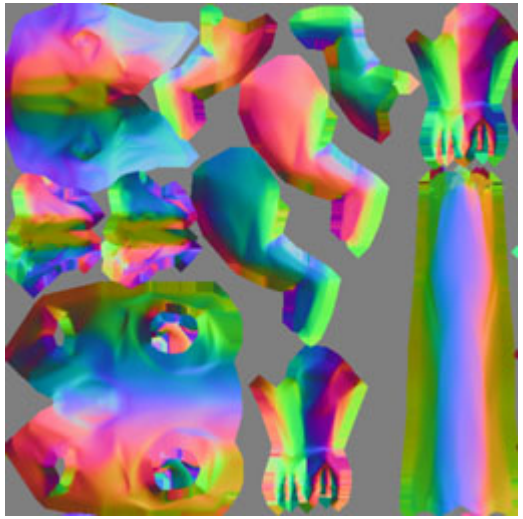
Normal mapping

+



=

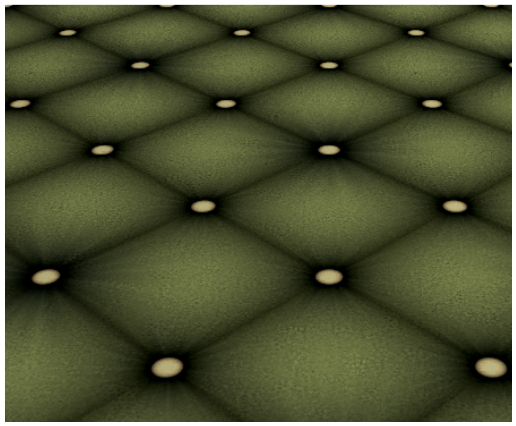
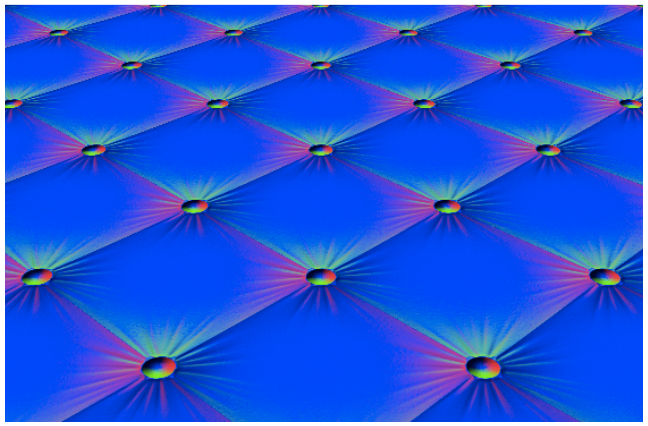
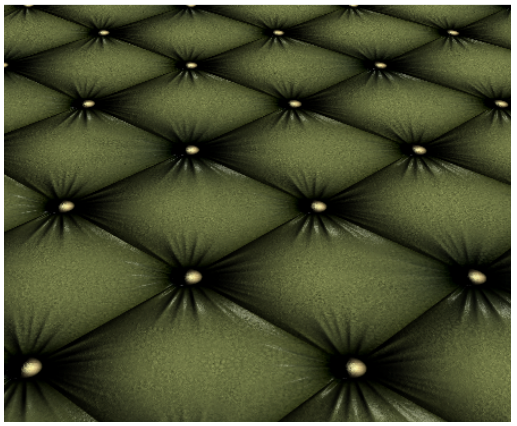
Crédits :

Gaël Guennebaud, Joëlle Thollot, Marco Salvi,
Akenine-Möller, Haines & Hoffman (« *Real-Time Rendering* », AK Peters)

Normal mapping

Idée générale

- représenter les **micro-détails** sous forme d'une **texture de normales** représentant les variations de la surface détaillée
- utiliser un maillage de basse résolution pour la rasterisation
- utiliser la carte de normales lors du shading **par fragment**

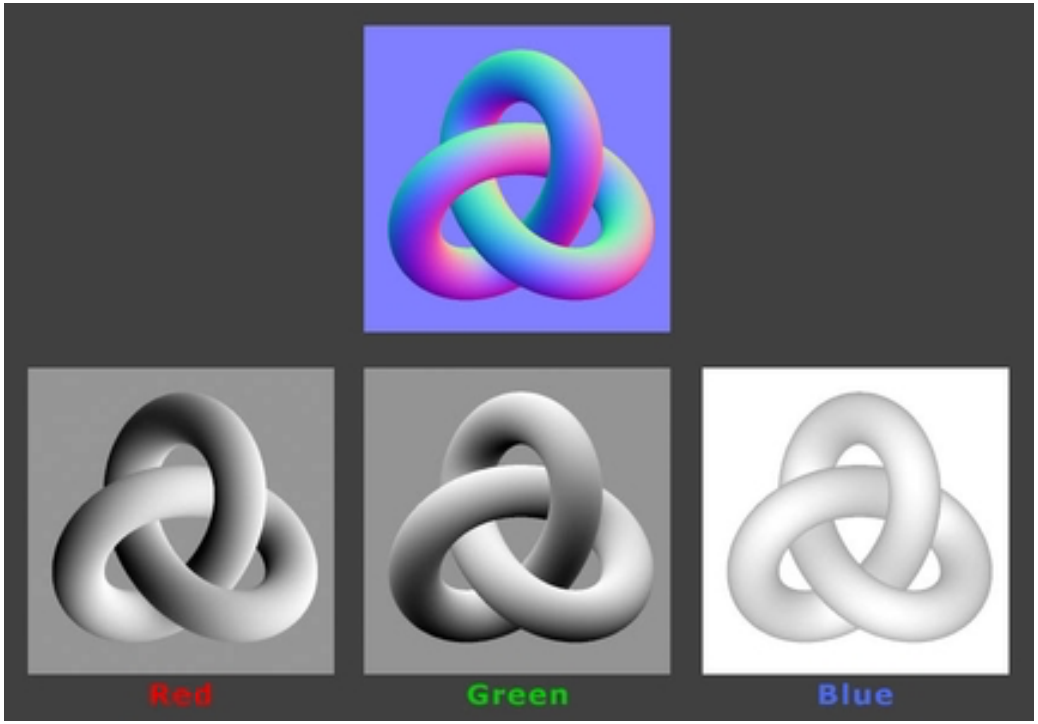




Stockage

Dans une image RGB

$$R = N_x, G = N_y, B = N_z$$

Mapping de [-1,1] ⟶ [0,1]

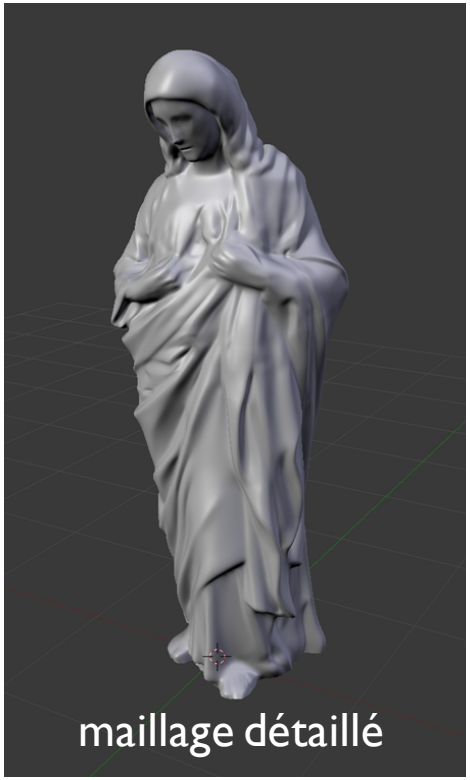


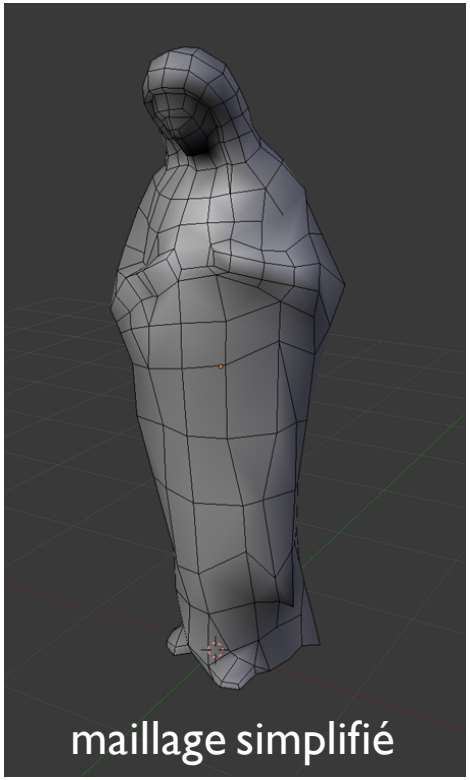
Images par Eric Chadwick

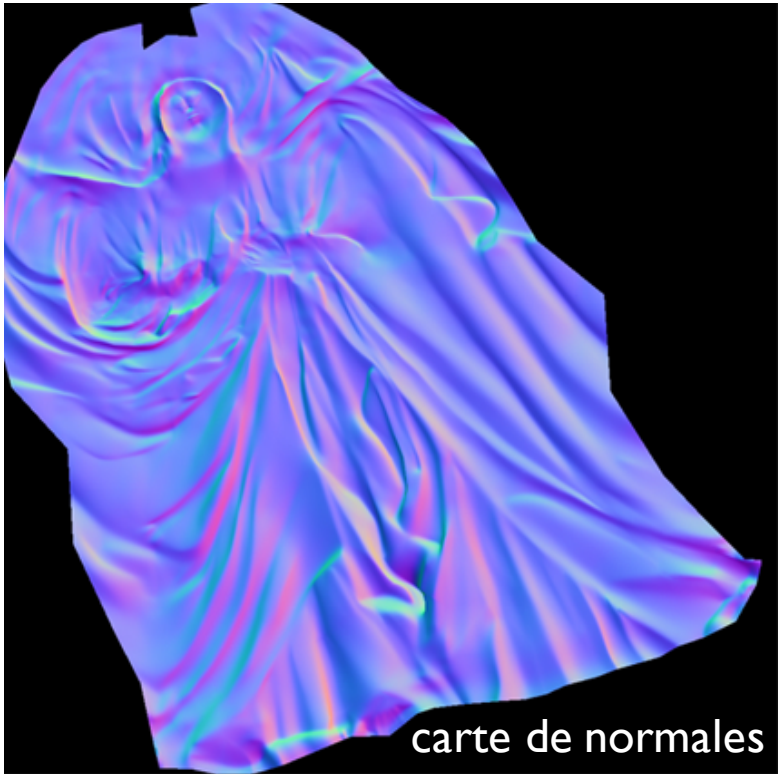
Calcul d'une carte de normales

À partir d'un maillage 3D très détaillé

1. projeter et rastériser le maillage dans la texture
2. calculer et stocker les normales en **espace objet** ou **tangent**







