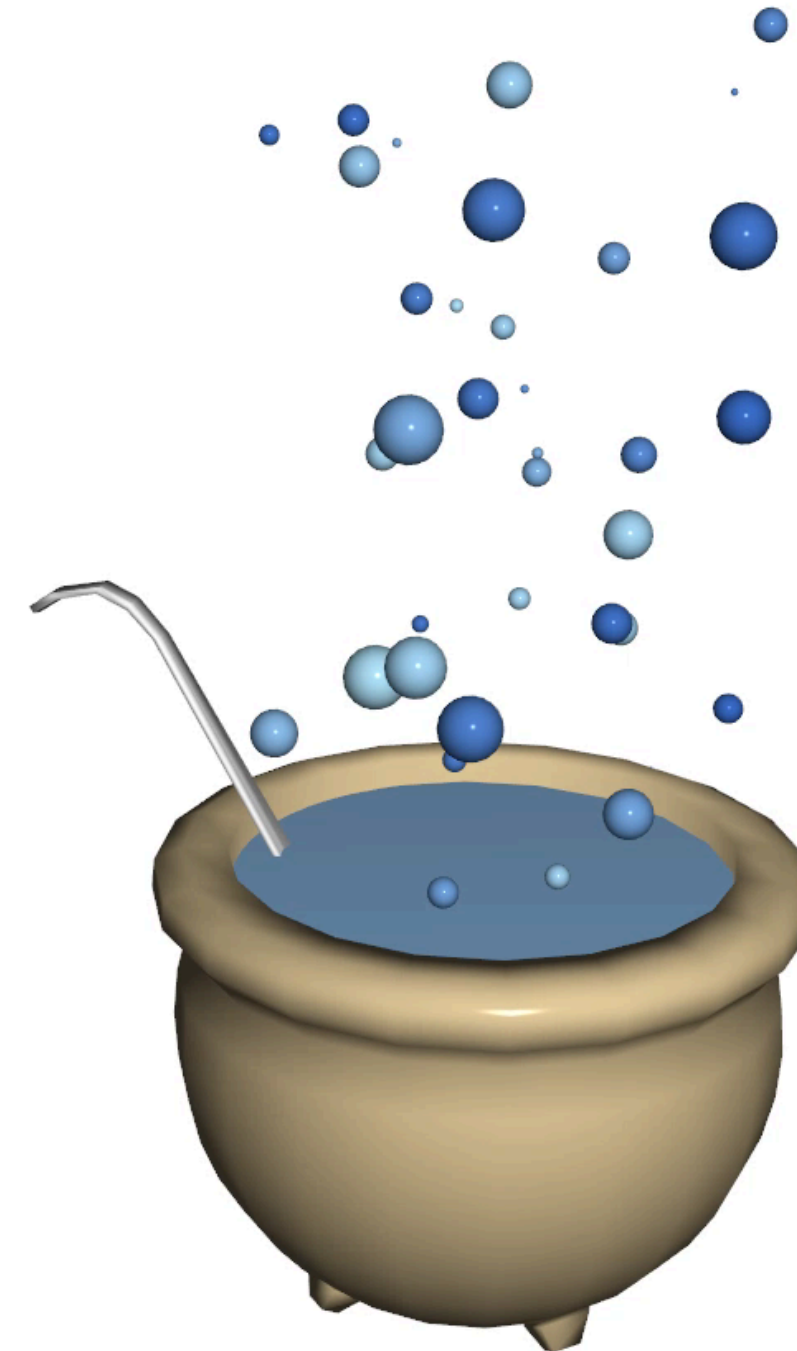


Mouvement générique

Trajectoires non restreintes à des mouvements physique.

Q. *Quels sont les paramètres (et valeurs initiales) associées à ces particules ?*

- position = ...
- radius = $\text{rand}([0,0.1])$
- ...
- *Quelle est l'équation du mouvement d'une particule ?*
 - $p(t) = \dots$



Billboards, Imposteurs, Sprites

Particules peuvent être affichées en tant qu'images
à la place de sphères

En pratique

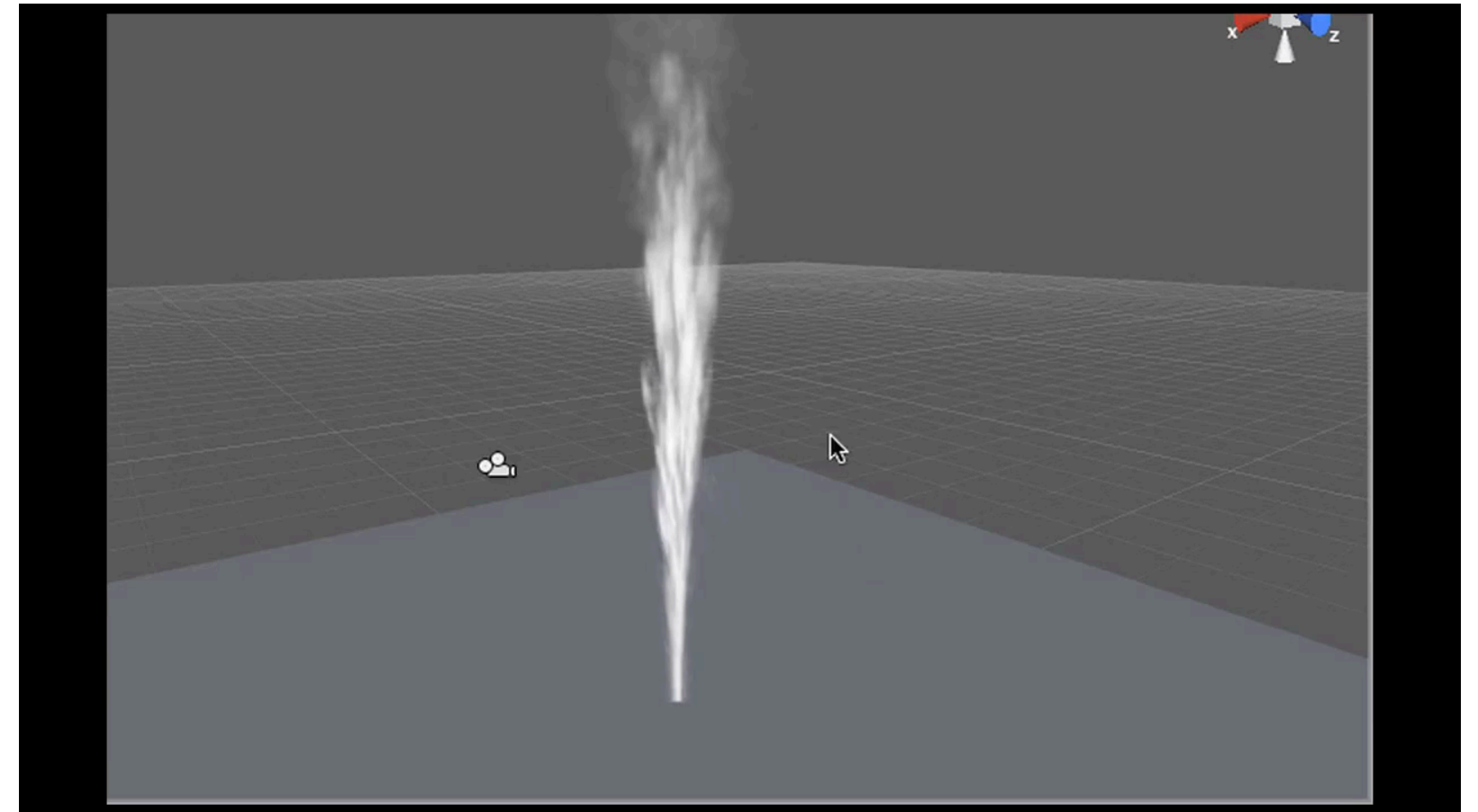
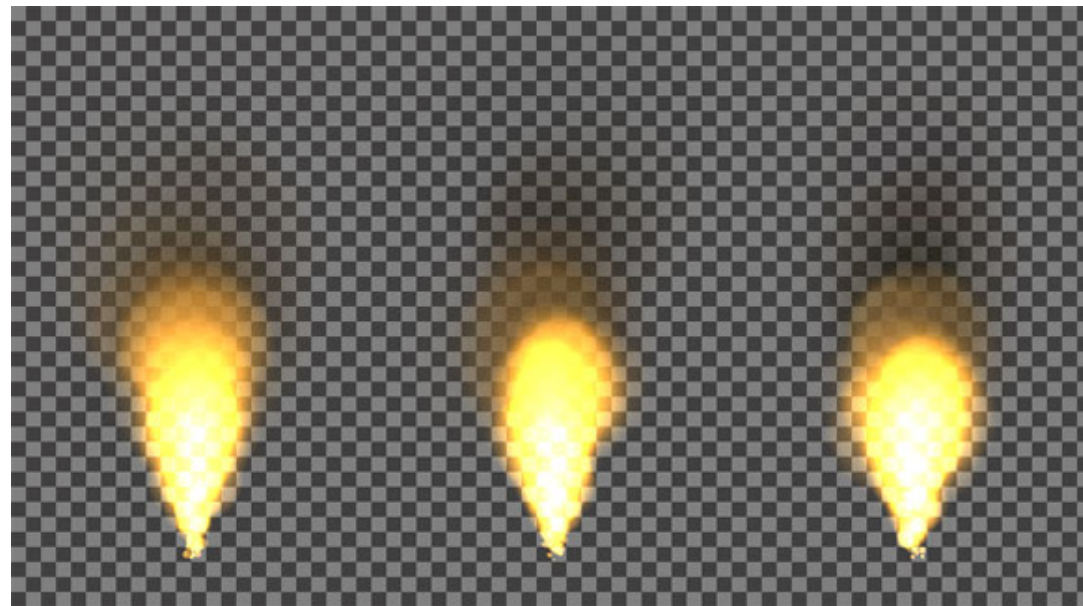
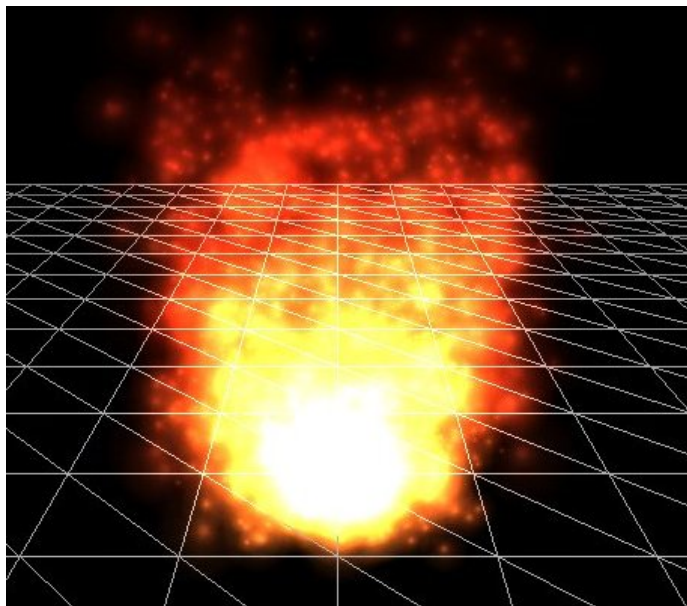
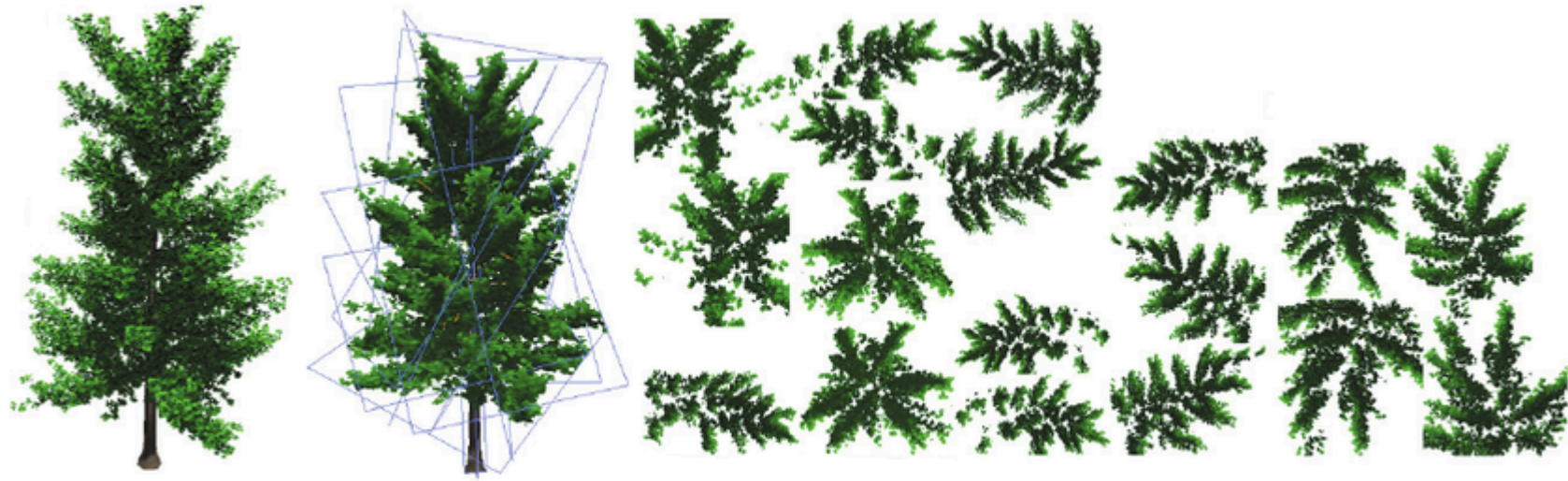
- Quand particule est affiché par un quadrangle.
- Une texture est appliqué sur le quadrangle.
- La texture peut contenir des parties transparents: La géométrie du quadrangle est invisible.
- Impression d'une géométrie complexe
- Le quad peut faire face à la caméra en permanence (billboard)
- La texture peut être animée (sprite)
- La texture peut s'adapter au point de vue (imposteurs)



Utilisation des billboards

Utilisation classique pour les modèles complexes

Végétation, feu, fumée, etc.



