MV52
Synthèse d'images

CM #10

Rendu d'images
Techniques classiques de
Fragment Shading

Fabrice LAURI
fabrice.lauri@utbm.fr

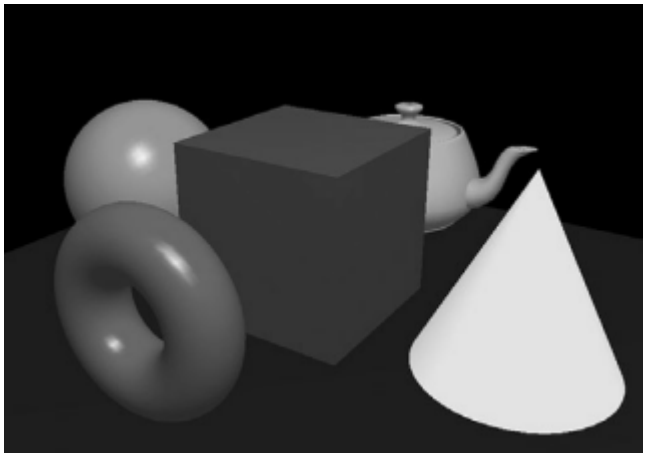
Plan du cours

- *Color Conversion*
- *Traitement d'images*
- *Cel-shading*
- **Textures spécifiques**
- **Textures procédurales**

Plan du cours

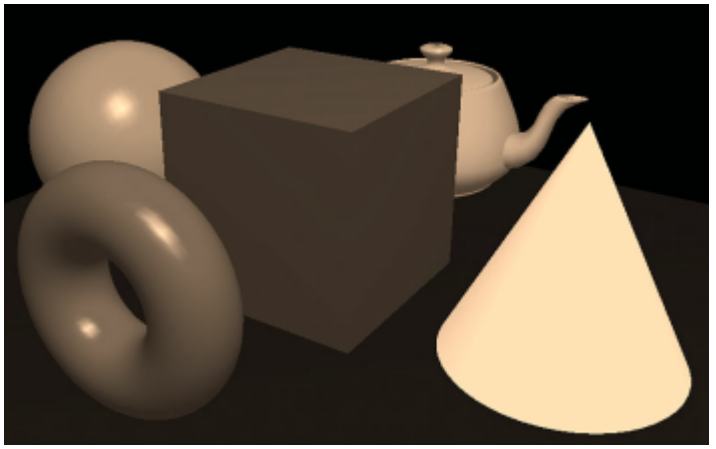
- *Color Conversion*
- *Traitement d'images*
- *Cel-shading*
- **Textures spécifiques**
- **Textures procédurales**

Techniques de conversion de couleurs



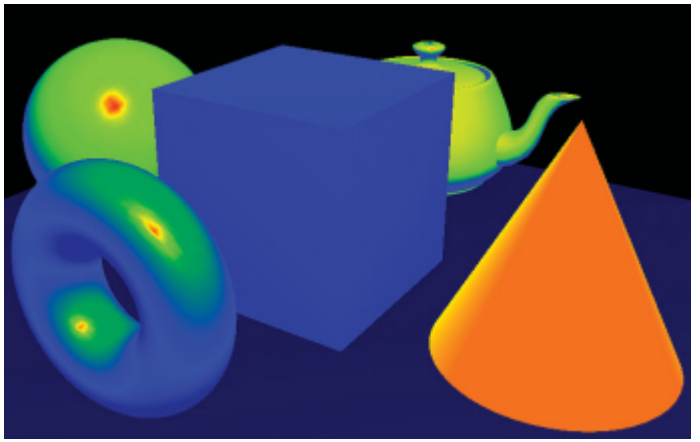
Gray conversion

Facteurs NTSC :
R : 0.299
V : 0.587
B : 0.114



Sepia Conversion

Facteurs NTSC + modulation du niveau de gris par composante RVB (1.2,1.0,0.8)



Heat signature Conversion

Plan du cours

- *Color Conversion*
- *Traitement d'images*
- *Cel-shading*
- **Textures spécifiques**
- **Textures procédurales**

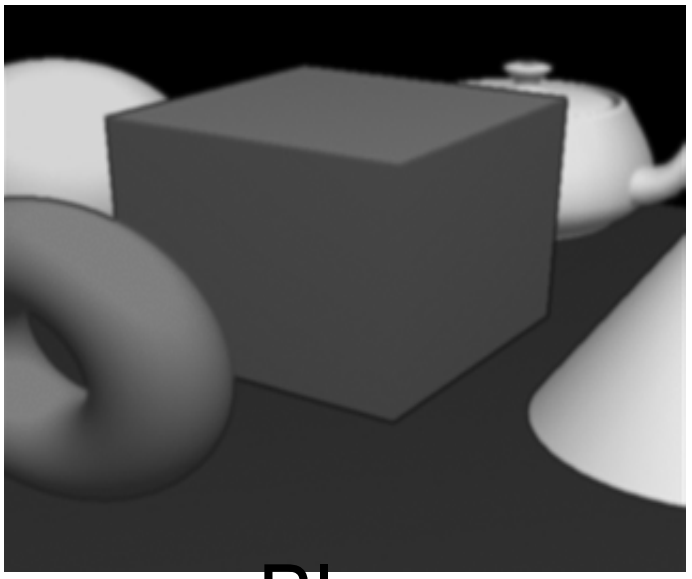
Traitement d'images

Le traitement d'images par *Fragment Shaders* consiste à appliquer un filtre sur l'image provenant de la scène.