Images : Normal Map Baking in Blender 2.5 by Jonathan Williamson

Calcul d'une carte de normales

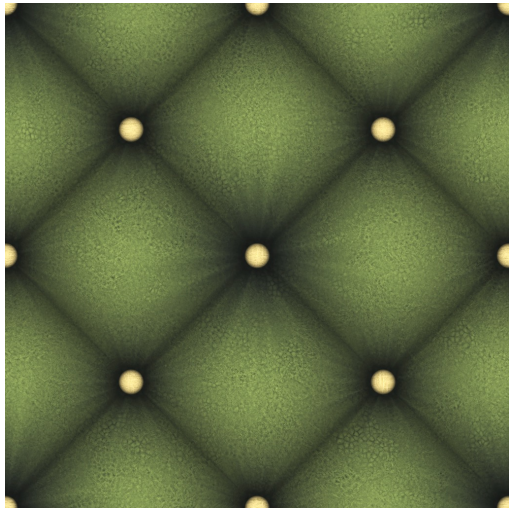
démo CrazyBump

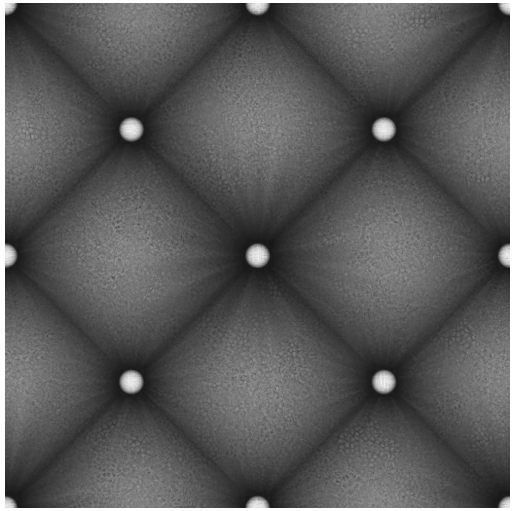
À partir d'une image

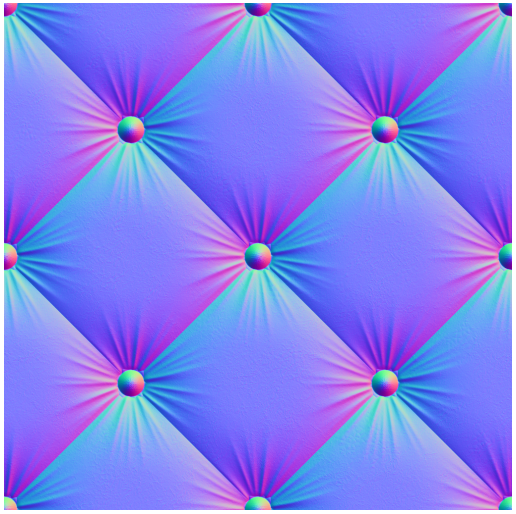
1. convertir l'image en champ d'élévation (niveaux de gris)
2. calculer les normales du champ d'élévation (gradient)

Attention : les normales sont représentées dans l'espace de la texture, c.-à-d. dans l'**espace tangent**

⟹ **transformation des normales dans le repère de l'objet avant de réaliser les calculs d'éclairage !**





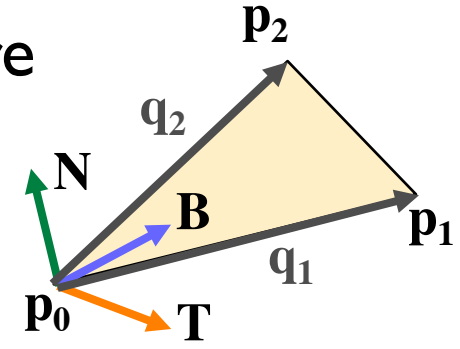


Calcul du repère tangent

Pour un triangle, repère tangent défini par :

$\mathbf{T}$: tangente, $\mathbf{B}$: cotangente, $\mathbf{N}$: normale

Un point $\mathbf{q}(s,t)$ du triangle $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ peut être définie comme : $\mathbf{q} = (s\text{-}s_0)\,\mathbf{T} + (t\text{-}t_0)\,\mathbf{B}$



Soient $\mathbf{q}_1, \mathbf{q}_2$ tels que : $\mathbf{q}_1 = \mathbf{p}_1 - \mathbf{p}_0$

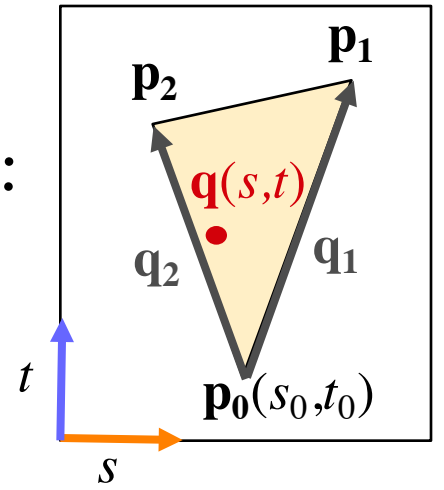
$\mathbf{q}_2 = \mathbf{p}_2 - \mathbf{p}_0$

On veut $\mathbf{T}, \mathbf{B}, \mathbf{N}$ tels que : $\mathbf{q}_1 = s_1\,\mathbf{T} + t_1\,\mathbf{B}$

$\mathbf{q}_2 = s_2\,\mathbf{T} + t_2\,\mathbf{B}$

$\Longrightarrow$ 6 équations, 6 inconnues (3 pour $\mathbf{T}$ et $\mathbf{B}$) :

$$\begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{T} & \mathbf{B} \end{pmatrix} \begin{pmatrix} s_1 & s_2 \\ t_1 & t_2 \end{pmatrix}$$



Calcul du repère tangent

$$\begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{T} & \mathbf{B} \end{pmatrix} \begin{pmatrix} s_1 & s_2 \\ t_1 & t_2 \end{pmatrix}$$

$$\Leftrightarrow \quad \begin{pmatrix} \mathbf{T} & \mathbf{B} \end{pmatrix} = \begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 \end{pmatrix} \begin{pmatrix} s_1 & s_2 \\ t_1 & t_2 \end{pmatrix}^{-1}$$

$$\Leftrightarrow \quad \begin{pmatrix} \mathbf{T} & \mathbf{B} \end{pmatrix} = \frac{1}{s_1 t_2 - s_2 t_1} \begin{pmatrix} \mathbf{q}_1 & \mathbf{q}_2 \end{pmatrix} \begin{pmatrix} t_2 & -s_2 \\ -t_1 & s_1 \end{pmatrix}$$

$$A^{-1} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}^{-1} = \frac{1}{(ad - bc)} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

http://www.terathon.com/code/tangent.html

Calcul du repère tangent

Pour un sommet, moyenne des repères tangents des faces adjacentes (comme pour les normales)

$\Longrightarrow$ **Ortho-normalisation** du repère en négligeant les déformations anisotropes (Gram-Schmidt)

$$\mathbf{T'} = \mathbf{T} - (\mathbf{N} \bullet \mathbf{T})\,\mathbf{N}$$

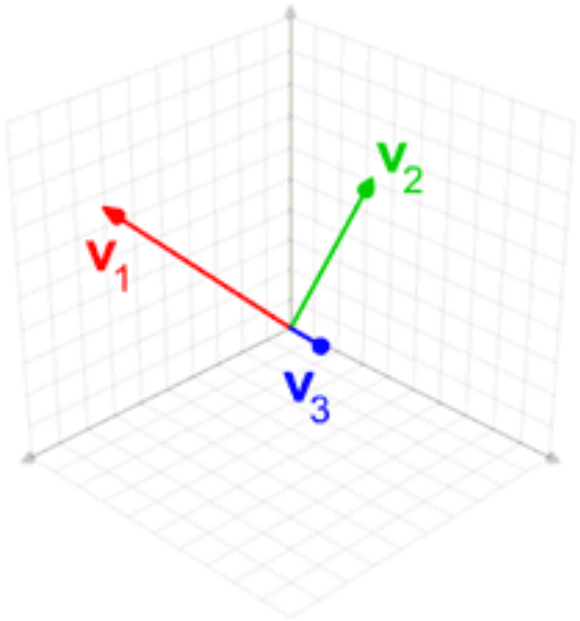
$$\mathbf{B'} = \mathbf{B} - (\mathbf{N} \bullet \mathbf{B})\,\mathbf{N} - (\mathbf{T'} \bullet \mathbf{B})\mathbf{T'}/\|\mathbf{T'}\|$$