Exemple d'utilisation en production

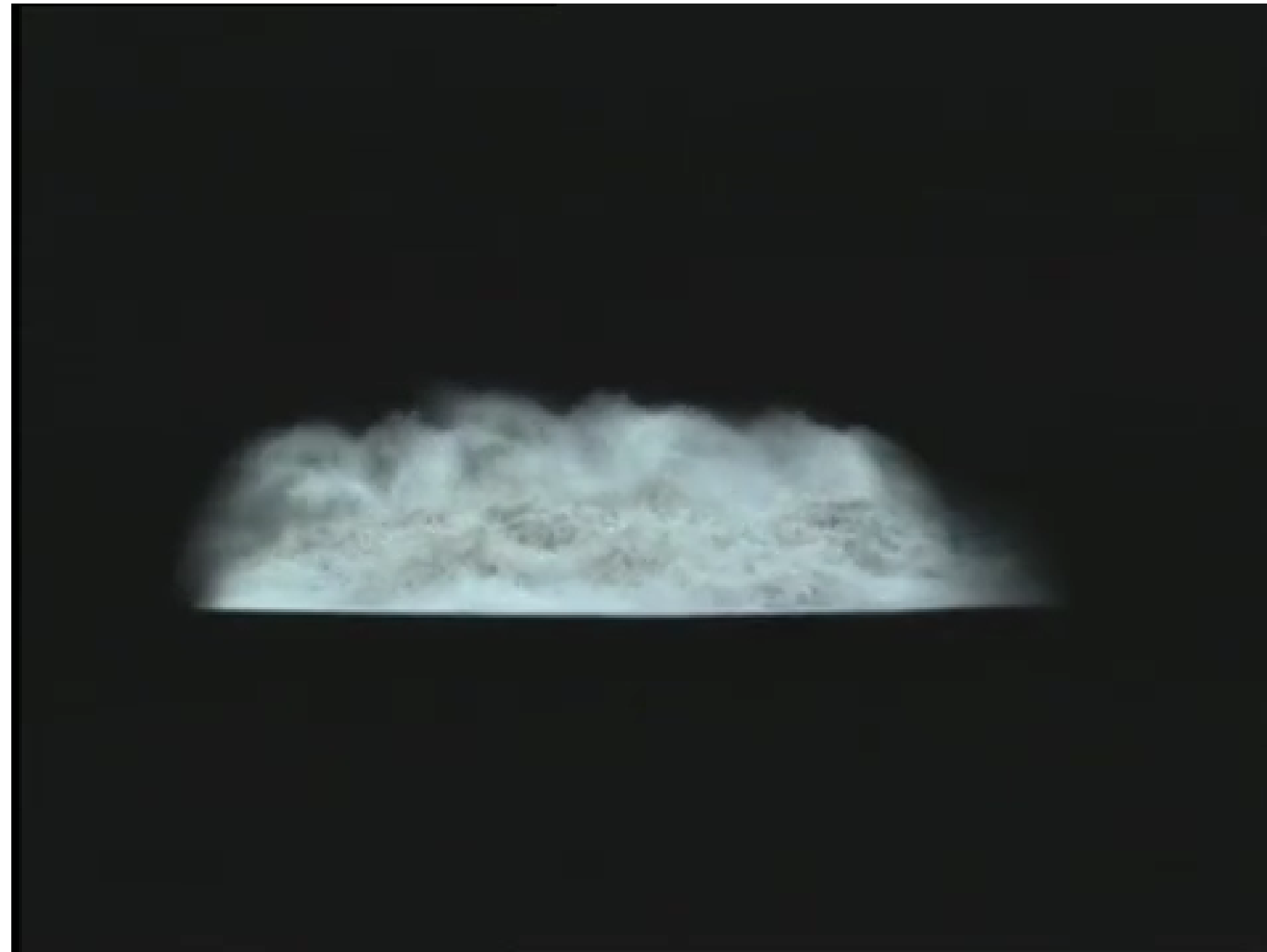
Comment sont réalisés les *têtes de chevaux en eau* ?



The Lord of the Rings

Exemple d'utilisation en production

Comment sont réalisés les *têtes de chevaux en eau* ?



The Lord of the Rings

Exemple d'utilisation en production

Making-Of complet



The Lord of the Rings

Implementation - transparence en OpenGL

Effet de transparence non trivial avec le rendu par projection / rasterisation.

Pas de solution parfaite

⇒ Possibilité de mélanger les couleurs pour chaque pixel en utilisant le canal α (rgba)

Nécessité d'activer explicitement le mode de *blending* en OpenGL

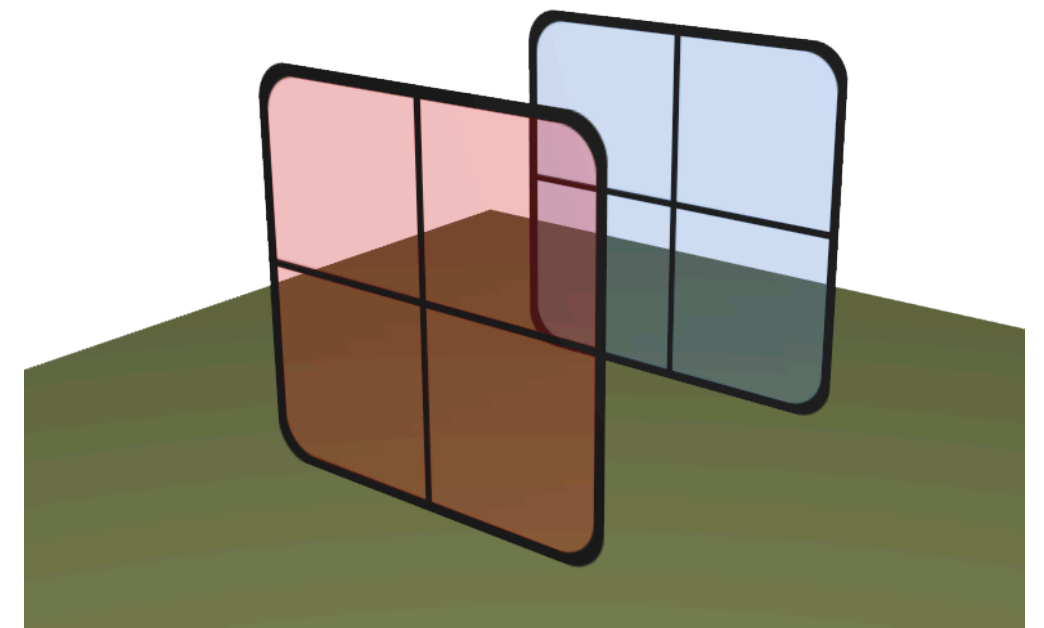
```
glEnable(GL_BLEND);  
glBlendFunc (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  
... Draw Calls ...  
glDisable(GL_BLEND);
```

- Plusieurs mode de mélange sont possibles pour différents effets (see [glBlendFunc](#))
- For transparency: (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
⇒ $\text{newColor} = \alpha \text{ currentColor} + (1 - \alpha) \text{ previousColor}$

Note:

L'**ordre** des appels à draw est important (previousColor / currentColor)

Le buffer de profondeur ne peut pas être utilisé pour des objets semi-transparents.



Implementation - transparency in OpenGL

Approche générale:

- 1. Draw all **non-transparent** objects
- 2.a Activate color blending, deactivate depth buffer write
- 2.b **Sort** transparent objects by depth to the camera

$$p_{proj} = Proj \times ModelView \times p$$

$$z_{depth} = p_{proj}.z / p_{proj}.w$$

- 3. **Draw transparent objects** from furthest to closest

... Draw all non-transparent objects ...

```
glEnable(GL_BLEND); // Color Blending
```

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

```
glDepthMask(false); // do not write on depth buffer
```

... Sort transparent objects by depth ...

... Draw all transparent objects from furthest to closest ...

```
glDisable(GL_BLEND);
```

```
// do not forget to re-activate depth buffer write
```

```
glDepthMask(true);
```