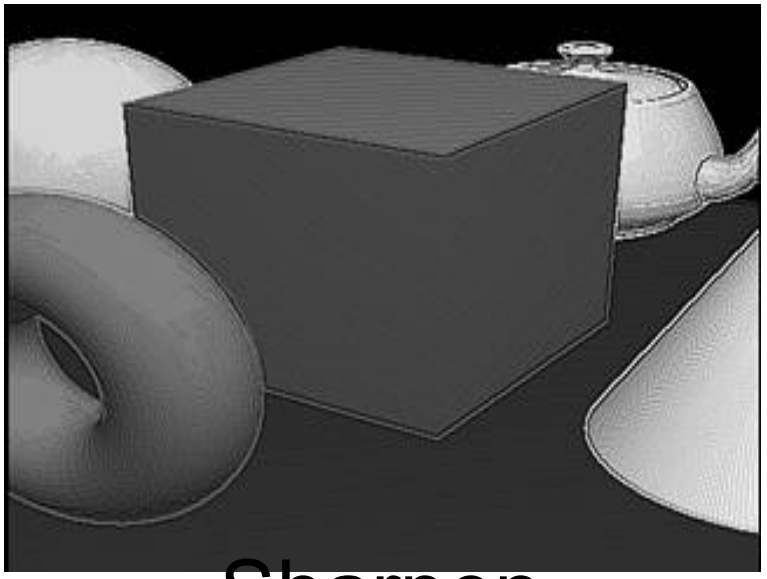
Within the sample application, glCopyTexImage2D is called to copy the contents of the framebuffer into a texture. The texture size is chosen to be the largest power-of-2 size smaller than the window. A fragment-shaded quad is then drawn centered within the window with the same dimensions as the texture, with a base texture coordinate ranging from (0,0) in the lower left to (1,1) in the upper right.

The fragment shader takes its base texture coordinate and performs a texture lookup to obtain the center sample of the 3x3 kernel neighborhood. It then proceeds to apply eight different offsets to lookup samples for the other eight spots in the neighborhood. Finally, the shader applies some filter to the neighborhood to yield a new color for the center of the neighborhood.

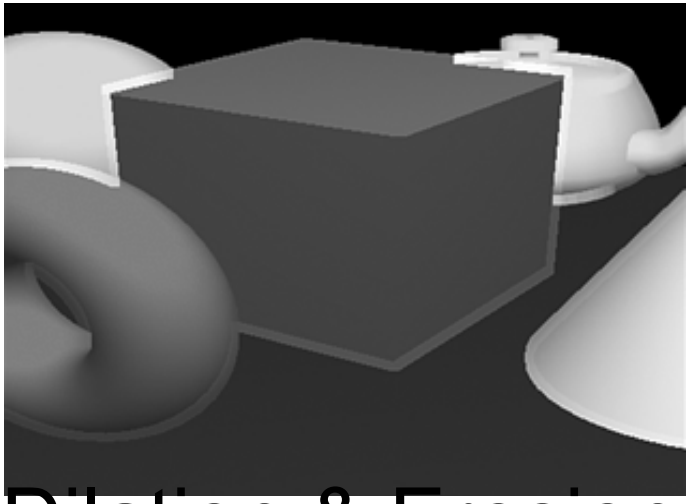
Techniques de traitement d'images



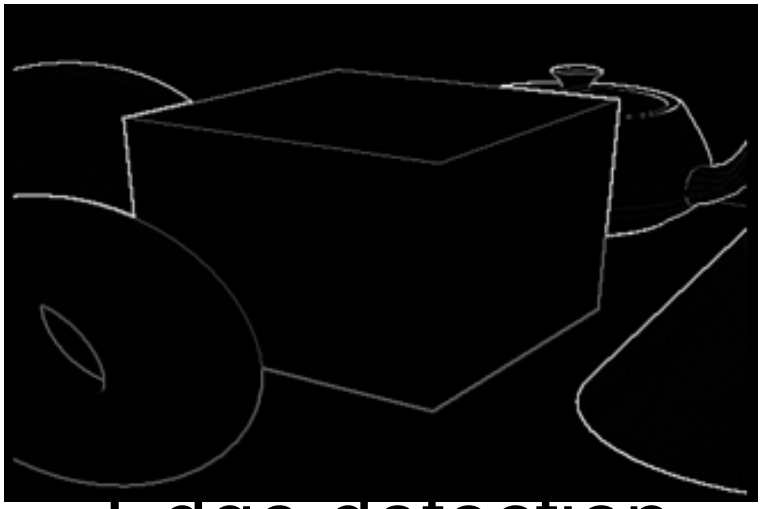
Blur



Sharpen



Dilation & Erosion



Edge detection

Technique *Blur*

```
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
out vec4 fColor ;
void main(void) {
    vec4 sample[9];
    for (int i = 0; i < 9; i++) {
        sample[i] = texture2D(sampler0, gl_TexCoord[0].st +
        tc_offset[i]);
    }
    // 1 2 1
    // 2 1 2 / 13
    // 1 2 1
fColor = (sample[0] + (2.0*sample[1]) + sample[2] +
(2.0*sample[3]) + sample[4] + (2.0*sample[5]) + sample[6] +
(2.0*sample[7]) + sample[8]) / 13.0;
}
```

Technique *Sharpen*

```
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
out vec4 fColor ;
void main(void) {
    vec4 sample[9];
    for (int i = 0; i < 9; i++) {
        sample[i] = texture2D(sampler0, gl_TexCoord[0].st +
         tc_offset[i]);
    }
    // -1 -1 -1
    // -1 9 -1
    // -1 -1 -1
    fColor = (sample[4] * 9.0) - (sample[0] + sample[1] +
    sample[2] + sample[3] + sample[5] + sample[6] + sample[7] +
    sample[8]);
}
```

Technique *Dilation*

```
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
out vec4 fColor ;

void main(void) {
    vec4 sample[9];
    vec4 maxValue = vec4(0.0);
    for (int i = 0; i < 9; i++) {
        sample[i] = texture2D(sampler0, gl_TexCoord[0].st +
        tc_offset[i]);
        maxValue = max(sample[i], maxValue);
    }
    fColor = maxValue;
}
```

Technique *Erosion*

```
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
out vec4 fColor ;

void main(void) {
    vec4 sample[9];
    vec4 minValue = vec4(1.0);
    for (int i = 0; i < 9; i++) {
        sample[i] = texture2D(sampler0, gl_TexCoord[0].st +
        tc_offset[i]);
        minValue = min(sample[i], minValue);
    }
    fColor = minValue;
}
```

Technique *Laplacian Edge Detection*

```
uniform sampler2D sampler0;
uniform vec2 tc_offset[9];
out vec4 fColor ;
void main(void) {
    vec4 sample[9];
    for (int i = 0; i < 9; i++) {
        sample[i] = texture2D(sampler0, gl_TexCoord[0].st +
         tc_offset[i]);
    }
    // -1 -1 -1
    // -1 8 -1
    // -1 -1 -1
    fColor = (sample[4] * 8.0) - (sample[0] + sample[1] +
    sample[2] + sample[3] + sample[5] + sample[6] + sample[7] +
    sample[8]);
}
```

Plan du cours

- *Color Conversion*
- *Traitement d'images*
- *Cel-shading*
- **Textures spécifiques**
- **Textures procédurales**

Cel-shading

Type de rendu non photo-réaliste.
Rendu proche d'un dessin de bande dessinée.