…puis re-normaliser

Remarque : attention aux discontinuités des coordonnées de texture

Calcul du repère tangent

Ortho-normalisation de Gram-Schmidt



Wikipedia

Calcul du repère tangent

Passage du repère tangent au repère local de l'objet :

$$\begin{pmatrix} x & y & z \end{pmatrix} = \begin{pmatrix} \mathbf{T}' & \mathbf{B}' & \mathbf{N} \end{pmatrix} \begin{pmatrix} s \\ t \\ r \end{pmatrix}$$

Utilisation du repère tangent

Au chargement de l'objet :

- calcul des repères tangents par sommet
- on peut ne stocker que $\mathbf{N}$, $\mathbf{T}'$ et m :

  $\mathbf{B}' = m\ (\mathbf{N} \times \mathbf{T}')$

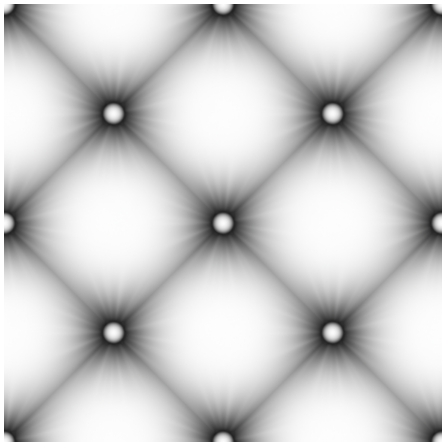
  avec $m = \pm 1 = \det(\mathbf{T}'\ \mathbf{B}'\ \mathbf{N})$

Utilisation du repère tangent

Au moment du rendu :

1. envoi des repères tangents ($\mathbf{N}$, $\mathbf{T}'$ et m) au vertex shader
2. interpolation des vecteurs par le *rasterizer*
3. re-normalisation des vecteurs dans le fragment shader
4. calcul de $\mathbf{B}'$
5. lecture de la carte des normales
6. *remapping* de la normale de [0,1] ⟶ [-1,1]
7. transformation de la normale du repère tangent vers le repère local, puis vers celui de la vue
8. calcul de l'éclairement

Bump vs. Normal Map



Bump mapping

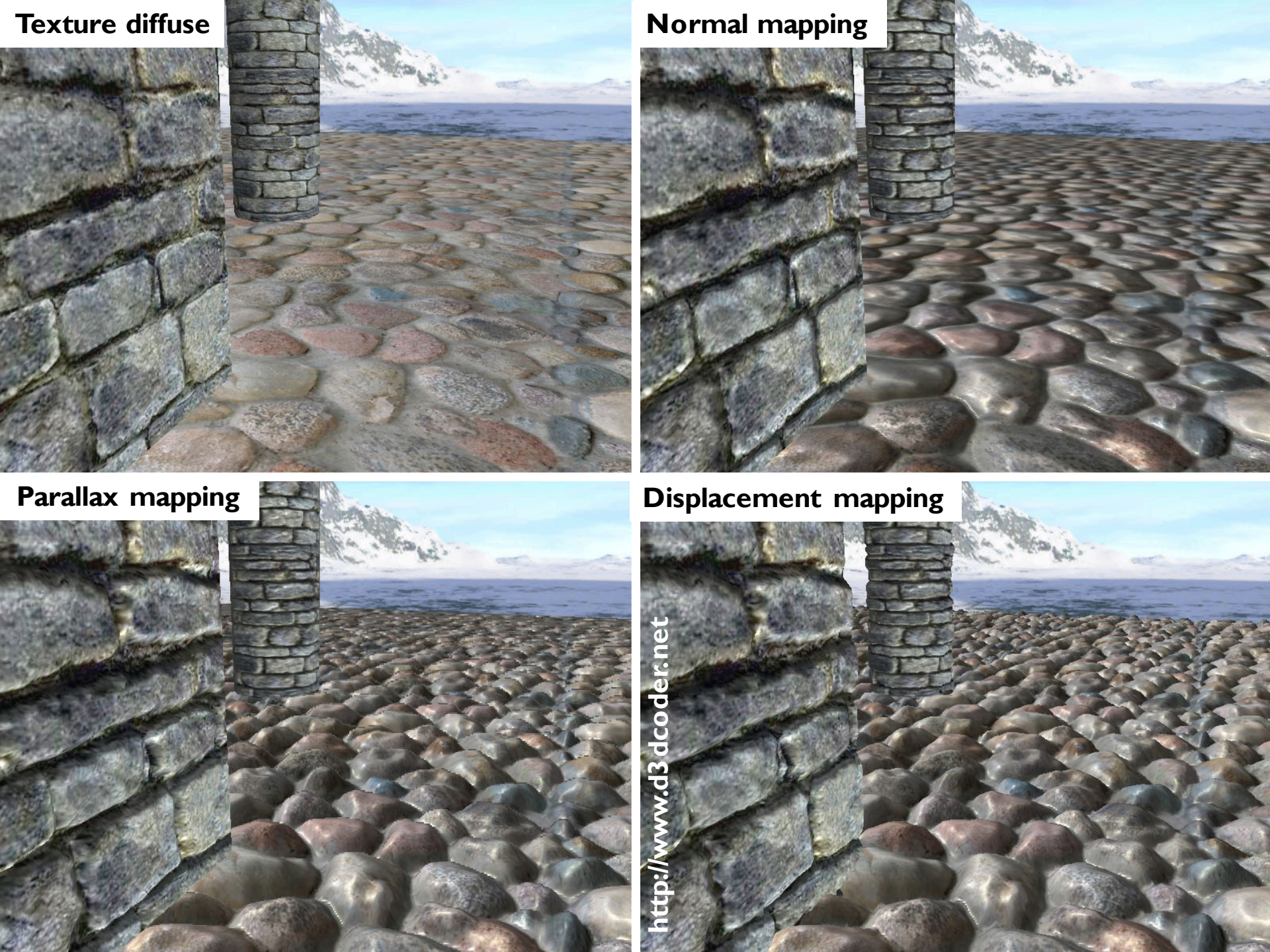
- Stockage de la **hauteur de la surface**
- Calcul du gradient à la volée pour obtenir le déplacement dans le plan tangent

Normal mapping

- Stockage direct des **normales perturbées**

Extensions

- *Parallax mapping* : déplacement des coordonnées de texture
- *Displacement mapping* : déplacement de la géométrie


Texture diffuse
Normal mapping
Parallax mapping
Displacement mapping
http://www.d3dcoder.net

Réflexions

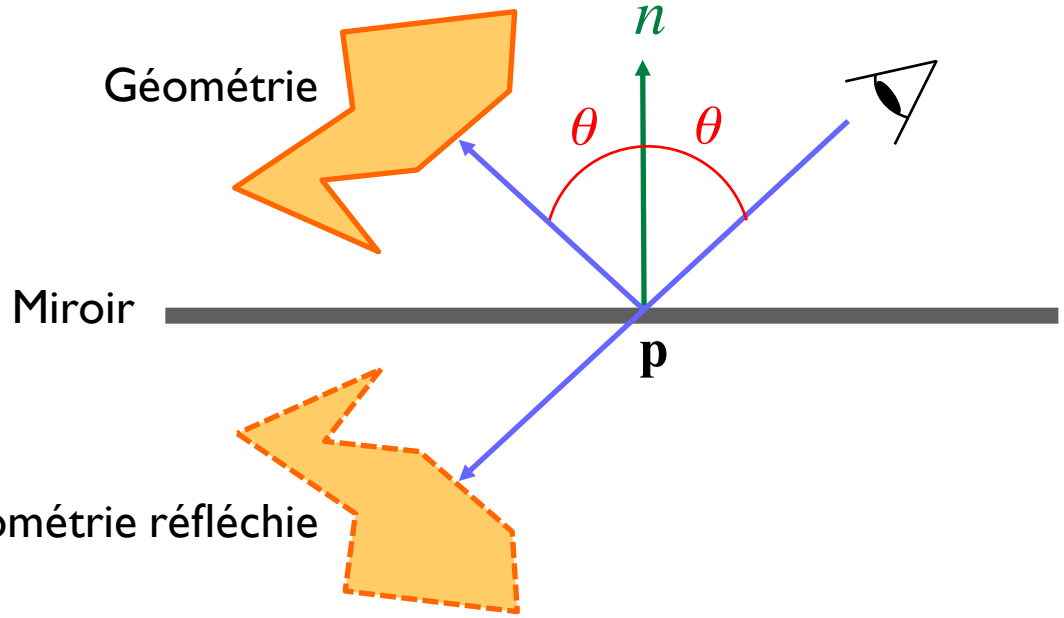
[Haeberli & Segal 1993]