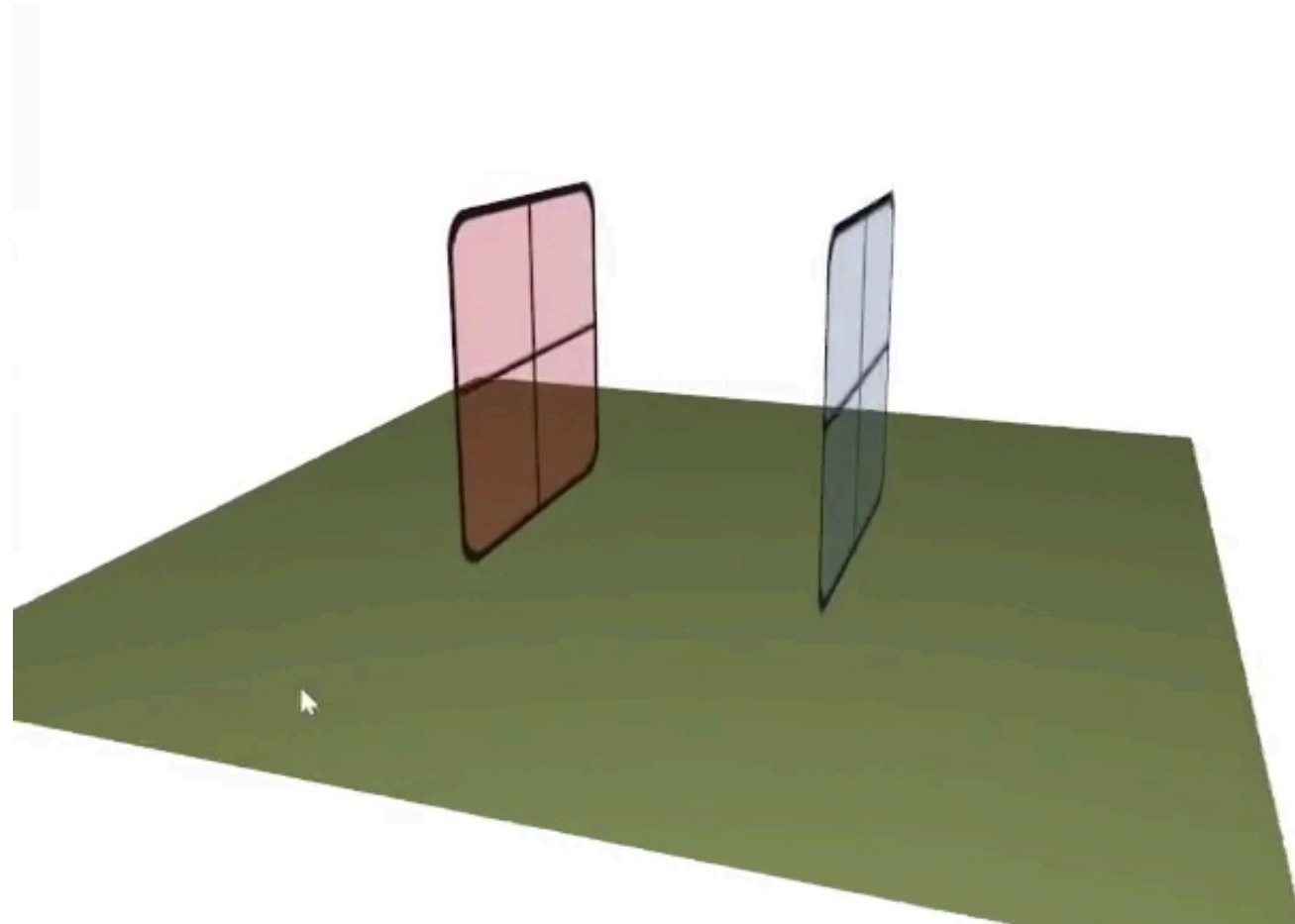
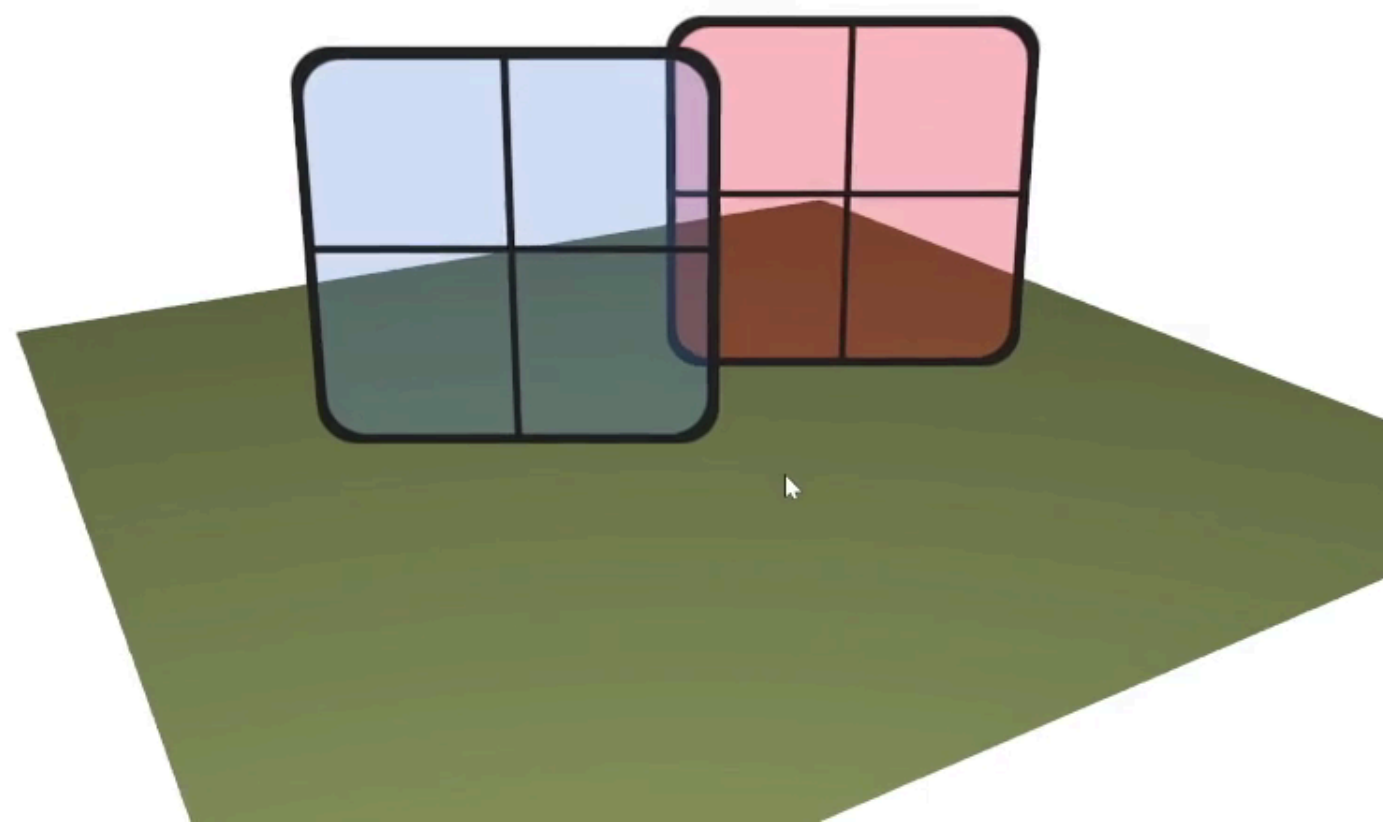


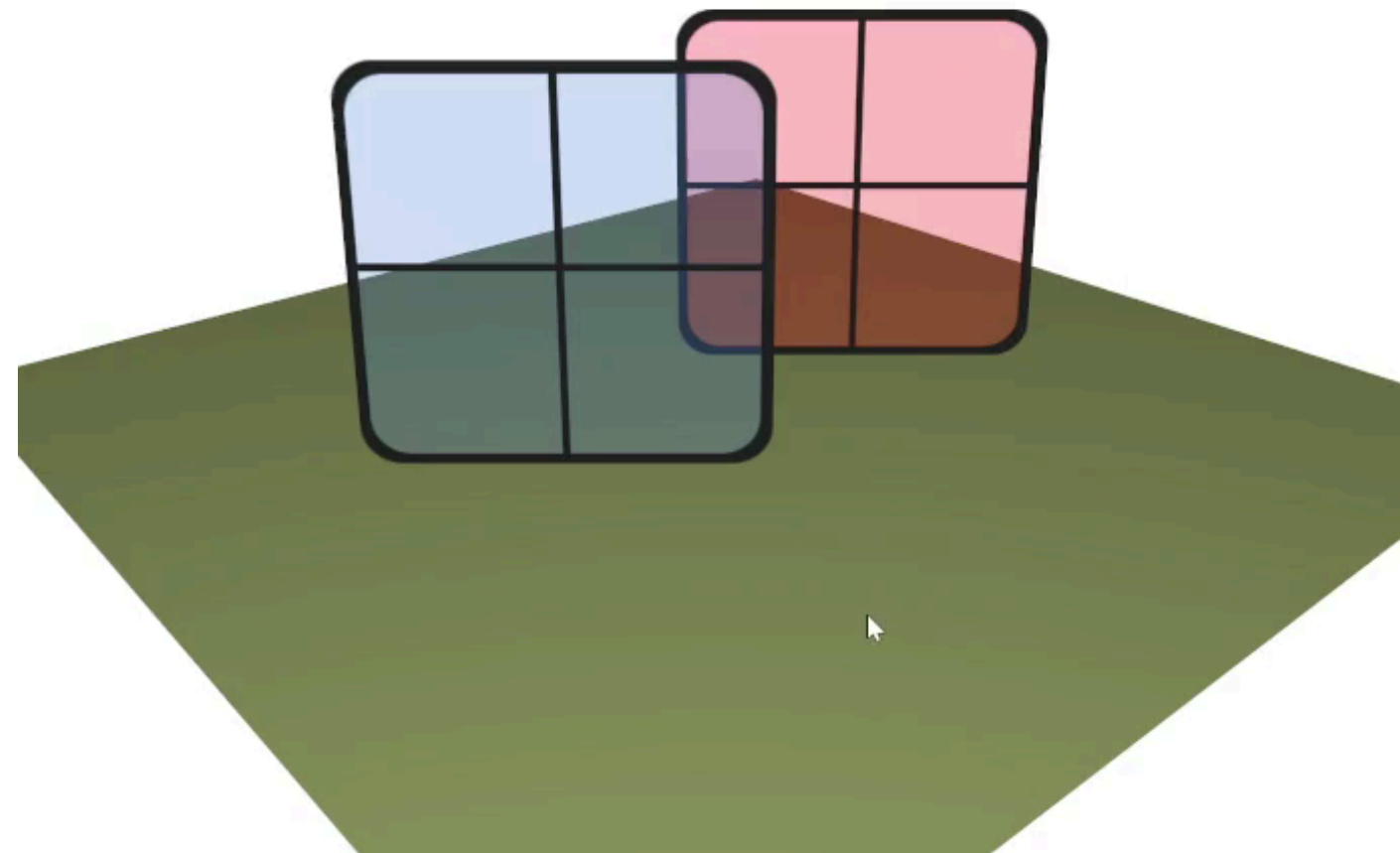
Image Blending - erreur classique

Garder l'écriture dans le buffer de profondeur
(sans tris)



Problème visibles

Pas de tris, mais écriture désactivé du depth-
buffer



Note:

OK pour des sprits flous (ex. fumée)
où l'ordre des éléments est difficile à distinguer.
Evite le cout du tri.

Implementation - transparence en OpenGL

Pour les imposteurs avec des fragments entièrement opaques / transparents:

discard peut être utilisé sur les fragments transparents

discard = Arrête l'exécution du fragment shader (rien n'est écrit)

(+) Générique, standard

pas besoin de trier les formes, le depth-buffer fonctionne

(-) Possibles artefacts d'aliasing

(-) Ne peut pas être utilisé sur des objets semi-transparents



Fragment shader:

```
float limit = 0.6;
if( texture.a < limit) {
    discard;
}
```


Instancing

Les système de particules nécessitent l'affichage de nombreuses formes

De multiples appels à `glDrawElements` peuvent être coûteux

chaque appel nécessite une synchronisation CPU-GPU

```
// Slow for large N
for(int k=0; k<N; ++k)
    someOperator(k)
    draw(shape) // ... glDrawElements(...)
```

Solution: Instancing

`glDrawArrays` → `glDrawArraysInstanced`

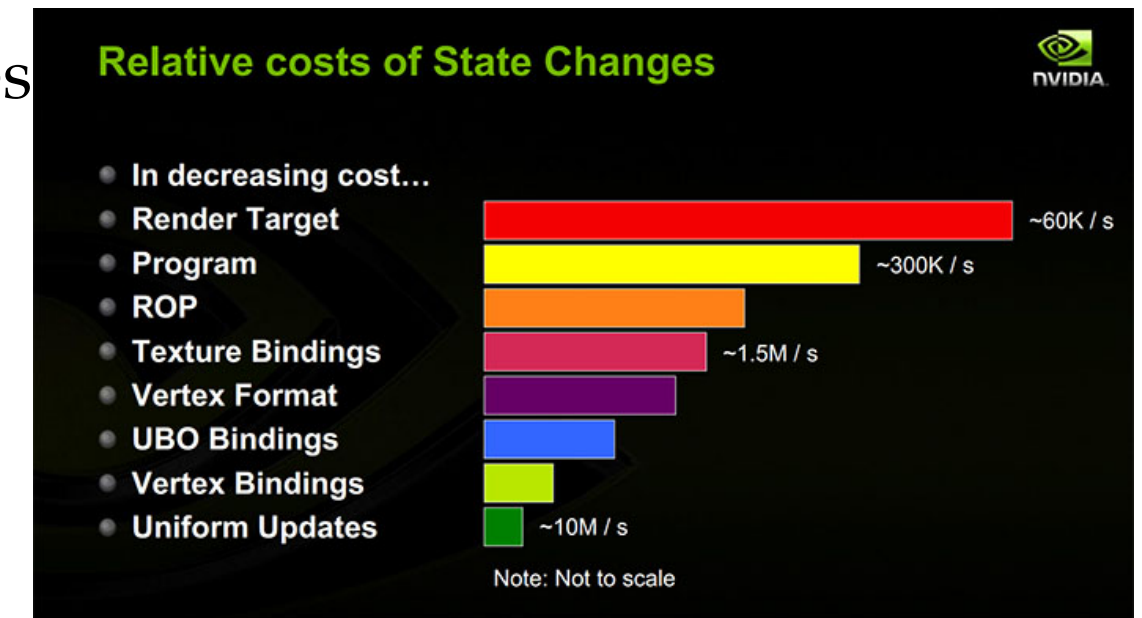
`glDrawElements` → `glDrawElementsInstanced`

In the C++ code

```
// 1 unique call, N is a parameter
drawInstanced(shape, N) // ... glDrawElementsInstanced(...,N)
// No more explicit loop
```