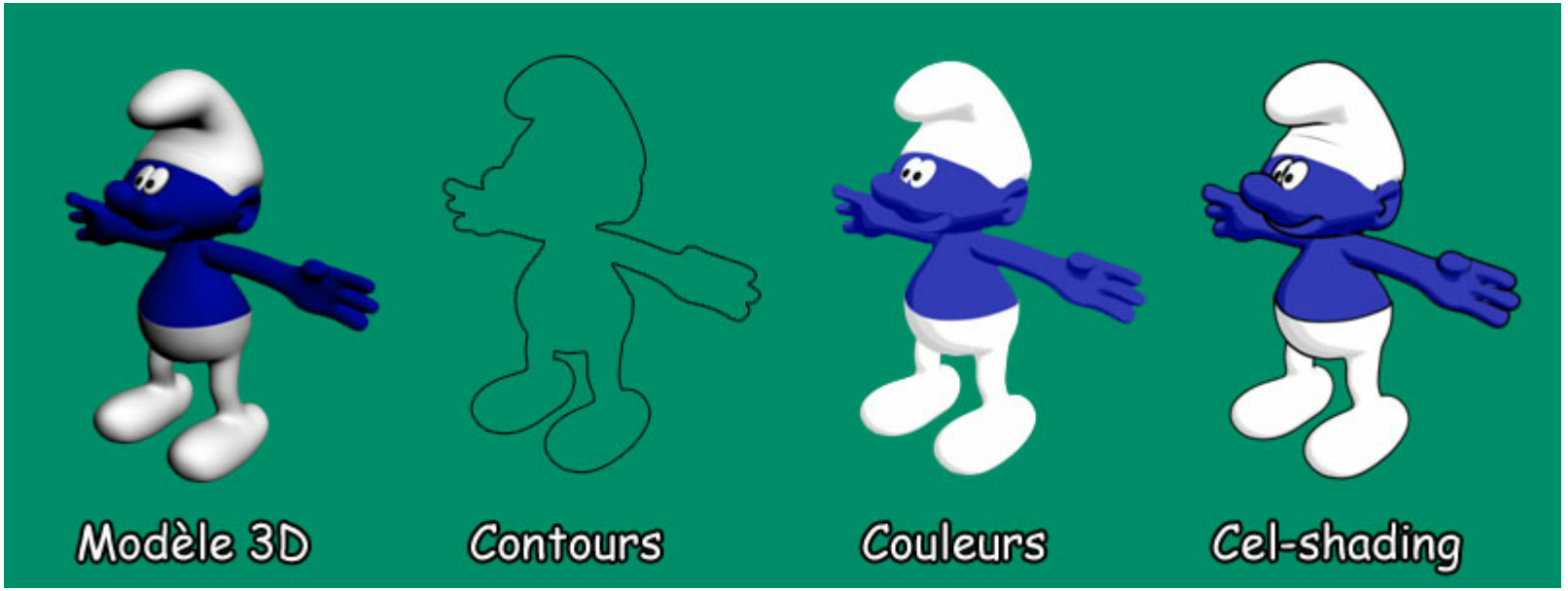
Jet set radio (2000)

XIII (2003)

Cel-shading

Caractéristiques d'un dessin de bande dessinée

- contour noir (correspondant au trait du dessinateur)
- palette de couleurs et dégradés plutôt que le respect de la qualité des jeux de lumières.



Cel-shading

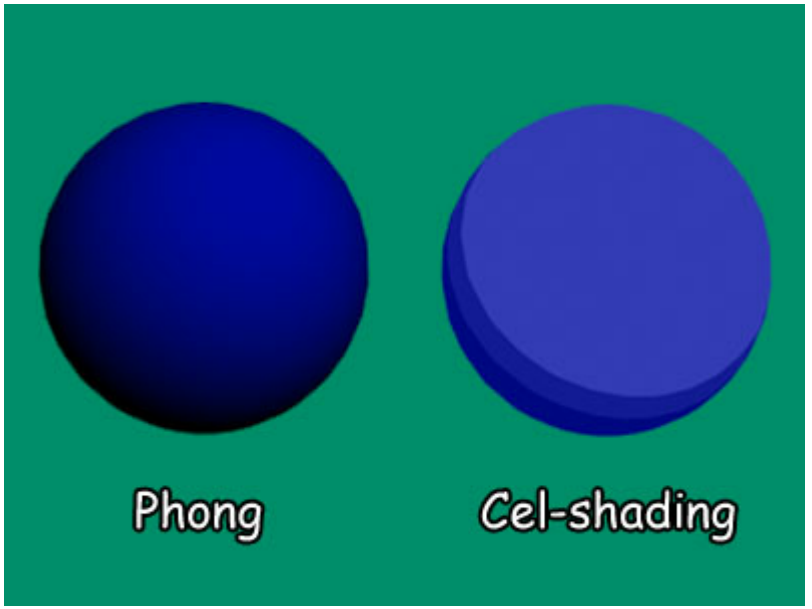
Rappel : modèle d'ombrage de Gouraud ou de Phong

Ils se composent de trois composantes de lumières : ambiant, diffuse et spéculaire.

- Composante ambiante : affecte la couleur d'un objet lorsque toutes les lumières sont éteintes.
- Composante diffuse : tient compte de l'absorption et de la transmission de la lumière par l'objet.
- Composante spéculaire : tient compte de la reflection de la lumière par l'objet.