Réflexions planaires – miroir

Miroir = surface planaire réfléchissante

Tracer la scène une deuxième fois,
réfléchie par rapport au plan du miroir

- Calculer la **matrice de réflexion** $\mathbf{M} =$

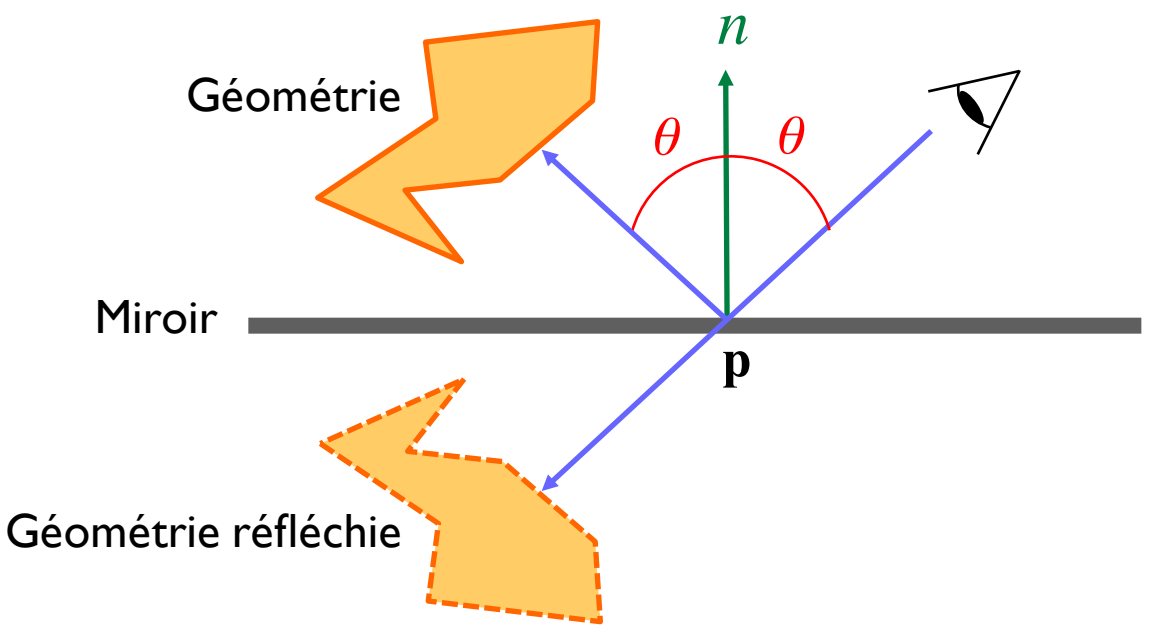


Réflexions planaires – miroir

Miroir = surface planaire réfléchissante

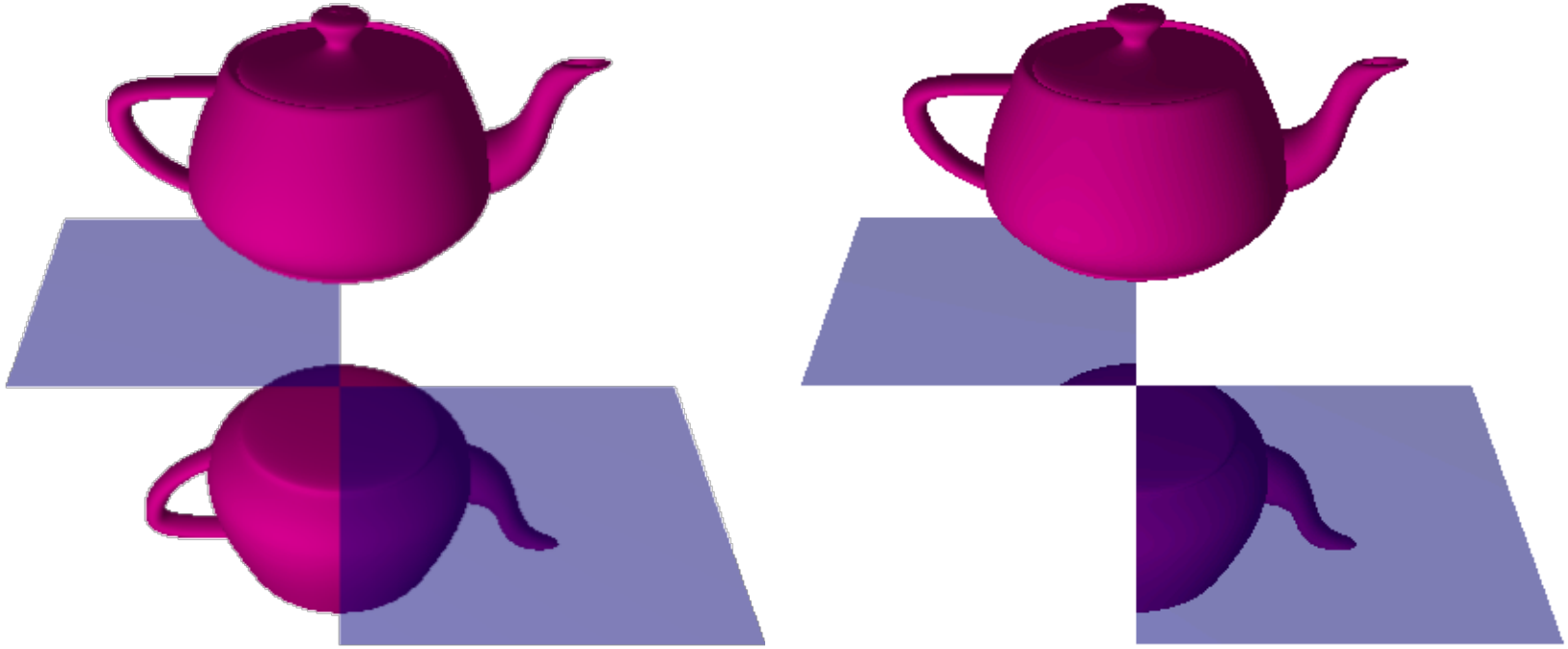
Tracer la scène une deuxième fois,
réfléchie par rapport au plan du miroir

- Calculer la **matrice de réflexion** $\mathbf{M} = \mathbf{F}^{-1}\,\mathbf{S}(1,-1,1)\,\mathbf{F}$
  avec $\mathbf{F} = \mathbf{R}(n,(0,1,0))\mathbf{T}(-\mathbf{p})$
- Concaténer $\mathbf{M}$ avec la matrice de vue