In the Shader:

```
// ...
someOperator(gl_InstanceID)
// ...
```



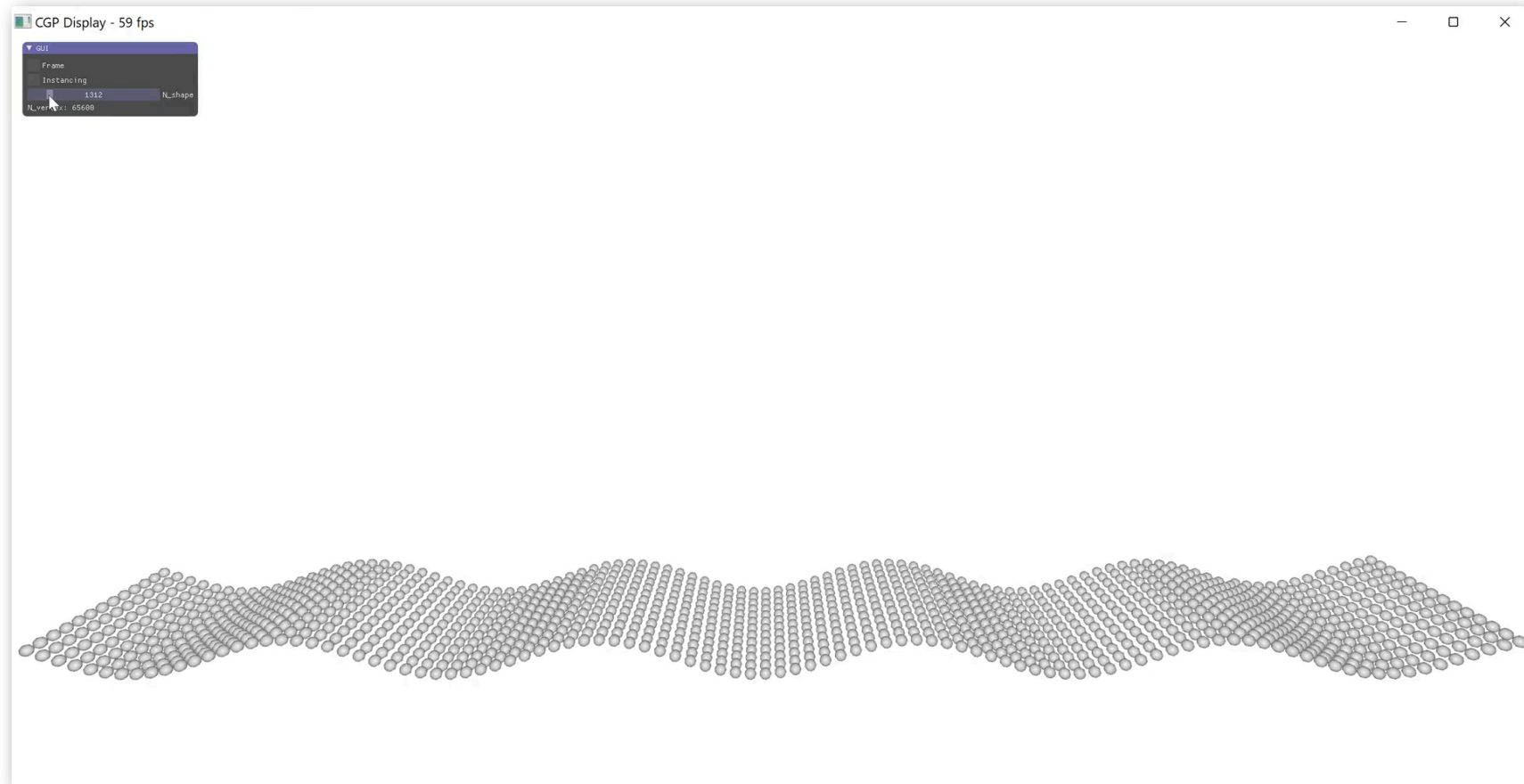
[NVIDIA]

(+) Full capability of your GPU

(-) Require a specific shader

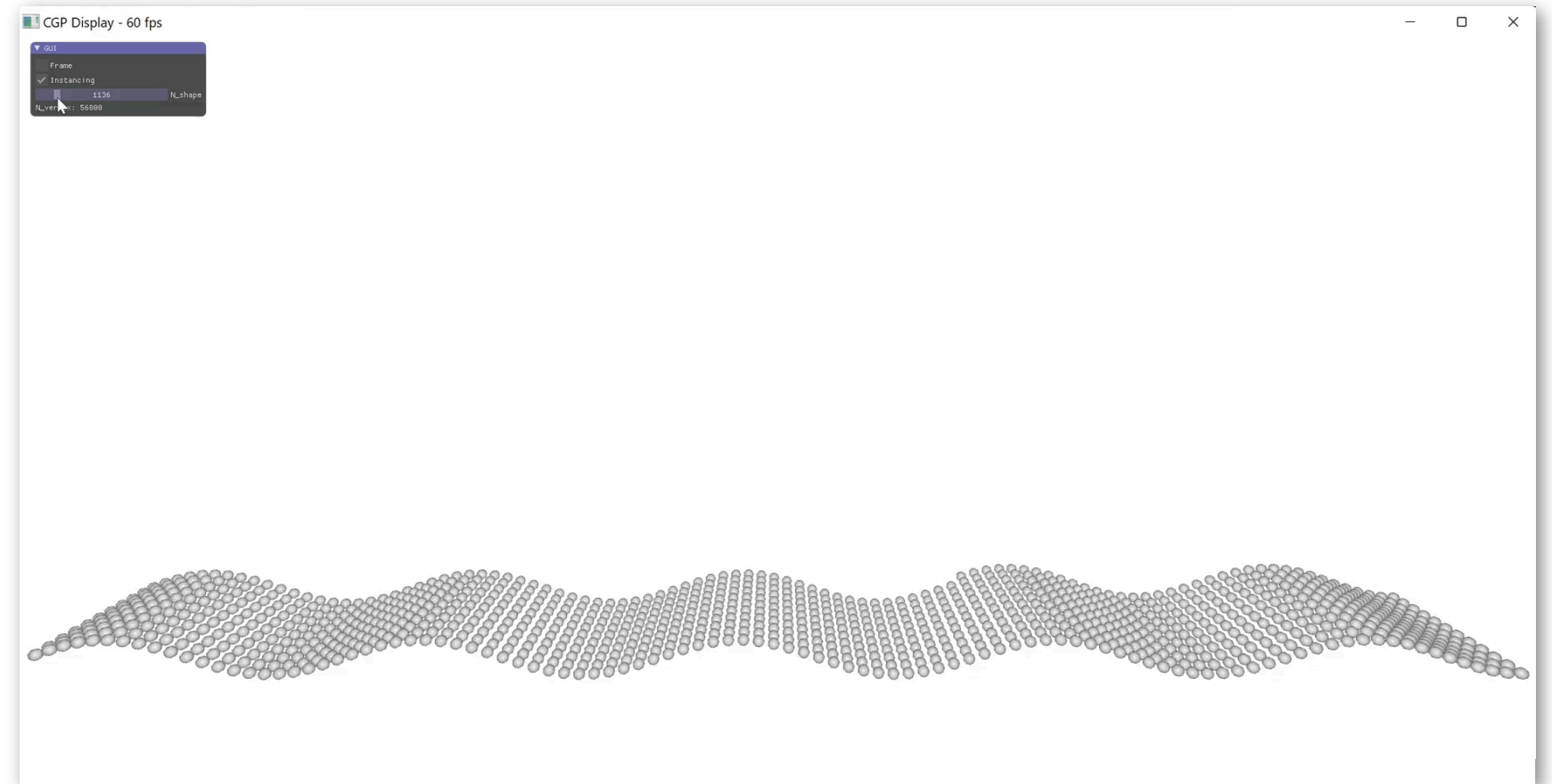
Instancing Demo

Sans Instancing



Lent pour 15,000 particles
#vertices=750,000

Avec Instancing



Affiche 450,000 particles à 60fps (dépend du GPU)
#vertices=22,000,000

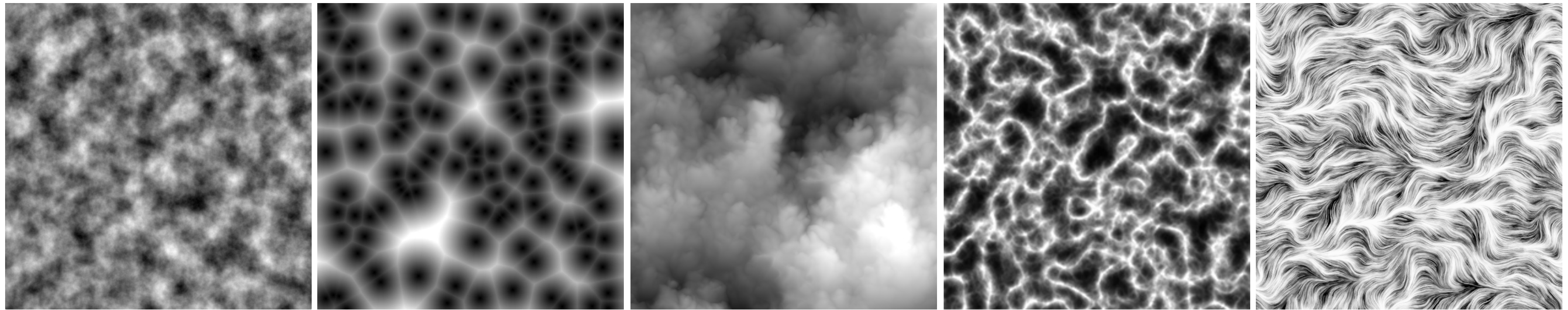
Animations procédurales

Noise function - Perlin Noise

Bruit Procéduraux

Qu'est ce qu'un bruit procédural:

- Fonction avec une structure / pattern visible (bande de fréquence limitée)
- Sans périodicité visible
- Déterministe (Même résultat pour une entrée donnée)



Exemples de textures 2D procédurales

Création d'un bruit procédural

Ex. Fonction continue $f(x)$ avec bande fréquentielle limitée.

Pour des valeurs entières: $f(n) = \text{pseudo-aléatoire, déterministe}$

ex. Fonction de hashage: `float hash(float n) { return fract(sin(n) * 1e4); }`

Interpole les valeurs entières avec une courbe lisse (Smooth step, polynômes cubiques, etc.)

Propriétés:

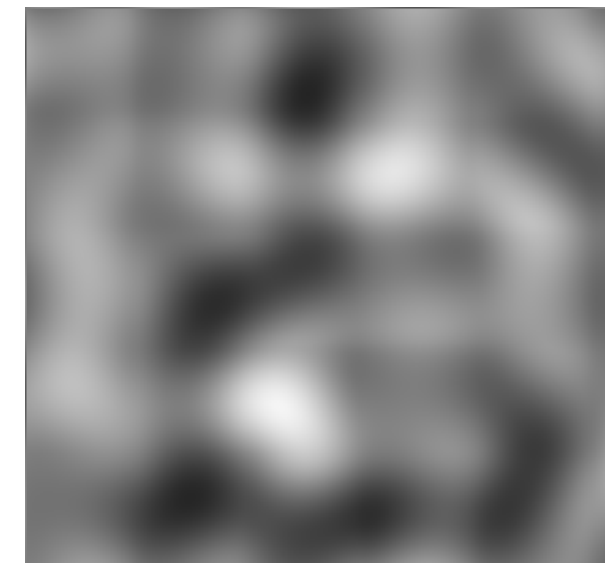
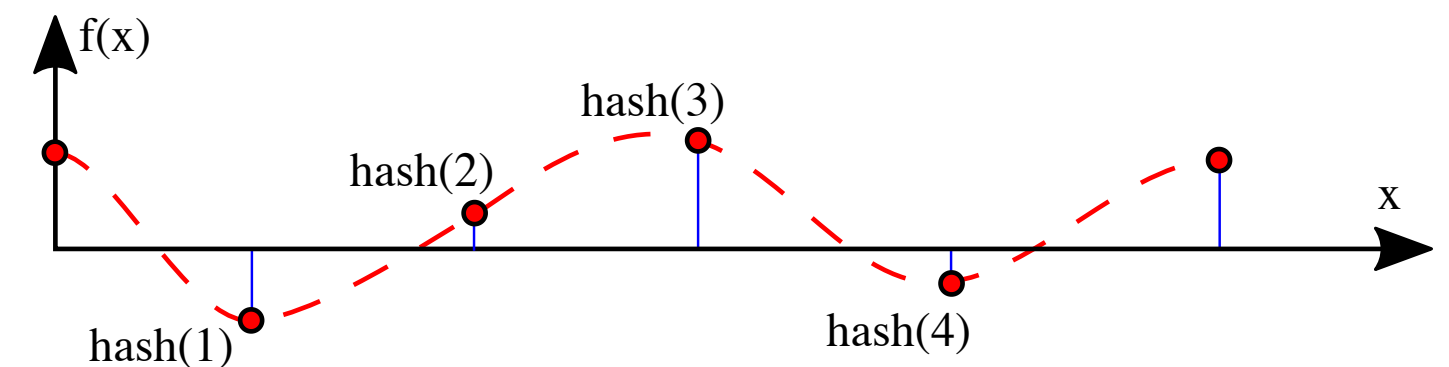
- Fonction continue
- Apparence non périodique
- Fréquence limitée autour de 1 Hz

Peut être calculé en 2D, 3D, etc.