Cel-shading

Modèle d'ombrage de Cel-Shading
Pour simplifier, pas de composante spéculaire.
Couleur résultante = ambiant+diffuse



Principe : simplifier le calcul de la lumière diffuse.

Cel-shading

Principe du modèle d'ombrage de Cel-Shading

- Utiliser une texture 1D stockant la contribution RGB de la lumière diffuse
- Accéder au texel de coordonnées s en tenant compte de l'angle a entre la direction normalisée L du vertex vers la lumière et la normale normalisée N au vertex, sachant que :

$$a = \max(0, L.N)$$

Texture de lumière classique

Texture de lumière seuillée

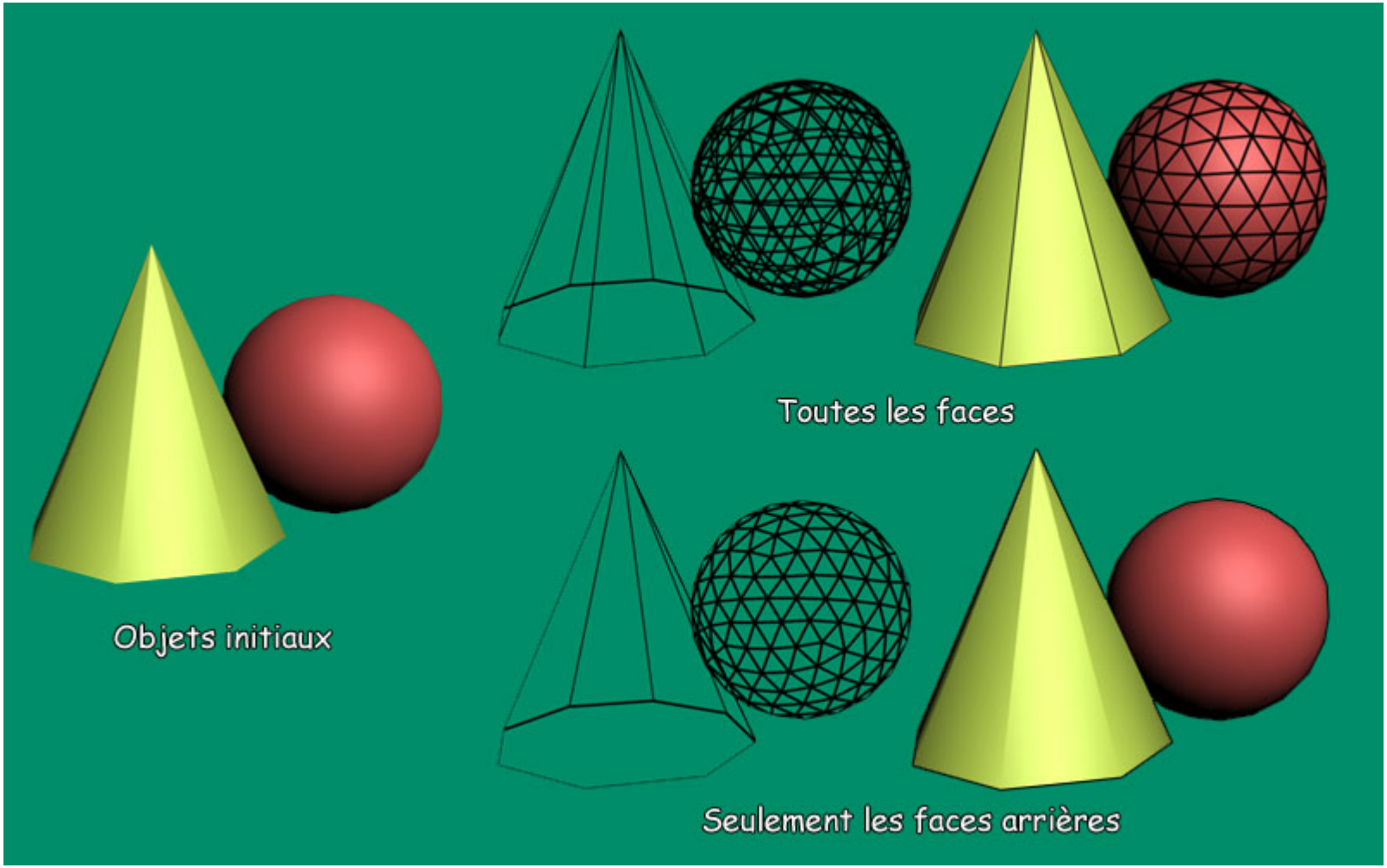
Cel-shading

Pour calculer les contours du modèle 3D, plusieurs techniques :

- *en disposant d'un tableau des arêtes visibles*
- en manipulant le *Backface Culling*
- *en appliquant un filtre de contour*