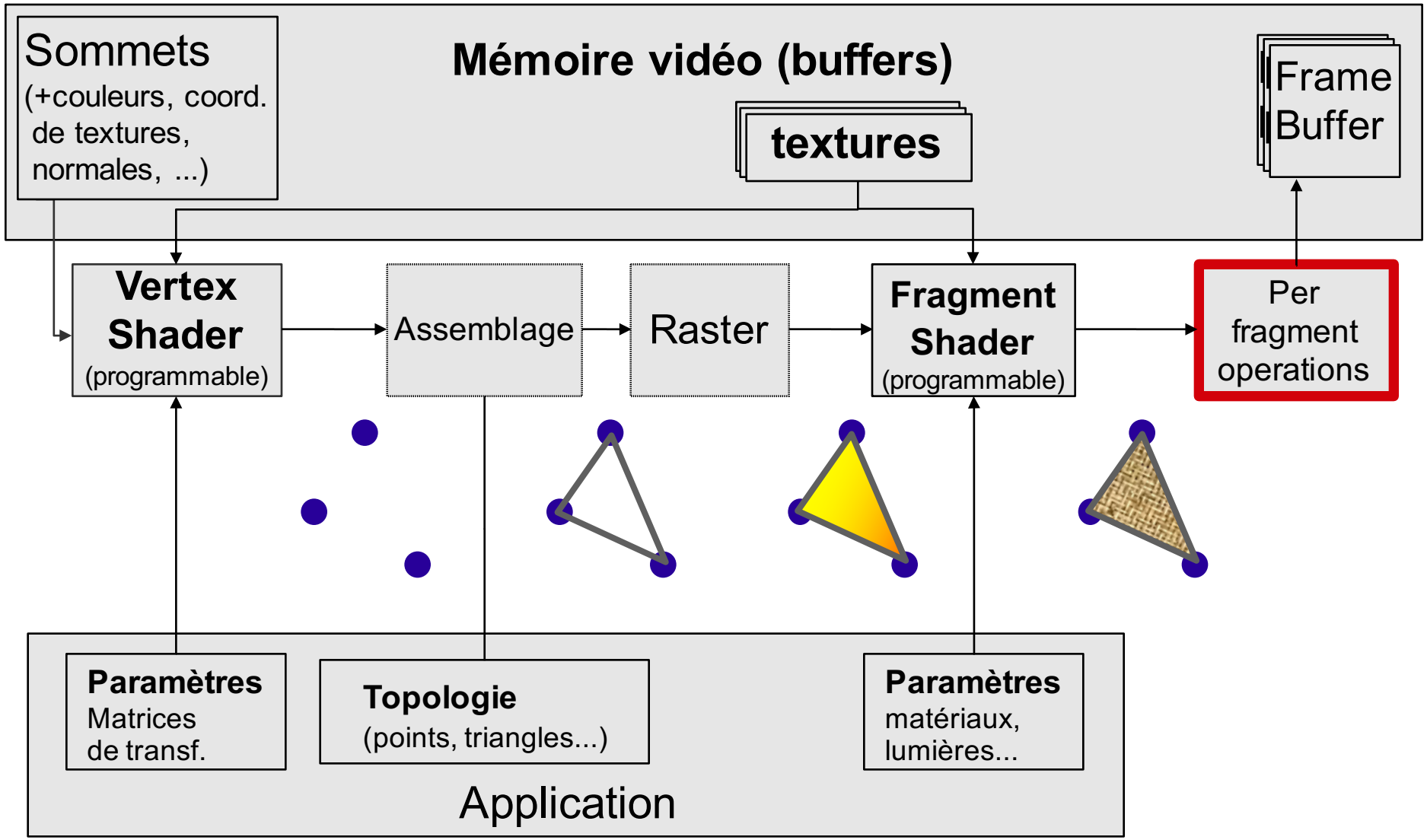


Réflexions planaires – miroir

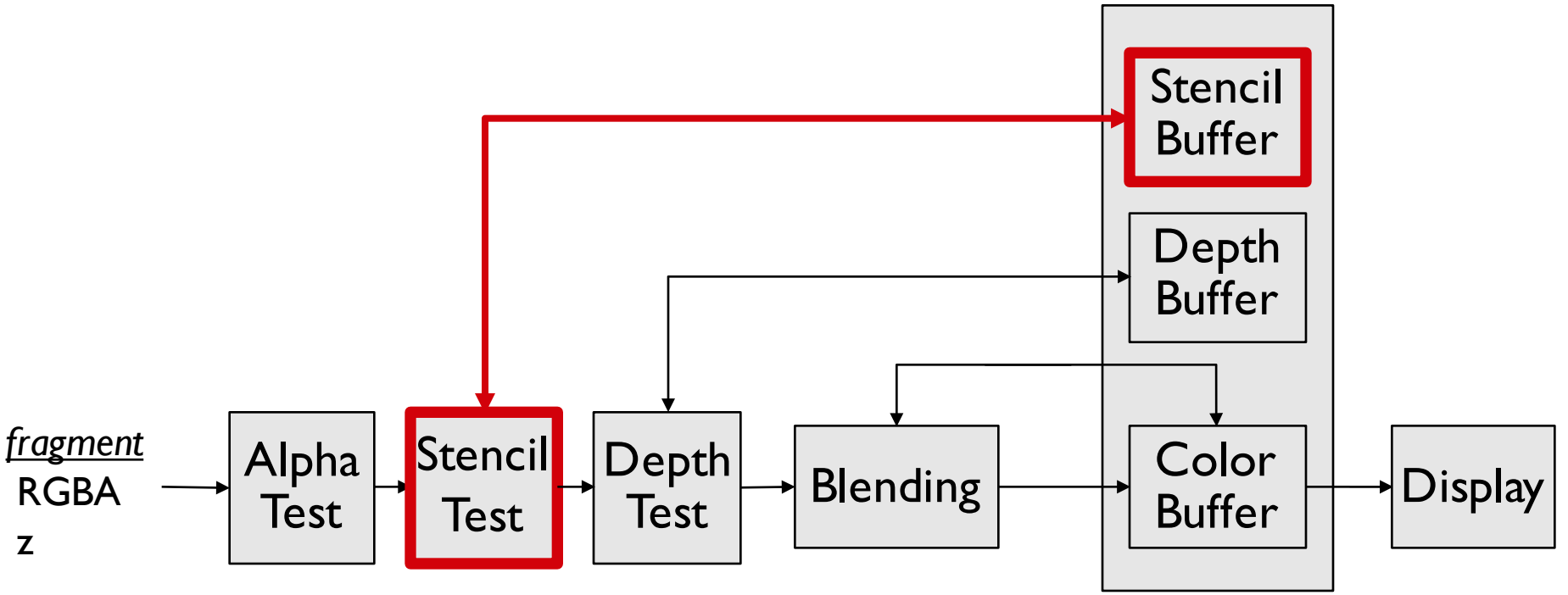
Utilisation du **stencil buffer** pour limiter la réflexion au polygone du miroir



OpenGL


Sommets
(+couleurs, coord. de textures, normales, ...)
Mémoire vidéo (buffers)
textures
Frame Buffer
Vertex Shader
(programmable)
Assemblage
Raster
Fragment Shader
(programmable)
Per fragment operations
Paramètres
Matrices de transf.
Topologie
(points, triangles...)
Paramètres
matériaux, lumières...
Application

Stencil Buffer



Utilisation du Stencil Buffer

Test par pixel sur la valeur du stencil buffer

- Fragment rejeté si test raté
- Test effectué avant le test de profondeur

Test = func (ref & mask , svalue & mask)

- **&** = ET logique bits à bits
- mask = entier 32 bits
- ref = valeur de référence (ce n'est pas un attribut du fragment)
- func = `GL_{ALWAYS, NEVER, LESS, EQUAL, GREATER, NOTEQUAL, GEQUAL, LEQUAL}`
- svalue = valeur dans le stencil buffer
- Cf. : `glStencilFunc(func, ref, mask)`
  `glEnable/Disable(GL_STENCIL_TEST)`

Utilisation du Stencil Buffer

Différentes opérations sur la valeur dans le stencil buffer selon :

- test de stencil raté (*sfail*)
- test de profondeur raté (*dpfail*)
- test de profondeur réussi (*dppass*)

  ⟹ `glStencilOp(fail, zfail, zpass)`

Opérations possibles

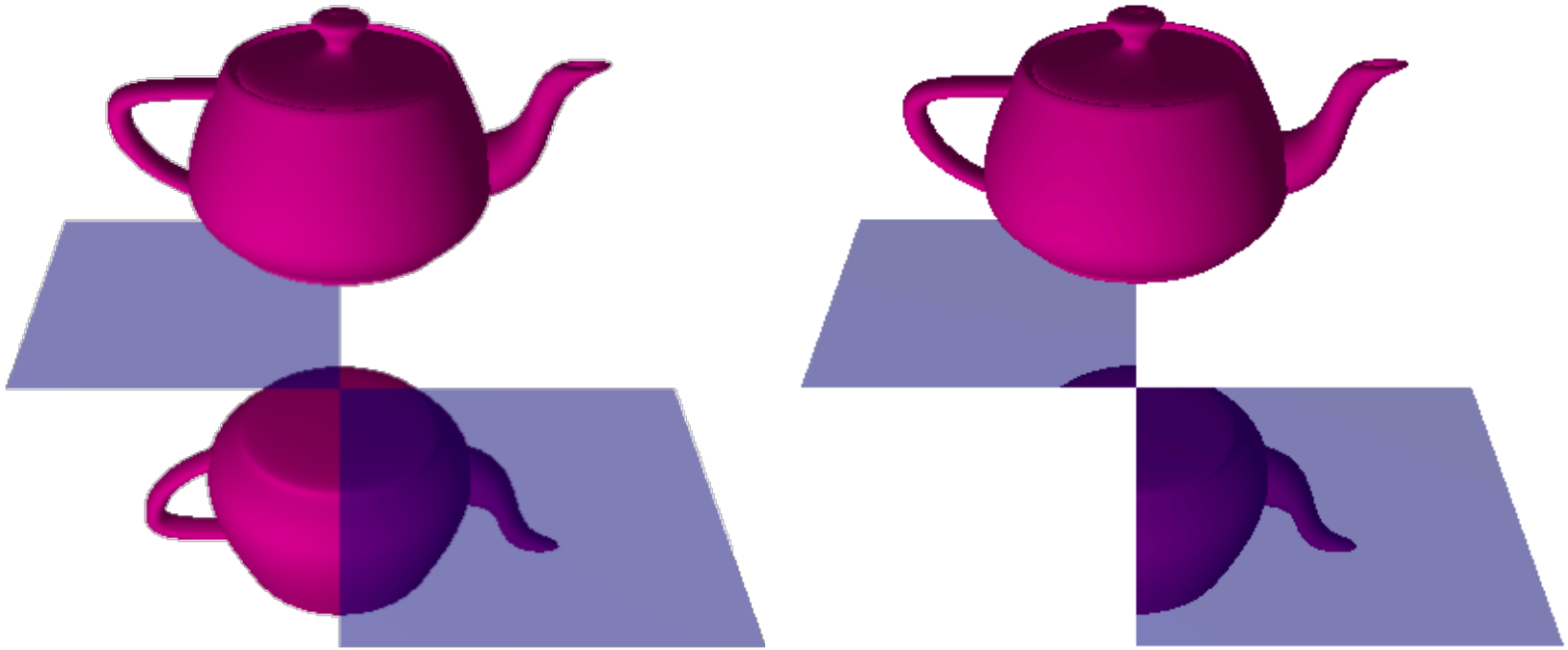
- `GL_KEEP` : garder la valeur
- `GL_REPLACE` : remplacer par la valeur référence
- `GL_ZERO` : remplacer par zéro
- `GL_INCR`, `GL_DECR` : incrémenter, décrémenter
  (avec saturation, c.-à-d. : 255++ ⟹ 255 et 0-- ⟹ 0)
- `GL_INVERT` : inverser bit à bit
- `GL_INCR_WRAP`, `GL_DECR_WRAP` : incrémenter, décrémenter
  (avec bouclage sur l'octet, c.-à-d. : 255++ ⟹ 0 et 0-- ⟹ 255)

Exemple du miroir

```
glStencilFunc(GL_ALWAYS, 1, 255);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
glEnable(GL_STENCIL_TEST);
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glDepthMask(GL_FALSE);
// tracer le plan, dans le stencil uniquement
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glDepthMask(GL_TRUE);
// activer un vertex shader avec calcul de la
symétrie
// et de la distance de clipping
glStencilFunc(GL_EQUAL, 1, 255);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
// tracer la scène
glDisable(GL_STENCIL_TEST);
// tracer le plan avec blending
// ou seulement dans le depth buffer
```

Réflexions planaires – miroir

Utilisation du **stencil buffer** pour limiter la réflexion au polygone du miroir