Algorithme simple - ex. en 1D

Pour un x donné, $n = \text{floor}(x)$

- Évaluer la fonction de hashage en $n, n + 1$
($(n \pm i)$ pour des échantillonnages supplémentaires)
- Calculer la valeur interpolée $f(x)$ à la position x

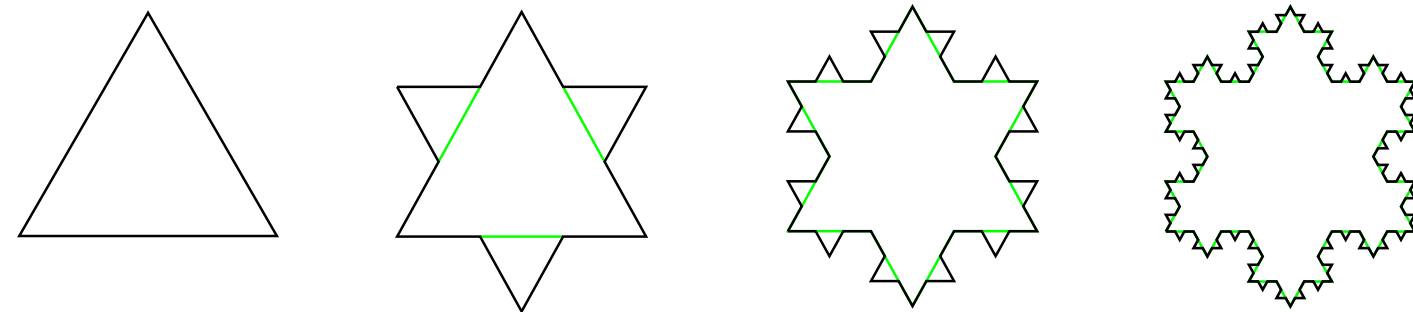


Ajout de détails à haute fréquence: Notion de fractale

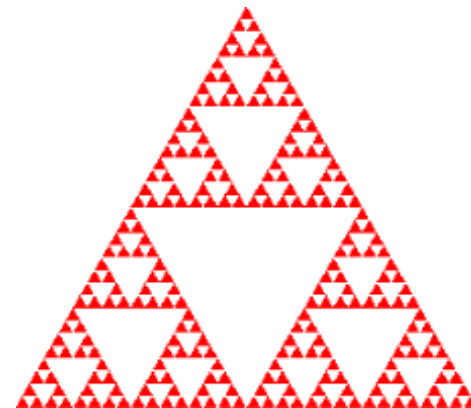
Idée: Ajouter récursivement des détails auto-similaires

Règle simple $\Rightarrow$ formes complexes

Peut ressembler à des détails naturels complexes



Koch Snowflake

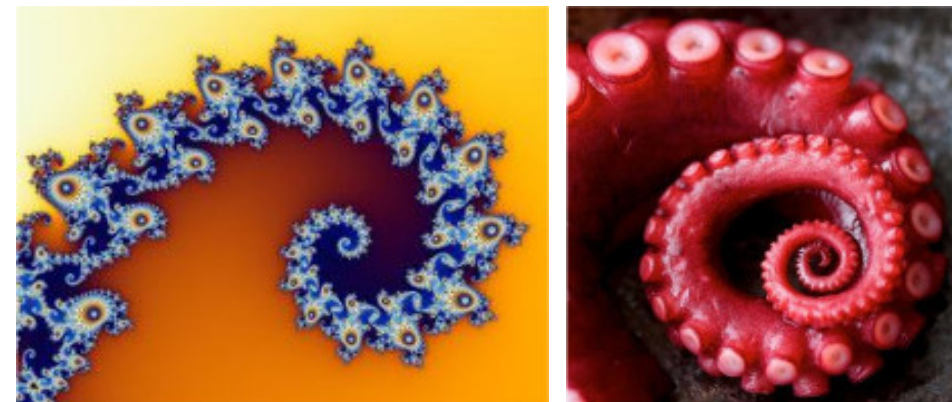


Sierpinski triangle, Shell:Oliva Porphyria



Standard fractales (Mandelbrots, Sierpinski, racines de Newton, etc) difficiles à contrôler.

Rarement utilisées directement en CG.



Bruit de Perlin

Premier bruit procédural proposé par

[Ken Perlin, *An Image Synthesizer*. SIGGRAPH 85]

Idée: Somme de fonctions pseudo-aléatoires avec fréquences croissantes et amplitudes décroissantes

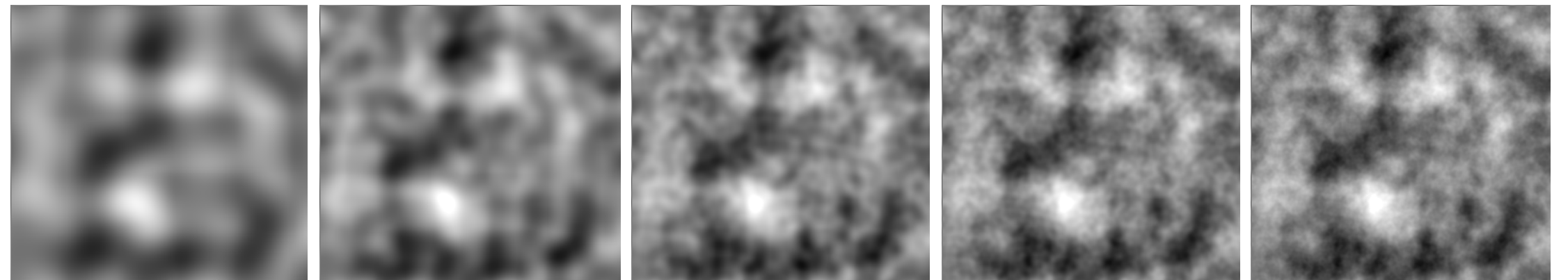
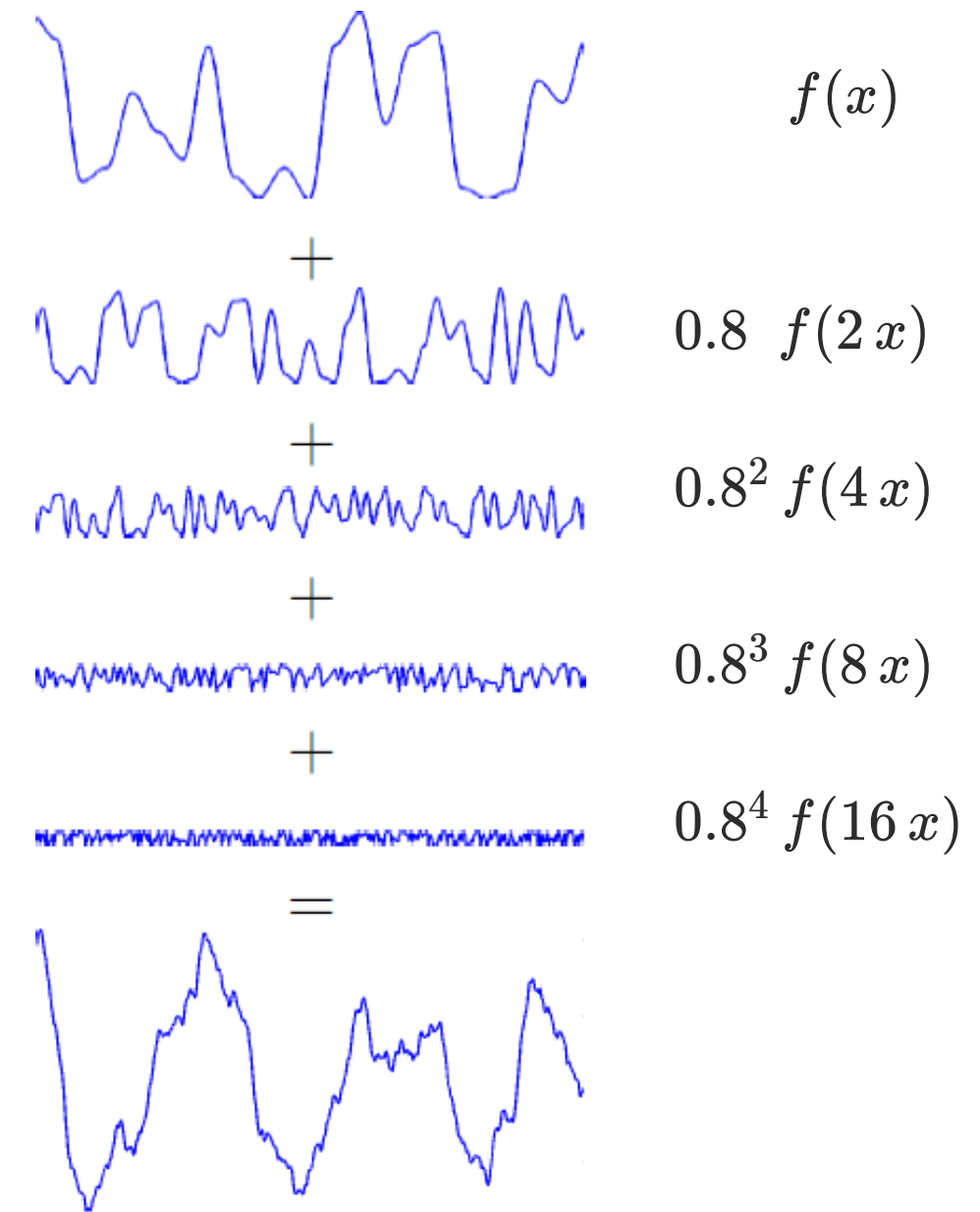
f : Smooth pseudo-random function

$$P(x) = \sum_{k=0}^N \alpha^k f(\omega^k x)$$

- N number of Octave

- α persistency ($1/\alpha$ attenuation)

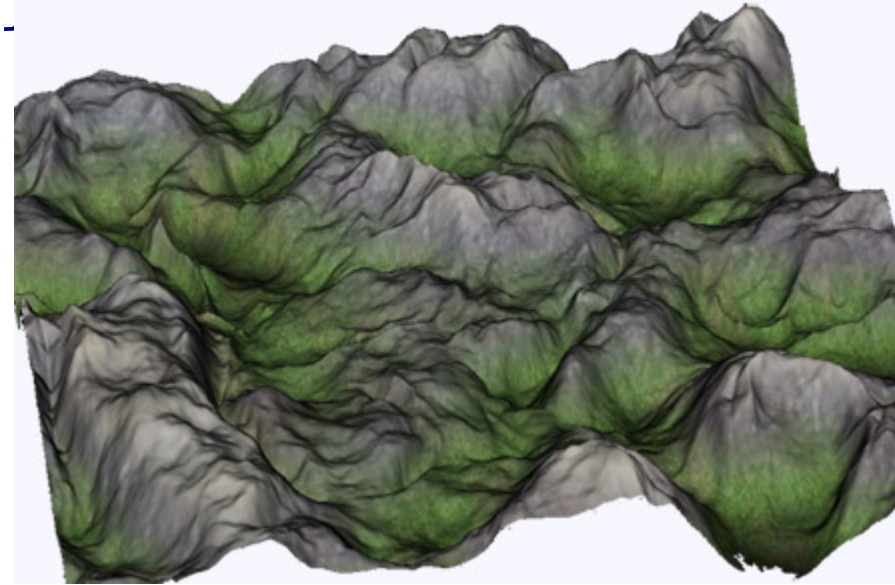
- ω frequency gain



Utilisation du bruit de Perlin

Directe: $z = P(x, y)$

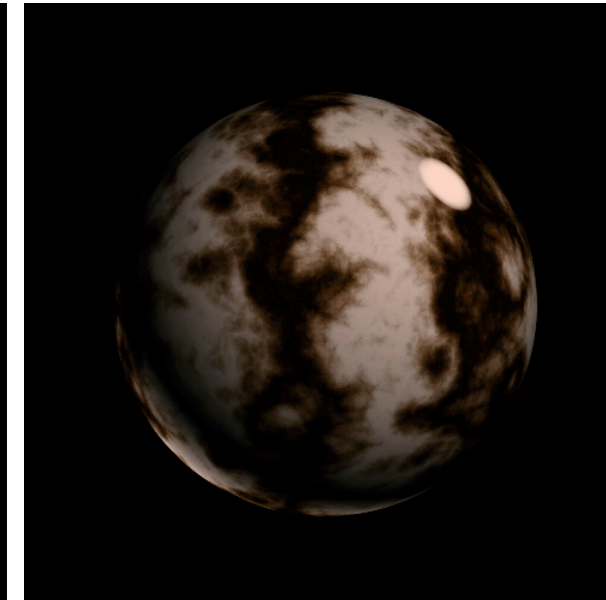
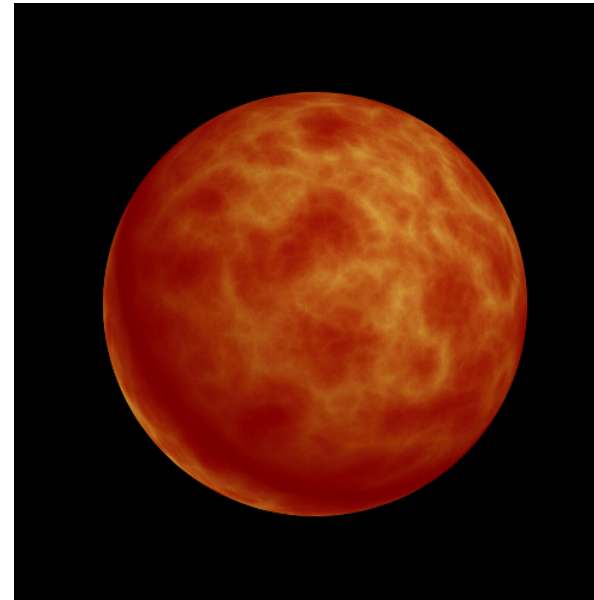
Apparence de montagne