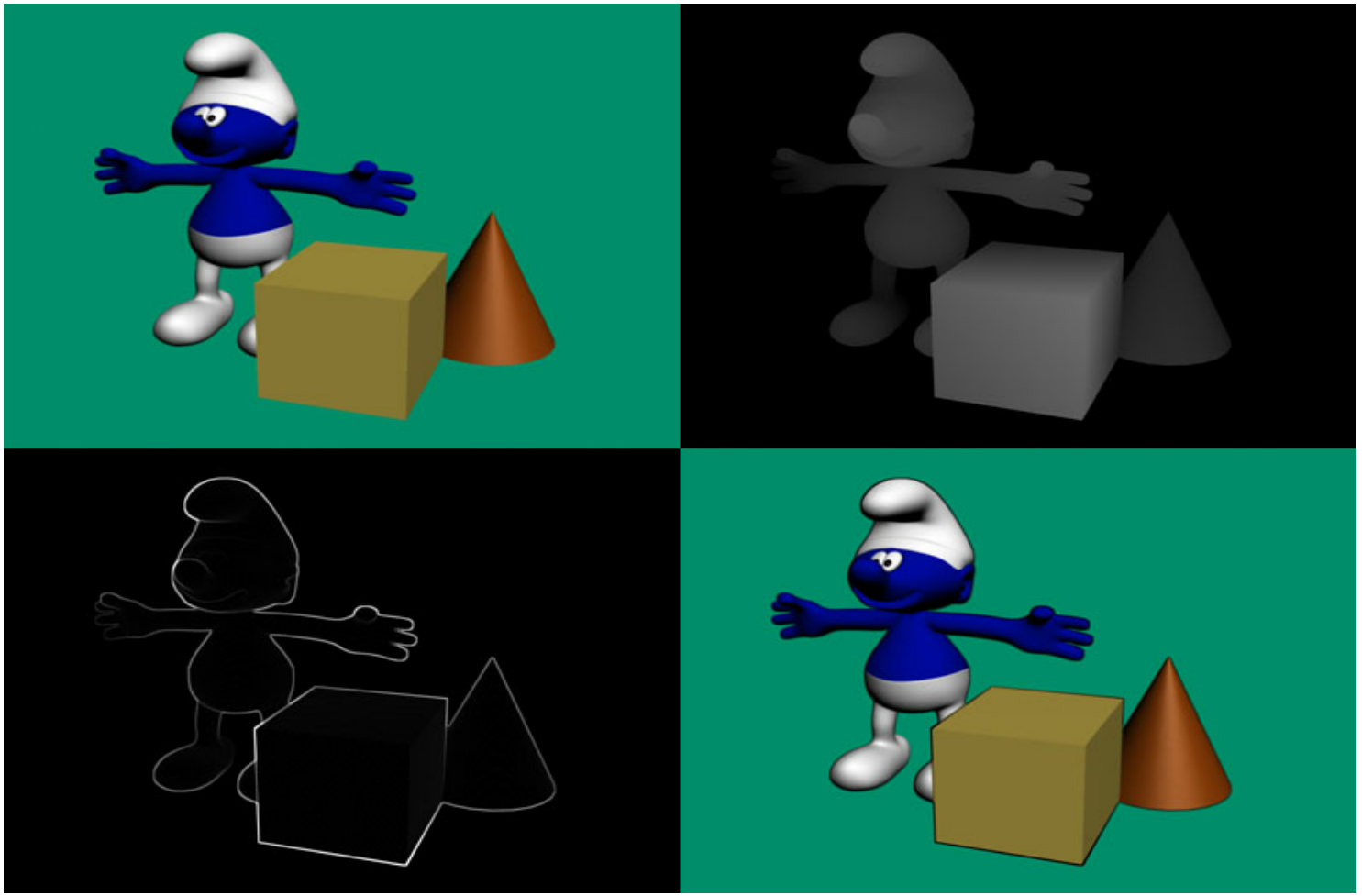
Cel-shading

Calcul des contours en manipulant le *Backface Culling*



Cel-shading

Calcul des contours par *Deferred Shading* et application d'un filtre de contour *(Laplacian, Sobel ou Prewitt)*



Plan du cours

- *Color Conversion*
- *Traitement d'images*
- *Cel-shading*
- **Textures spécifiques**
- **Textures procédurales**

Textures spécifiques

Des textures spécifiques, c'est-à-dire dont les texels ne représentent pas des couleurs RGB, sont utilisées dans les techniques de mapping suivantes :

- *Light Mapping*
- *Gloss Mapping*
- *Alpha Mapping*
- *Emissive Mapping*
- *Displacement Mapping*
- *Bump Mapping*

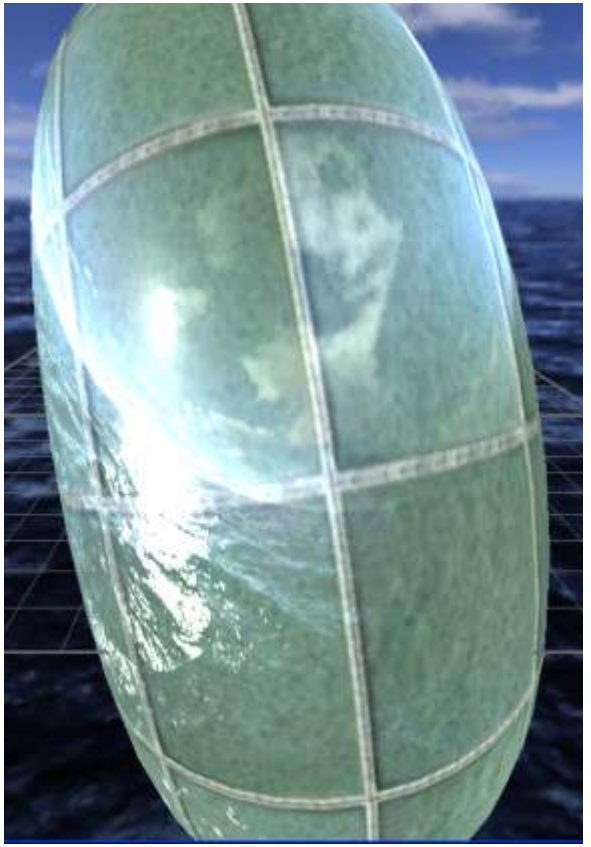
Le *Light Mapping*

Simuler l'illumination reçue par une surface en utilisant une texture.

Applicable lorsque les lumières sont statiques...

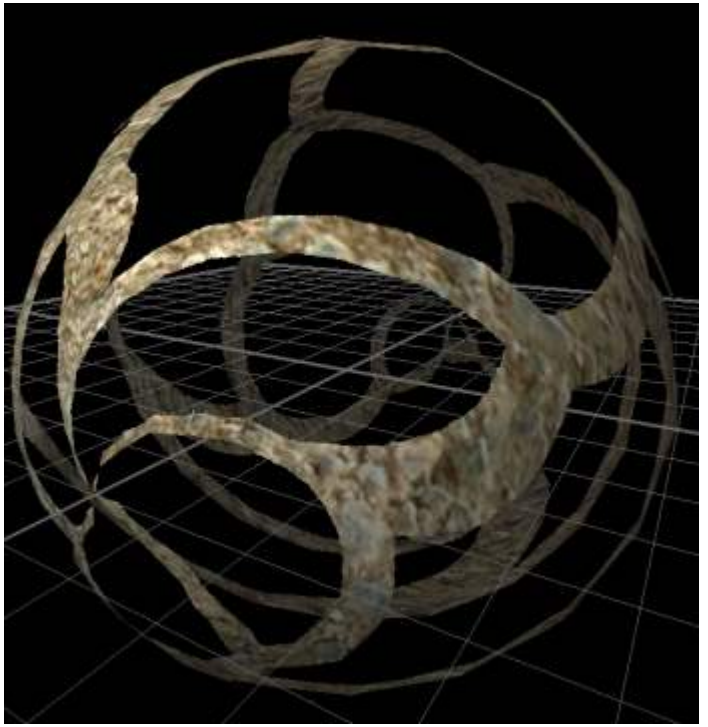
Le *Gloss Mapping*

Moduler la contribution de la lumière spéculaire reçue par une surface à l'aide d'une texture.