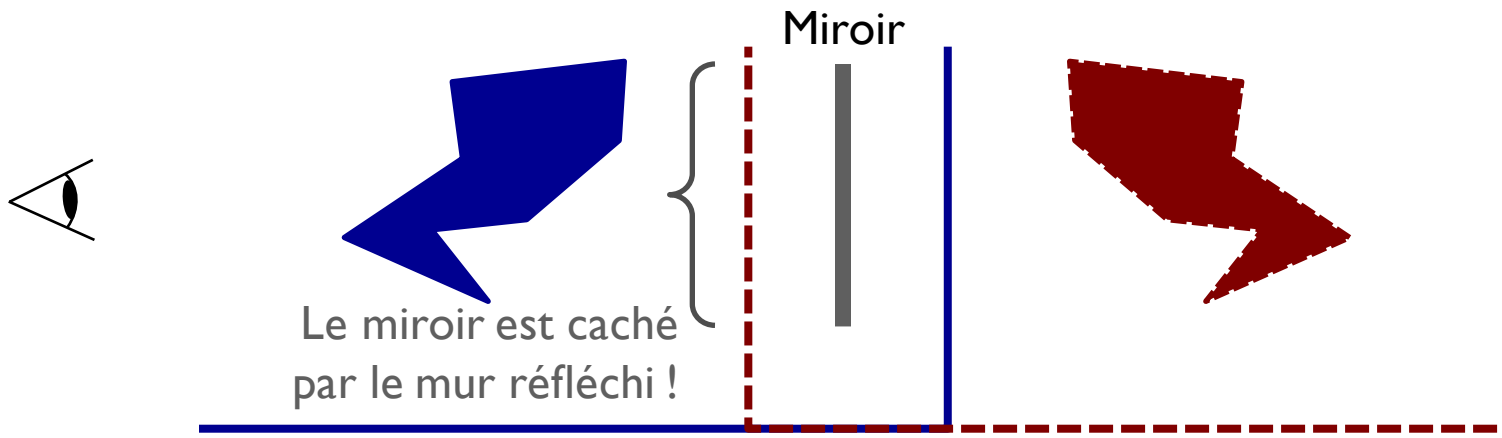
Mettre le stencil buffer à 0

Dessiner le polygone du miroir en mettant le stencil à 1

Dessiner la scène réfléchie seulement là où le stencil est à 1

Miroir et plan de clipping

Attention, si le miroir est **devant** un mur…



…il ne faut pas redessiner le mur

⟹ Utilisation de **plans de *clipping* supplémentaires**

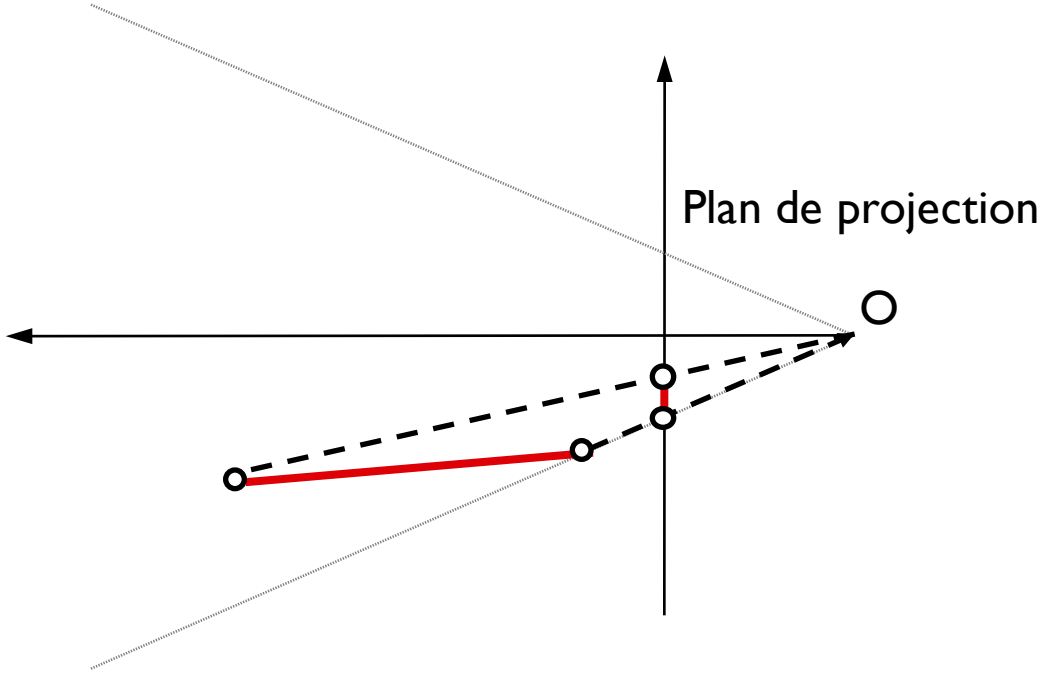
Clipping

Projection perspective = division par z

Que se passe-t-il quand un point est derrière la caméra ?

Sur le plan de projection ?

$\Longrightarrow$ **Solution : *clipping***



Plans de clipping

CPU :

```
glEnable/Disable(GL_CLIP_DISTANCEi)
```

Vertex shader output :

```
float gl_ClipDistance[i]
```

- Interpolée linéairement dans la primitive
- Partie de la primitive dont gl_ClipDistance < 0.0 « clippée »

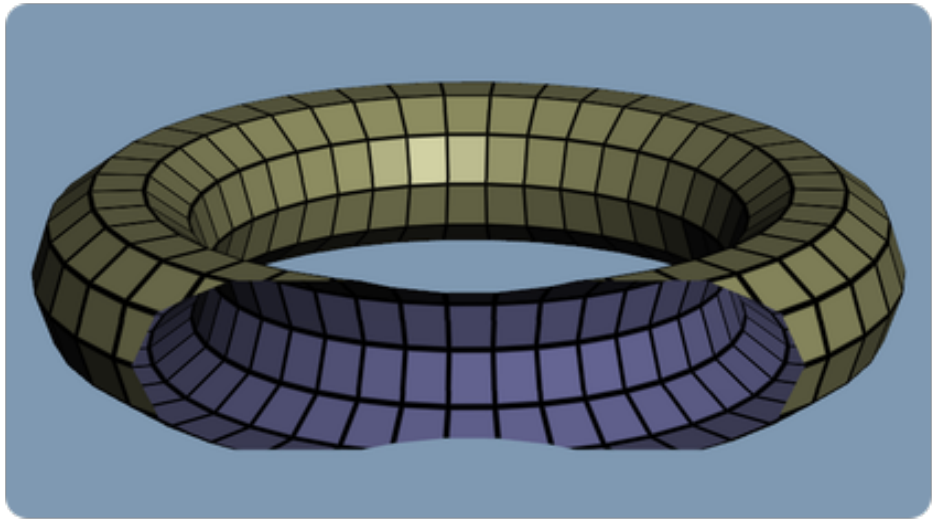


Image : http://github.prideout.net/clip-planes/

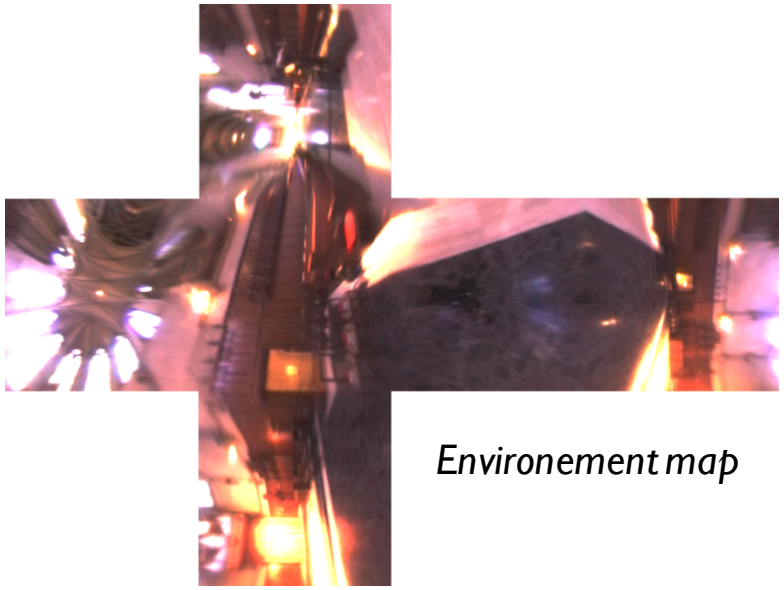
Environment Mapping

Hypothèse

réflexion dépend uniquement de la **direction**

$\Longrightarrow$ objets réfléchis et source lumineuses **lointaines**

$\Longrightarrow$ surface réfléchissante ne se réfléchit **pas elle-même**



Environement map

Image d'Henrik Wann Jensen - Environment map de Paul Debevec

Spherical mapping

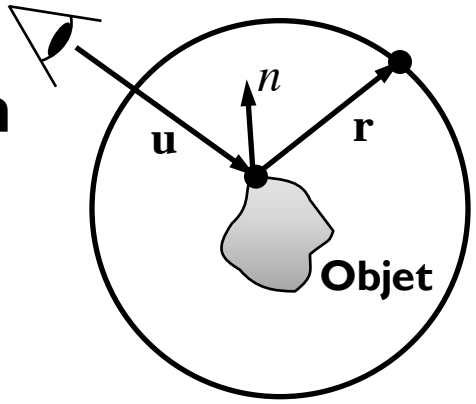
[Williams 83, Miller & Hoffman 84]

Utilisation

- Texture d'environnement mappée sur une (hémi)sphère
- Calcul des coordonnées de textures à la volée
  - calculer le vecteur réfléchi : $\mathbf{r} = \mathbf{u} - 2n(n \bullet \mathbf{u})$ = `reflect(u,n)`
  - puis mapping 3D unitaire ⟶ 2D
    …différentes possibilités (1 hémisphère, 2 hémisphères, etc.)
- Par sommet (vertex shader) ou par fragment (fragment shader)