Textures procédurales

Crête (/ Ridge): $\sum_k \alpha^k |f(\omega^k x)|$

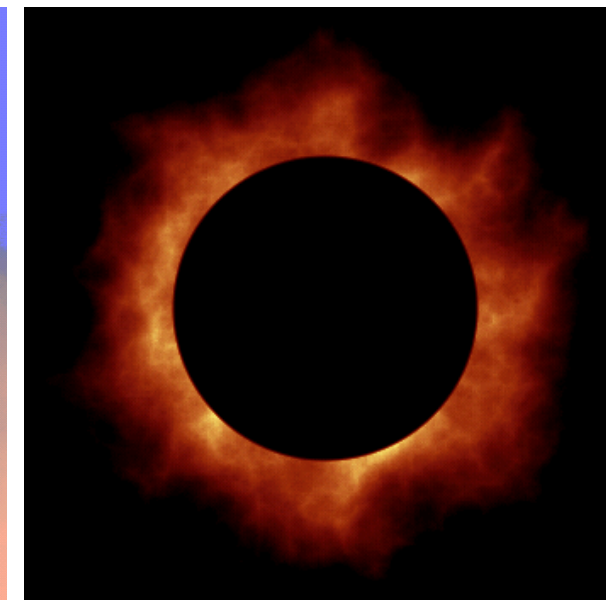
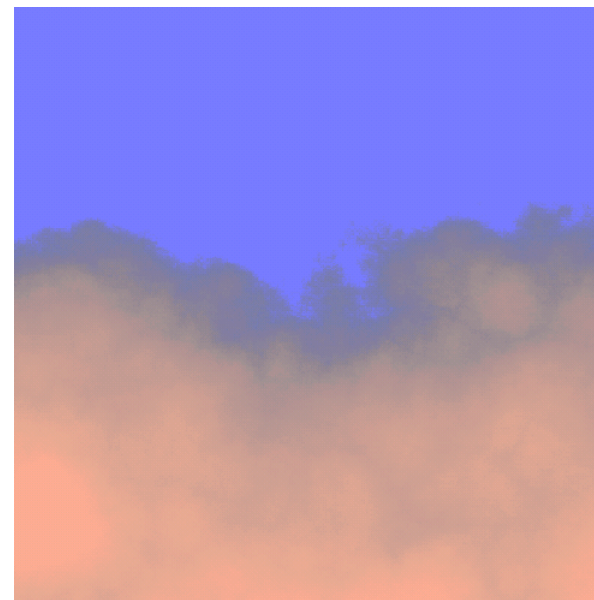
Marbre: $\sin(x + \sum_k \alpha^k |f(\omega^k x)|)$



Textures animées

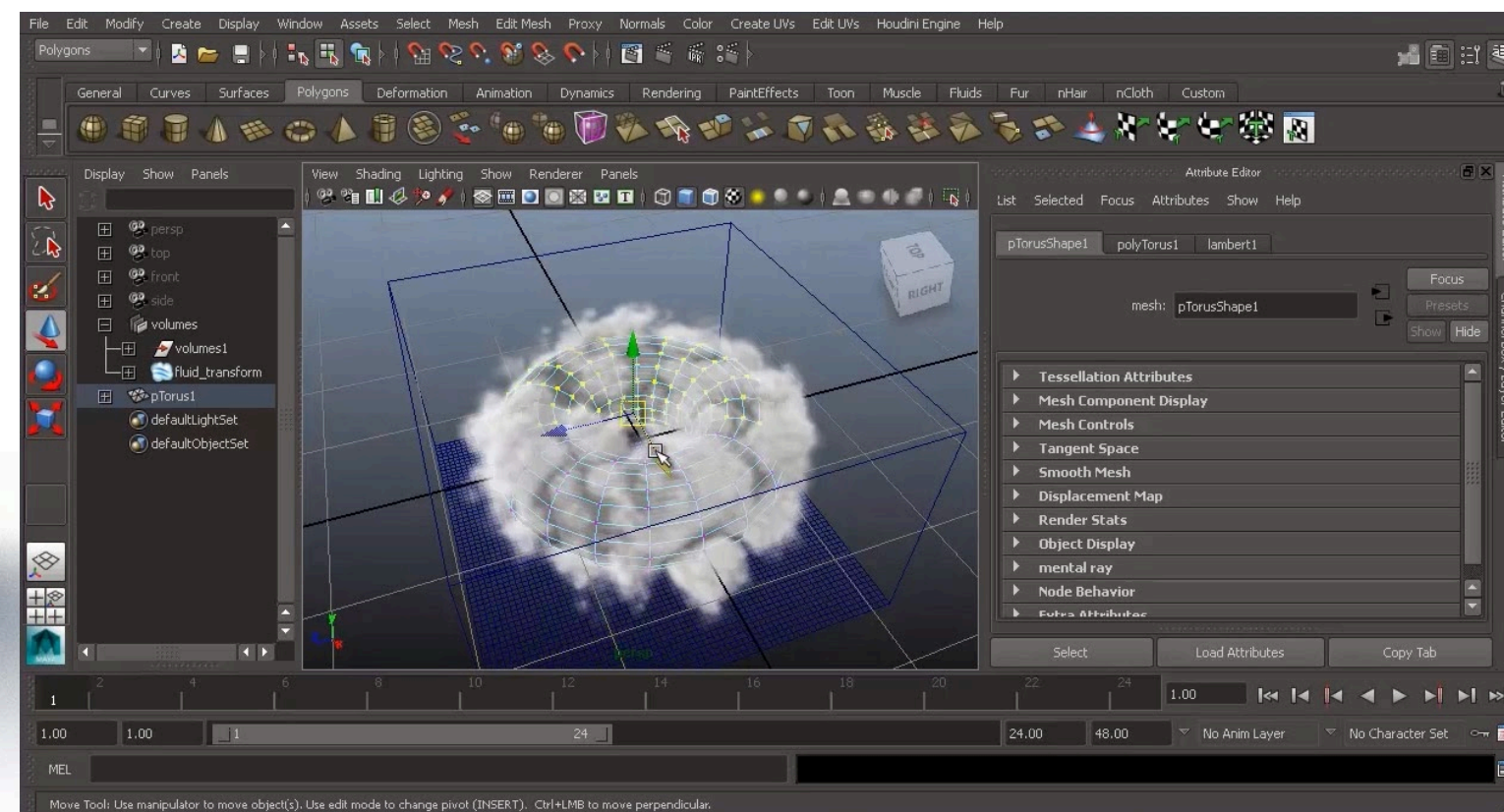
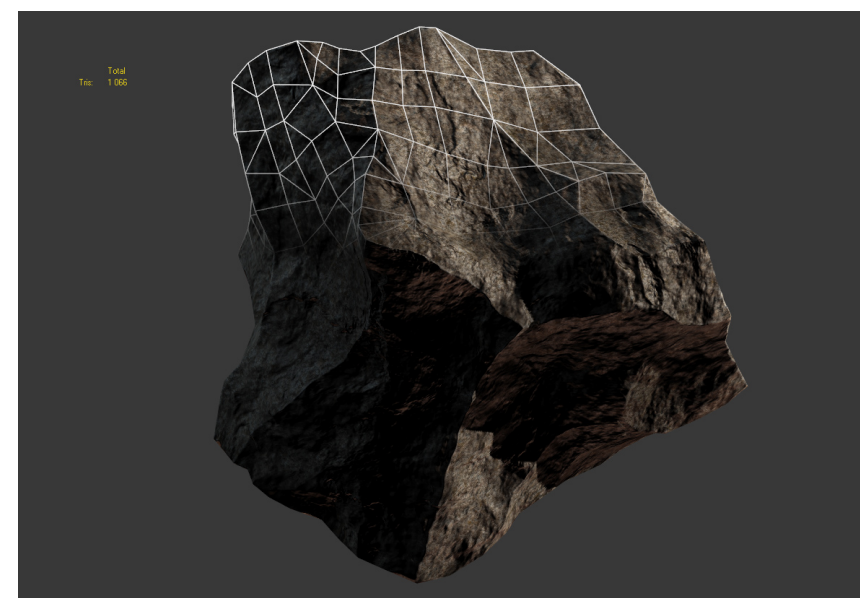
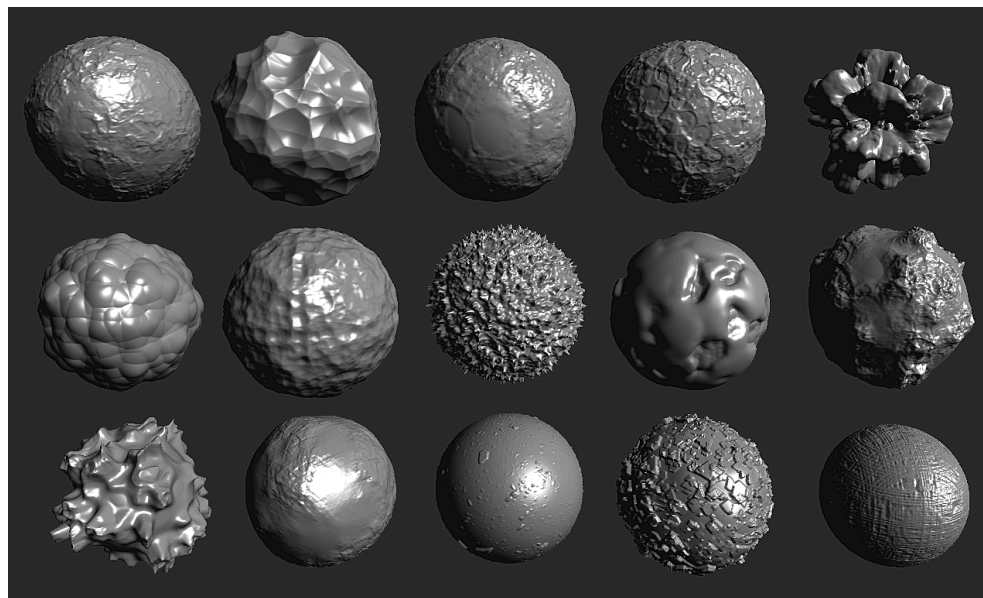
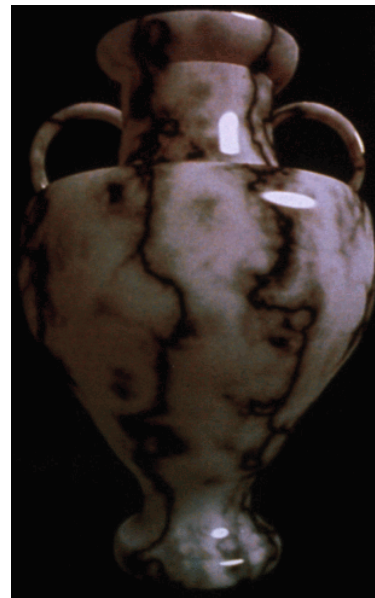
Translation: $P(x, y + t)$

Evolution lisse: $P(x, y, t)$



Applications du bruit de Perlin

Pour quasiment toutes les formes complexes ...