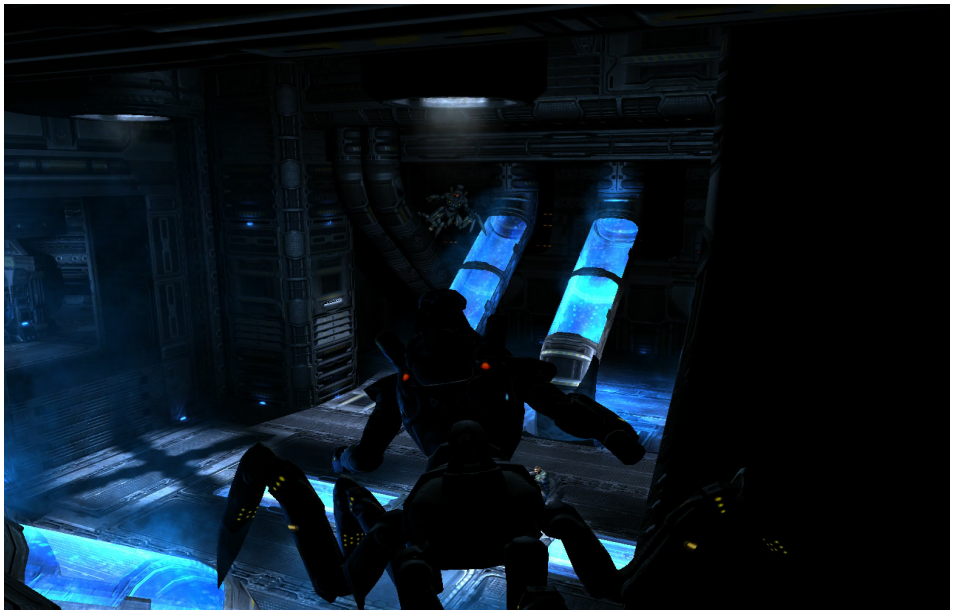
L'*Alpha Mapping*

Déterminer les fragments qui seront rendus à l'aide d'une texture.



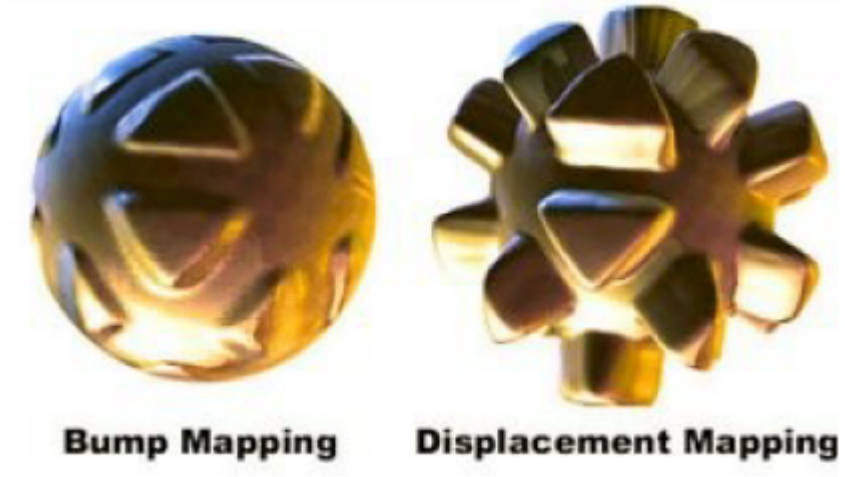
L'*Emissive Mapping*

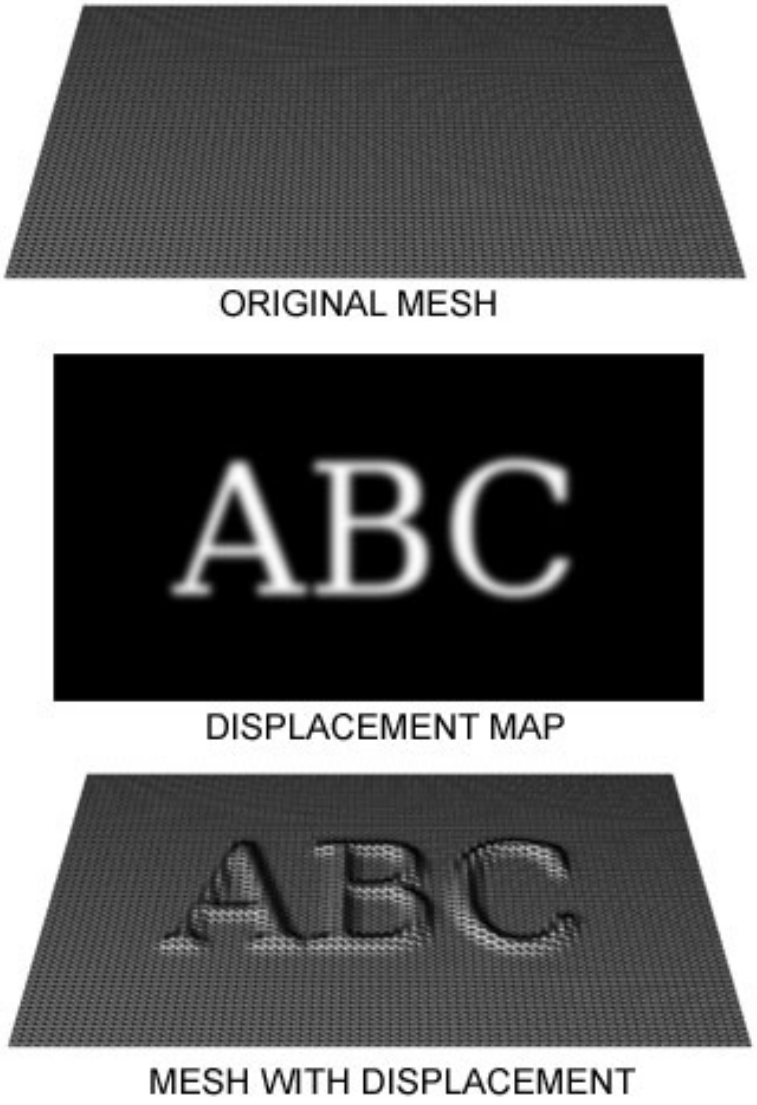
Déterminer la couleur des fragments qui simulent l'émission de lumière.