Qualité dépend de la paramétrisation

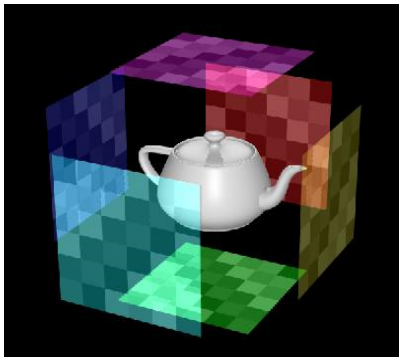
⟹ fortes déformations

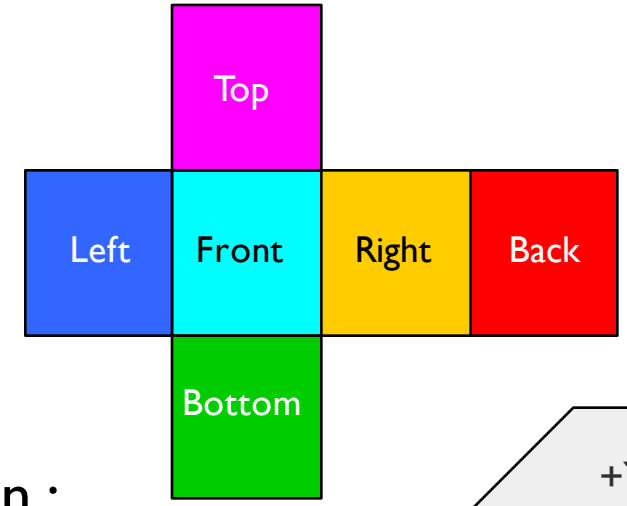


Cube mapping [Green 1986]

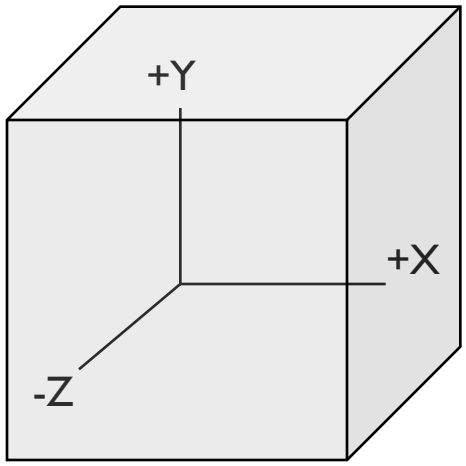
Paramétrisation de la sphère par un cube

- Une « cube map » est composée de six textures carrées de même taille, représentant un cube centré à l'origine :





- Chaque texture du cube correspond à une direction :
  +X, -X, +Y, -Y, +Z, -Z
- **Équivalent à 6 textures 2D** avec un accès particulier…



Cube mapping

La cube map est accédée par des **coordonnées de textures 3D** (r_x, r_y, r_z)

Une **transformation 3D → 2D** est réalisée (automatiquement en GLSL) pour accéder à l'une des 6 textures 2D :

- sélectionner la texture par la composante la plus grande : soit T la plus grande et (sc, tc) les deux autres
- alors, coordonnées 2D (s,t) :

$$s = (sc / abs(T) + 1) / 2$$

$$t = (tc / abs(T) + 1) / 2$$

- utiliser (s,t) pour accéder au texel de la texture sélectionnée

Cube mapping en pratique

OpenGL

- Nouveaux paramètres de destination : `GL_TEXTURE_CUBE_MAP_{POSITIVE,NEGATIVE}_{X,Y,Z}`
- Exemple :

```
glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB,
  w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, face_px);
glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0,
  GL_RGB, w, h,0, GL_RGB, GL_UNSIGNED_BYTE, face_nx);
// idem pour les 2 autres directions
```

Shader

- Déclaration du sampler : `uniform samplerCube cubemap;`
- Utilisation :

```
vec3 r = reflect(u,n);
vec4 color = texture(cubemap, r);
```

Réflexions dynamiques

⟹ **Rendu hors-écran :**
FrameBuffer Object

FrameBuffer Objects (FBO)

Collection de buffers : lorsqu'un FBO est lié, tous les rendus sont écrits dans les buffers associés

Un FBO dispose de plusieurs **attachements** :

- un ou plusieurs ***color buffers*** (max. spécifique au GPU)
  $\Longrightarrow$ ***render buffer*** : optimisé, mais pas réutilisable dans un 2nd shader
  $\Longrightarrow$ **texture**
- au maximum **un** ***depth buffer***
- au maximum **un** ***stencil buffer***

FBO – création

```
glGenFramebuffer(1,&id) // on crée un FBO
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, id) // on l'active

glGenTexture(…) // on crée une texture pour stocker la couleur
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,
      GL_COLOR_ATTACHMENT0 …) // on l'attache au FBO

glGenRenderbuffer(…) // on crée un render buffer
                     // pour stocker la depth map
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER,
            GL_DEPTH_ATTACHMENT, …)  // on l'attache au FBO
```

FBO : http://www.opengl.org/wiki/Framebuffer_Object

***Render buffers* :**

http://www.opengl.org/wiki/Renderbuffer_Object

FBO – utilisation

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER,id) // On l'active
// On associe la variable out du fragment shader out_color
// au color buffer 0
glBindFragDataLocation(prg_id, 0, "out_color");
// Important : on redéfini le viewport à la taille du FBO
glViewport(0, 0, fbo_width, fbo_height);
// On réinitialise les couleurs et le depth buffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// on dessine dans le FBO