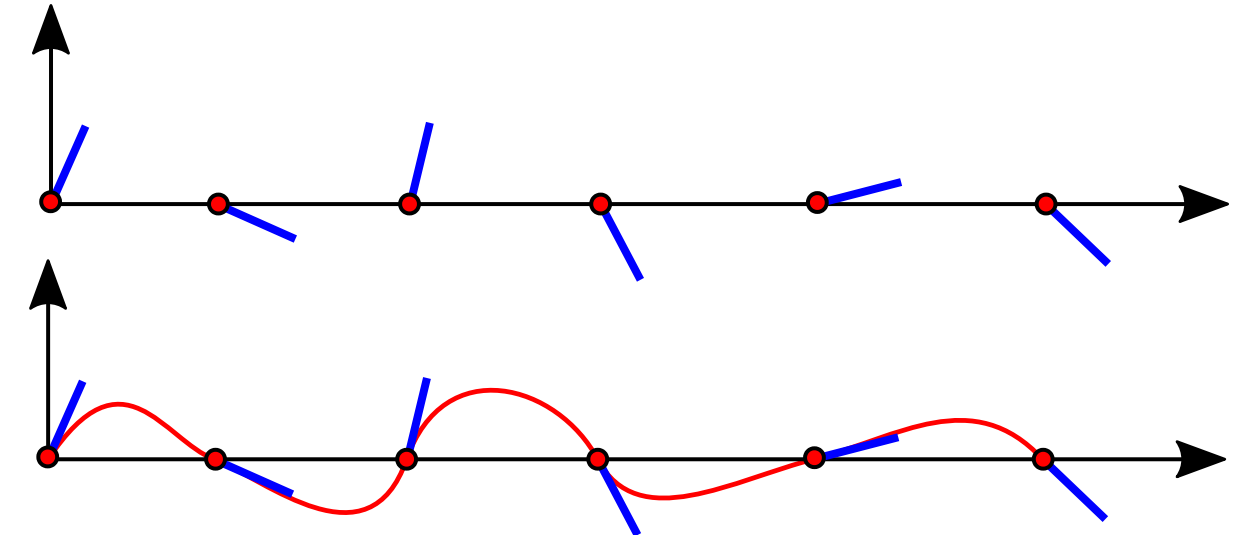
Look at [Shader Toy](#) + Noise [example](#)

Bruit de Perlin: Améliorations

Gradient Noise

Utilisation de gradients pseudo-aléatoires au lieu de positions

Contrôle de la fréquence amélioré: Oscillation à période 1



Simplex Noise

Interpolation dans des polytopes simples à la place d'une interpolation sur grille.

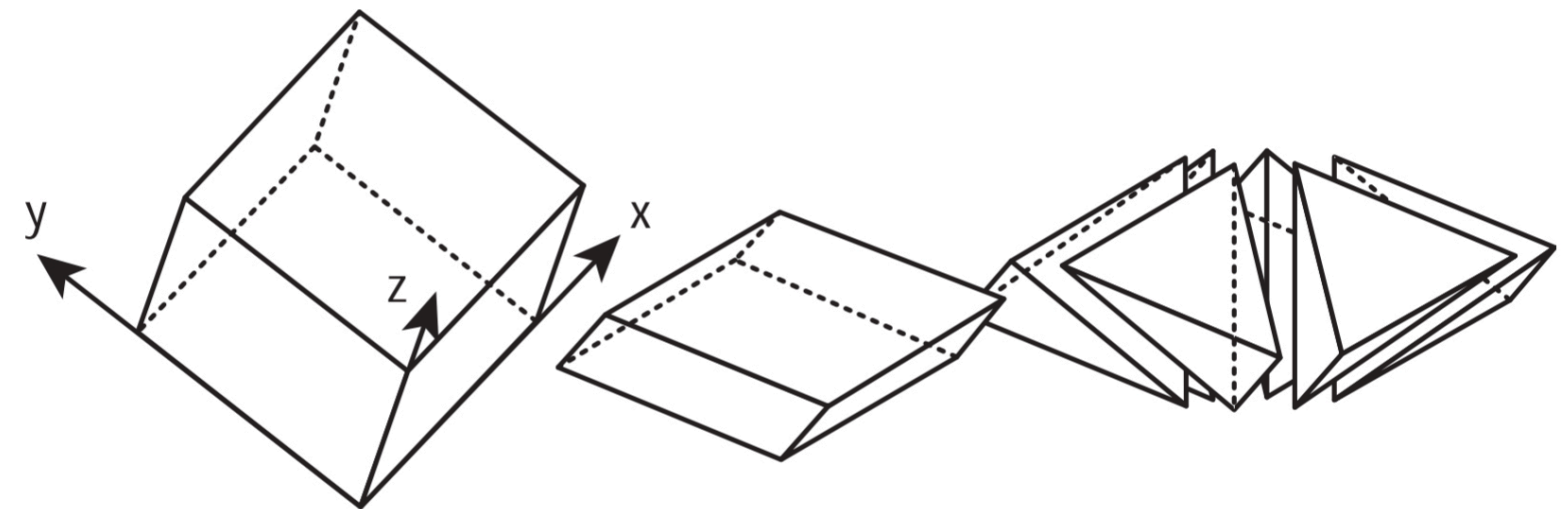
Evite les artefacts de direction de grille

Plus rapide pour des dimensions élevées

[Stefan Gustavson Simplex noise demystified]

Pour aller plus loin:

[A Lagae et al., STAR in Procedural Noise Functions. EG 2010]



Bruit de Perlin: Terrain

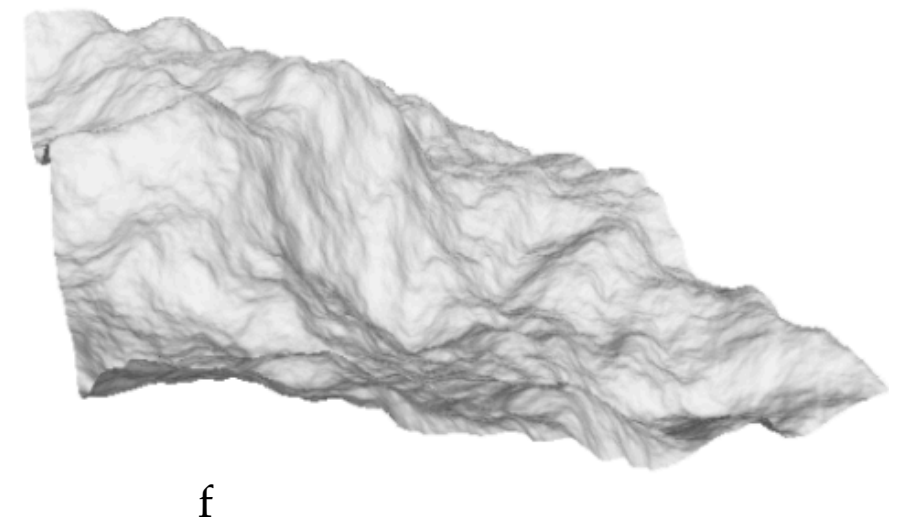
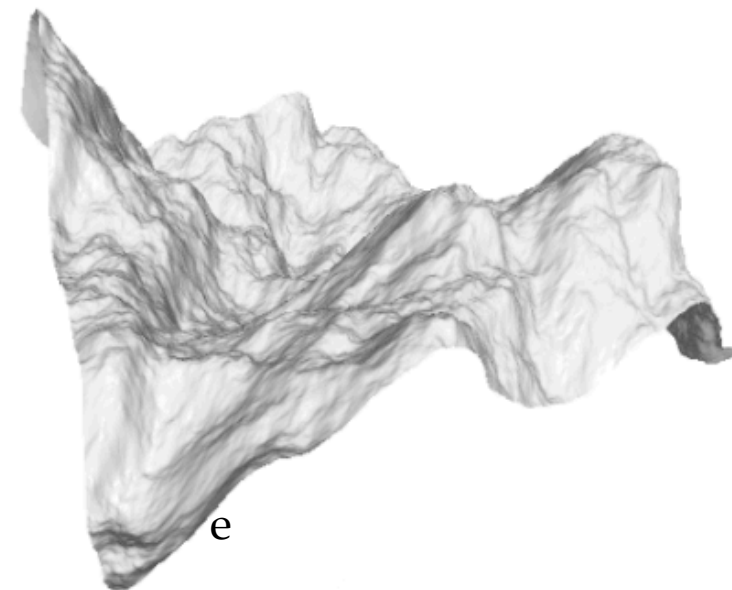
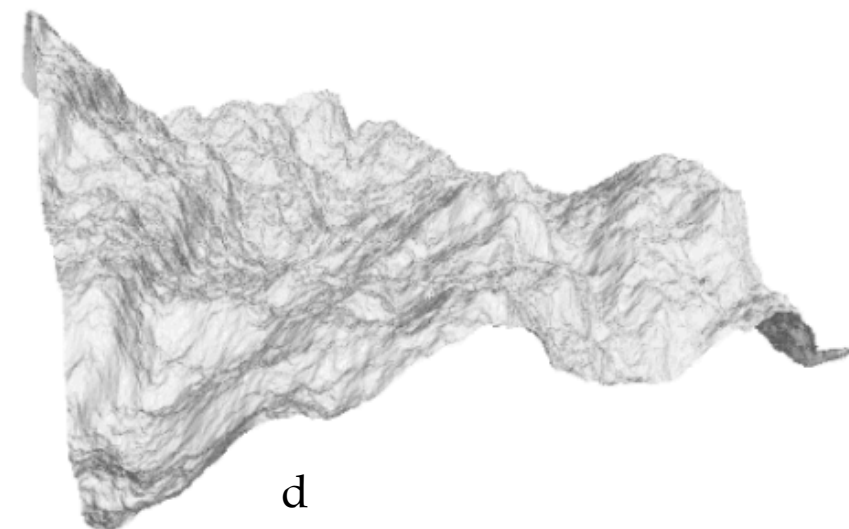
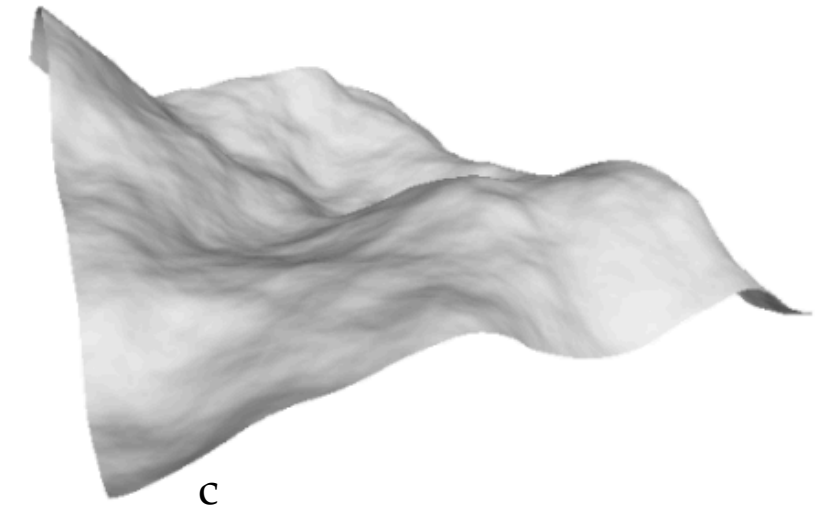
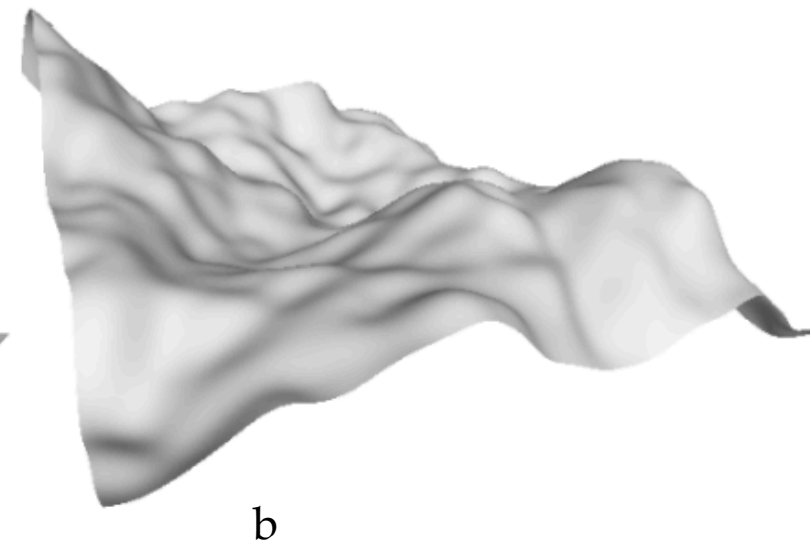
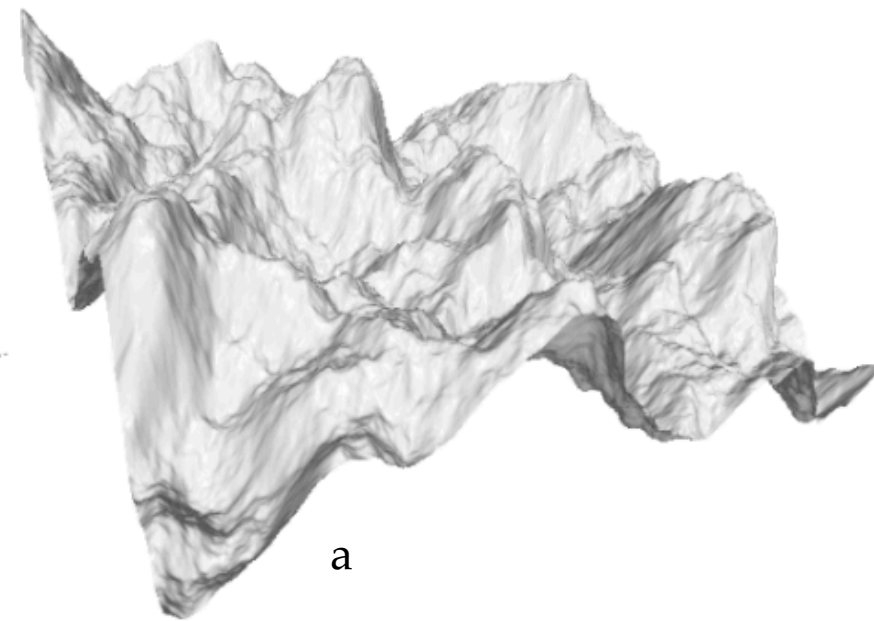
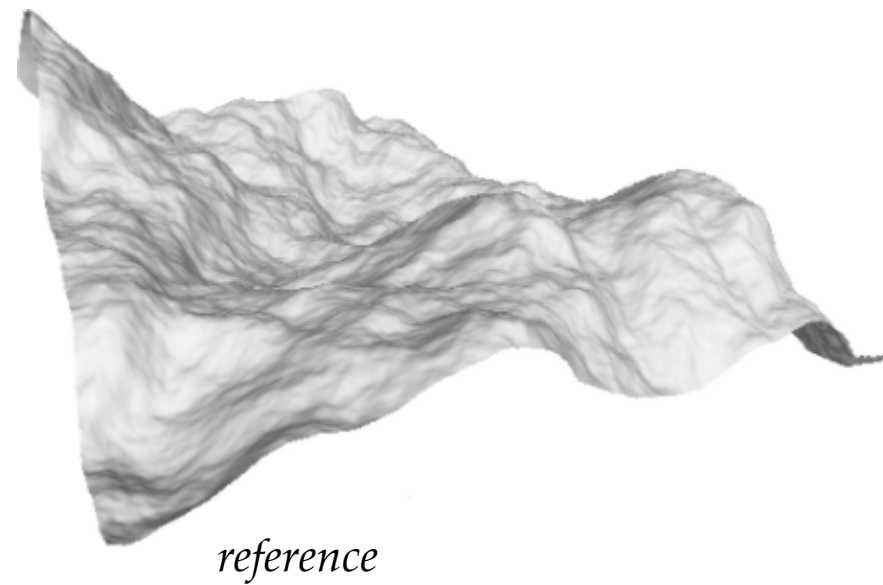
On considère la surface S définie par:

$$S(u, v) = \begin{cases} x(u, v) = u \\ y(u, v) = v \\ z(u, v) = h P(s(u + o), s(v + o)) \end{cases}$$

Le bruit de Perlin

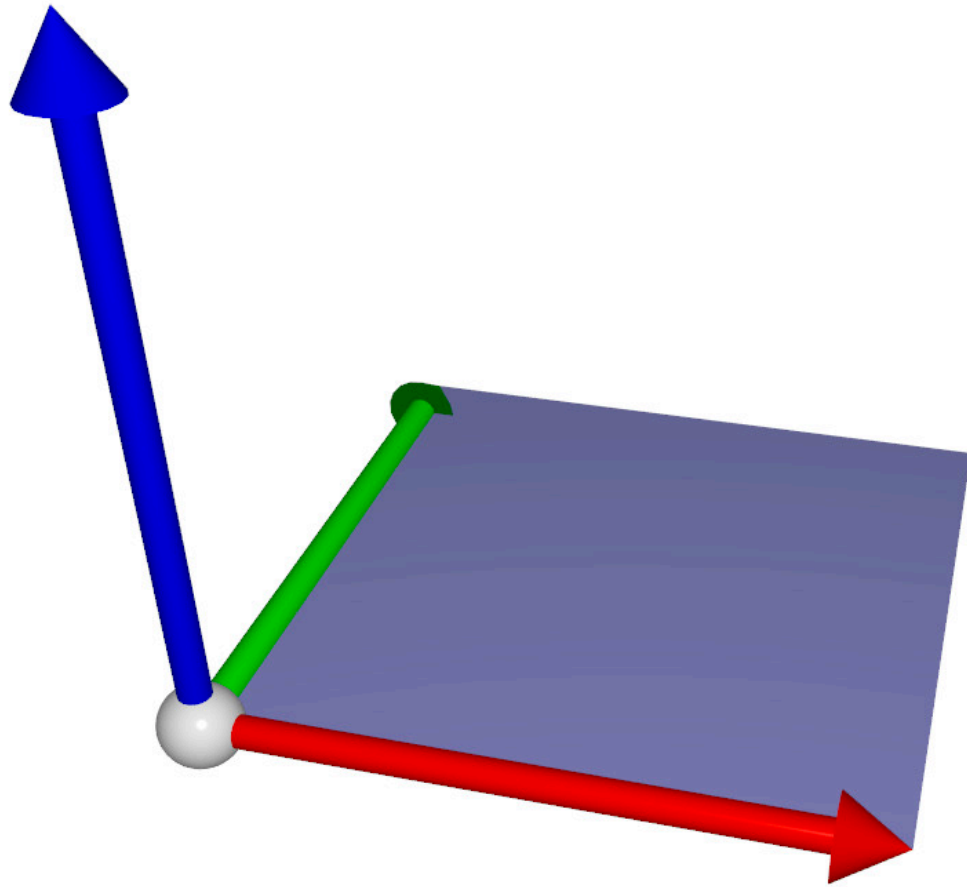
$$P(u, v) = \sum_{k=0}^N \alpha^k f(2^k u, 2^k v)$$

$N = 9$
 $\alpha = 0.4$
 $h = 0.3$
 $s = 1$
 $o = 0$



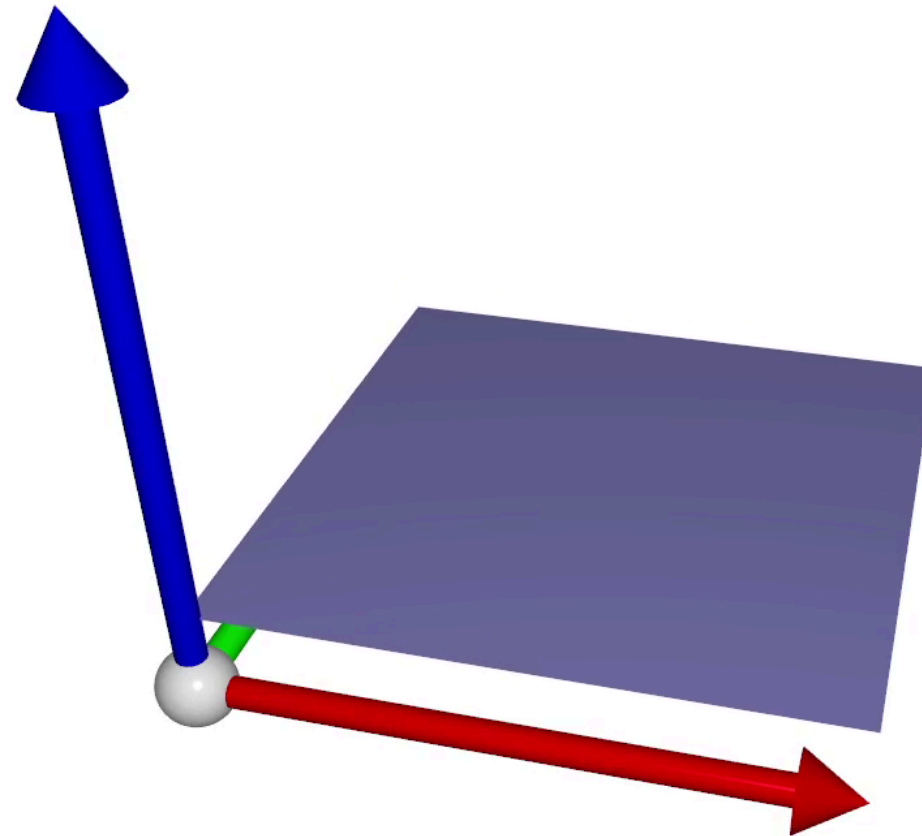
Q. Quels paramètres correspondent
aux terrains (a,b,c,d,e,f) ?

Perlin Noise - Animation



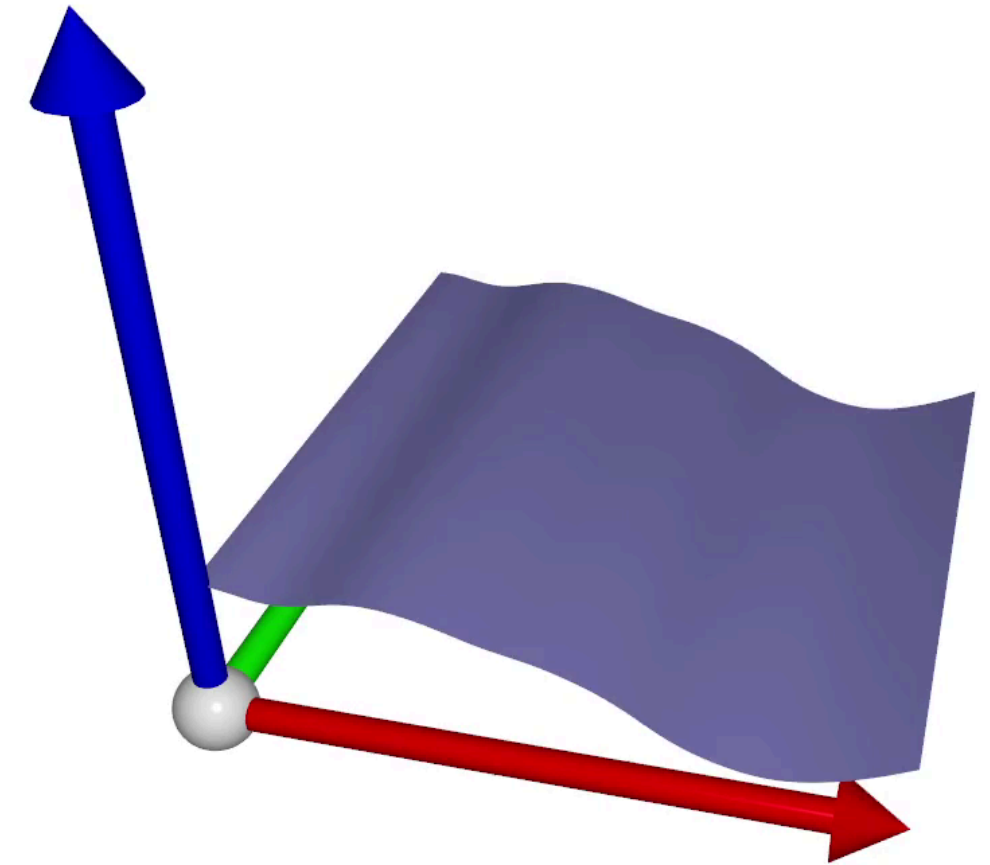
Fonction de la surface de référence

$$(u, v) \in [0, 1]^2, S(u, v) = (u, v, 0)$$



Bruit de perlin dépendant du temps

$$S(u, v, t) = (u, v, P(t))$$



Dépendance à l'espace et au temps

$$S(u, v, t) = (u, v, P(u, t))$$

Bruit de Perlin - Animation

Surface de référence: $(u, v) \in [0, 1]^2$, $f(u, v) = (u, v, 0)$
Q. Comment générer les animations suivantes ? $S(u, v, t) = \dots$