Le *Displacement Mapping*

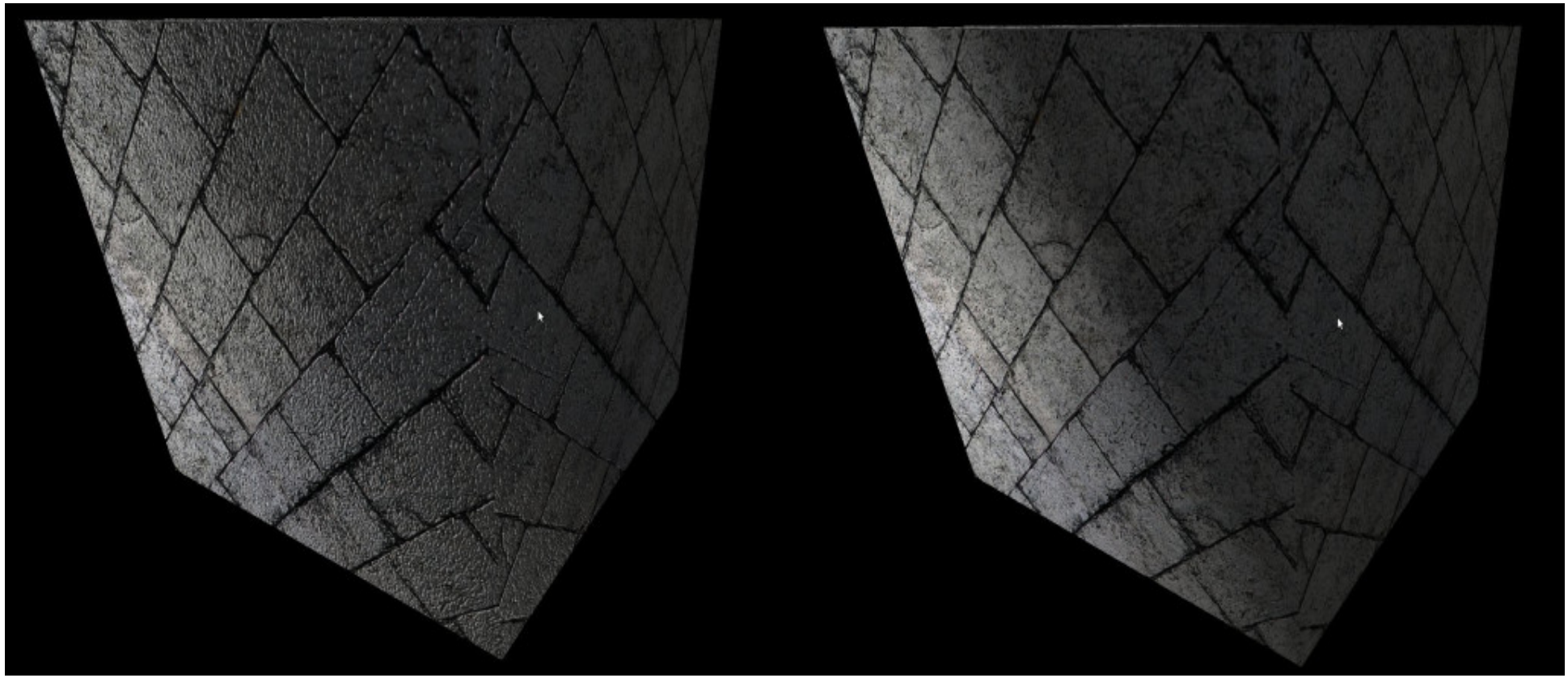
Modifier la position de vertices à l'aide d'une texture.