glBindFramebuffer(GL_DRAW_FRAMEBUFFER,0) //on le désactive

// on peut utiliser la texture et/ou le depth buffer
// pour une seconde passe de rendu
```

Réflexions dynamiques

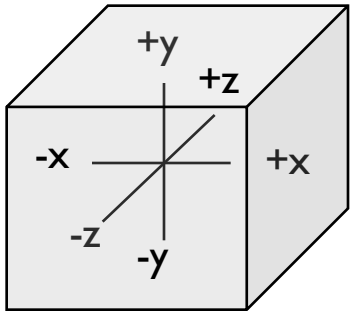
À chaque image

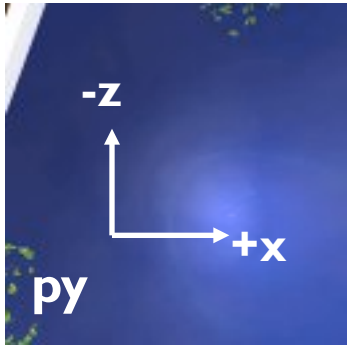
1. Générer les six vues du cube, par ex. pour la 1ère vue :
   - positionner la caméra au centre de l'objet/scène
     - angle d'ouverture de 90°
     - direction : +X
   - activer le rendu dans la 1ère image du cube map (FBO)

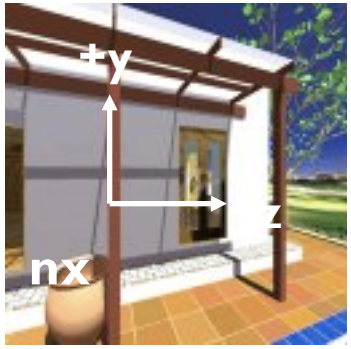
```
glFramebufferTexture2D(fbo_id,GL_COLOR_ATTACHMENT0,
    GL_TEXTURE_CUBE_MAP_POSITIVE_X,cubemap_id,0,0);
```

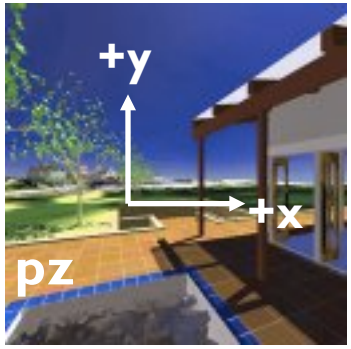
   - tracer le reste de la scène
2. Activer la caméra principale
3. Tracer le reste de la scène
4. Activer la cube map et le shader correspondant
   (avec calcul du vecteur réfléchi)
5. Tracer l'objet réfléchissant

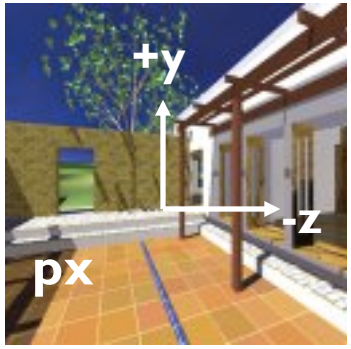
Réflexions dynamiques

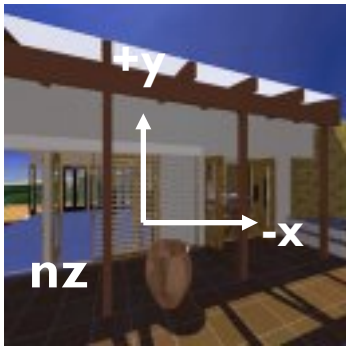


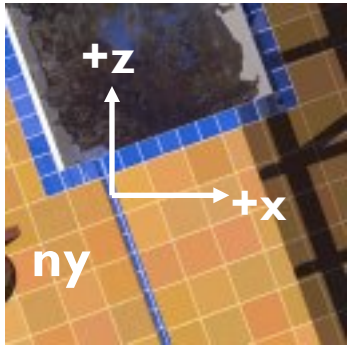




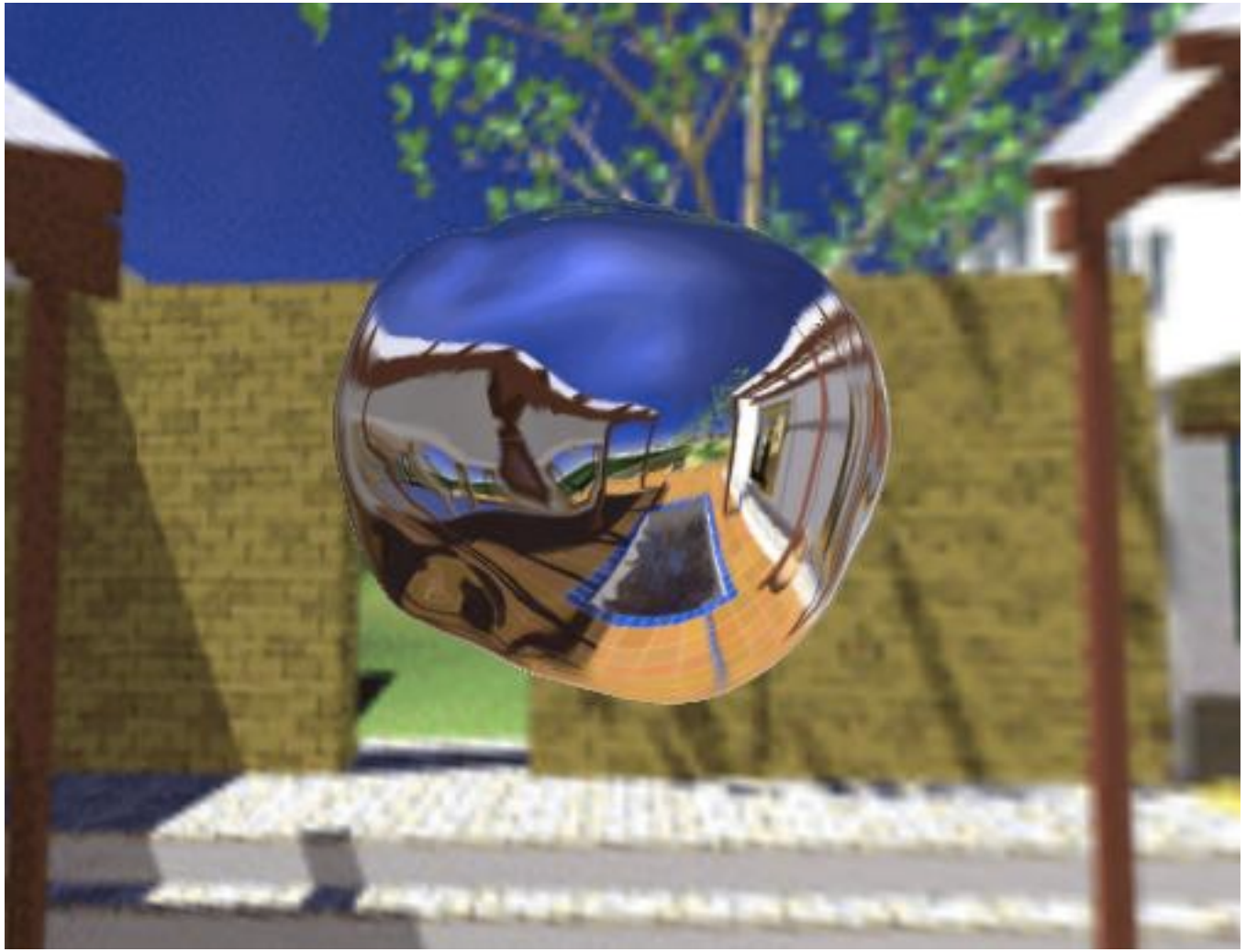








Réflexion dynamique



Réfraction

Snell-Descartes : $\frac{sin\theta_1}{sin\theta_2} = \frac{i_1}{i_2} = i$

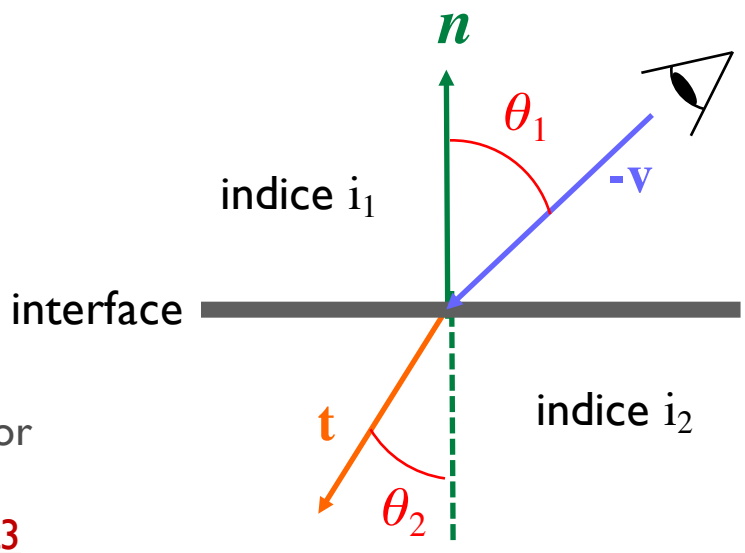
indice de réfraction relatif

Vecteur réfracté : $\mathbf{t} = (w - k)\boldsymbol{n} - i\mathbf{v}$

$w = i\,(\mathbf{v} \bullet \boldsymbol{n})$

$k = \sqrt{1 + (w - i)(w + i)}$

GLSL : `refract(v,n,i)`

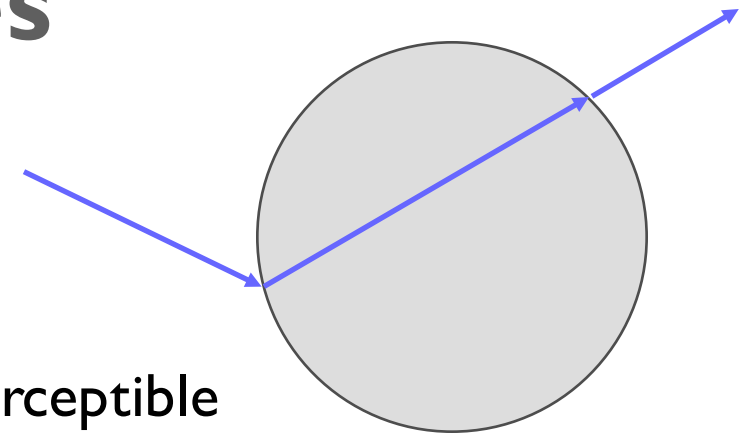


Bec, Xavier, "Faster Refraction Formula, and Transmission Color Filtering," in Ray Tracing News, vol. 10, no. 1, January 1997
http://tog.acm.org/resources/RTNews/html/rtnv10n1.html#art3

Réfractions multiples

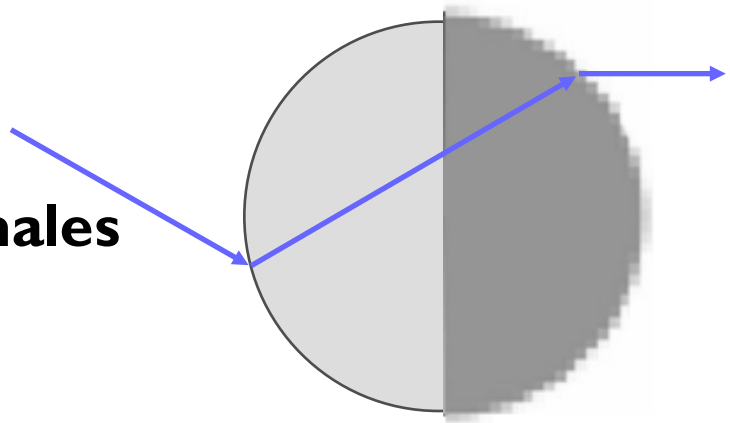
Dans la majorité des cas,
pas prises en compte

- grosse approximation
  …mais parfois différence quasi imperceptible



2 interfaces [Oliveira & Brauwers 2007]

- Rendu des faces arrières :
  stockage des **profondeurs** et **normales**
- Rendu des faces avants :
  - calcul de la direction réfracté,
  - *ray marching* dans le buffer de profondeur (champ de hauteur)
  - calcul de la second direction de réfraction



Réfractions multiples [Oliveira & Brauwers 2007]

1 réfraction

2 réfractions

Réflexions / réfractions *glossy*

Pas une seule direction de réflexion / réfraction

Solutions :

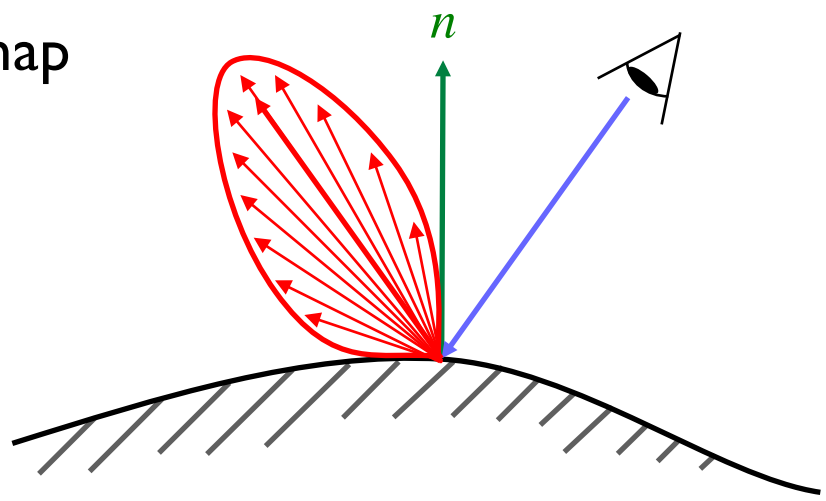
- **Échantillonnage**

  $\Longrightarrow$ couteux, bruité

- **Pré-filtrage de la carte d'environnement**

  hypothèse : même forme du lobe quelque soit le point de vue

  $\Longrightarrow$ stockage possible de différentes rugosités dans une MIP-map



Réflexions *glossy*

[Ashikhmin & Ghosh 2002]

Réfractions *glossy*

[de Rousiers et al. 2012]