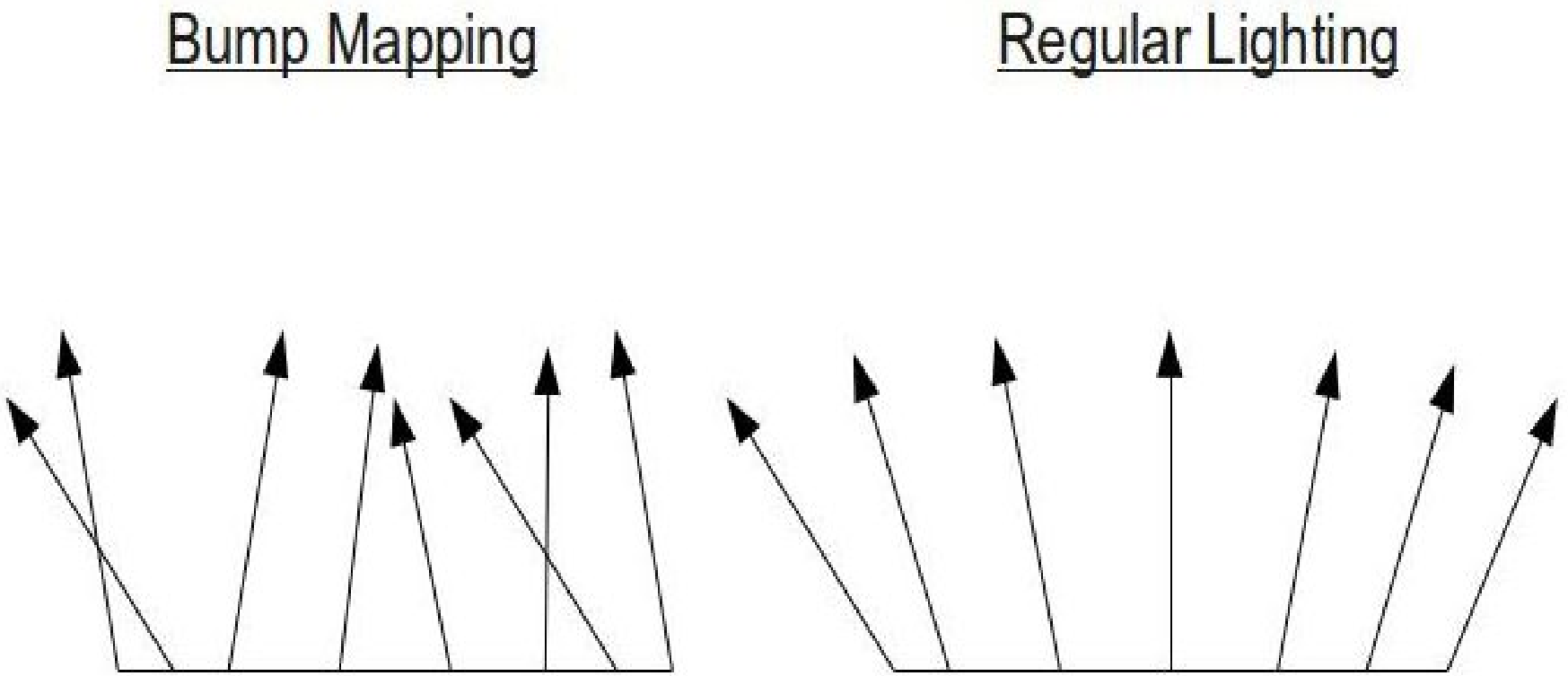


Le *Bump Mapping*

Simuler du relief sur une surface plane en modifiant les normales interpolées associées aux fragments à l'aide de normales stockées dans une texture.

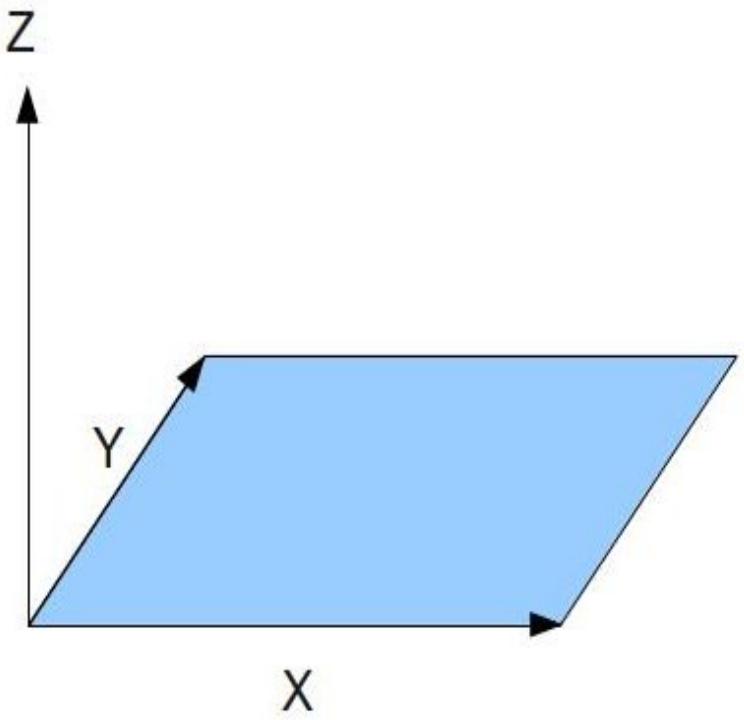


Le *Bump Mapping*

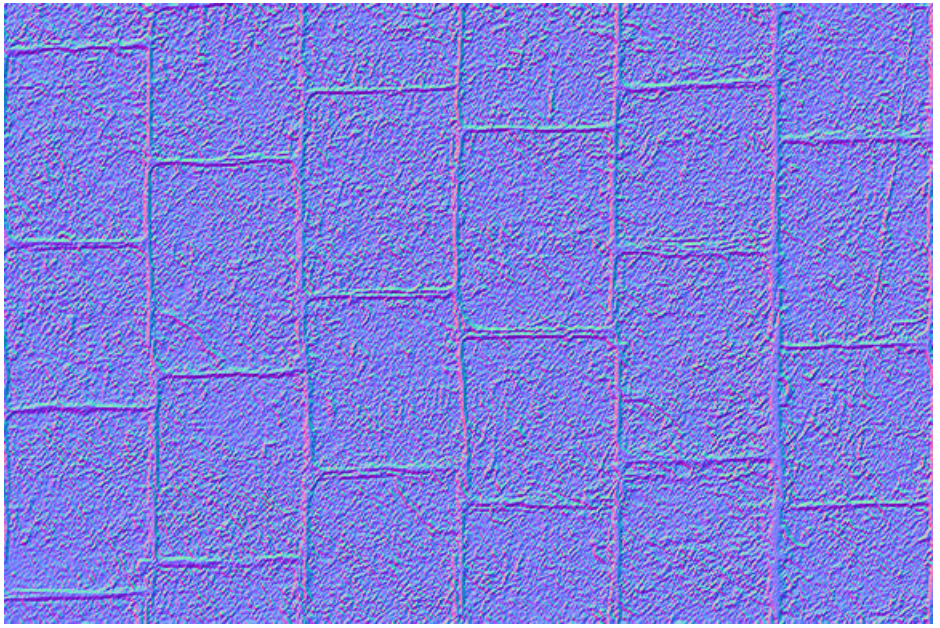


Normal Map

Le *Bump Mapping*

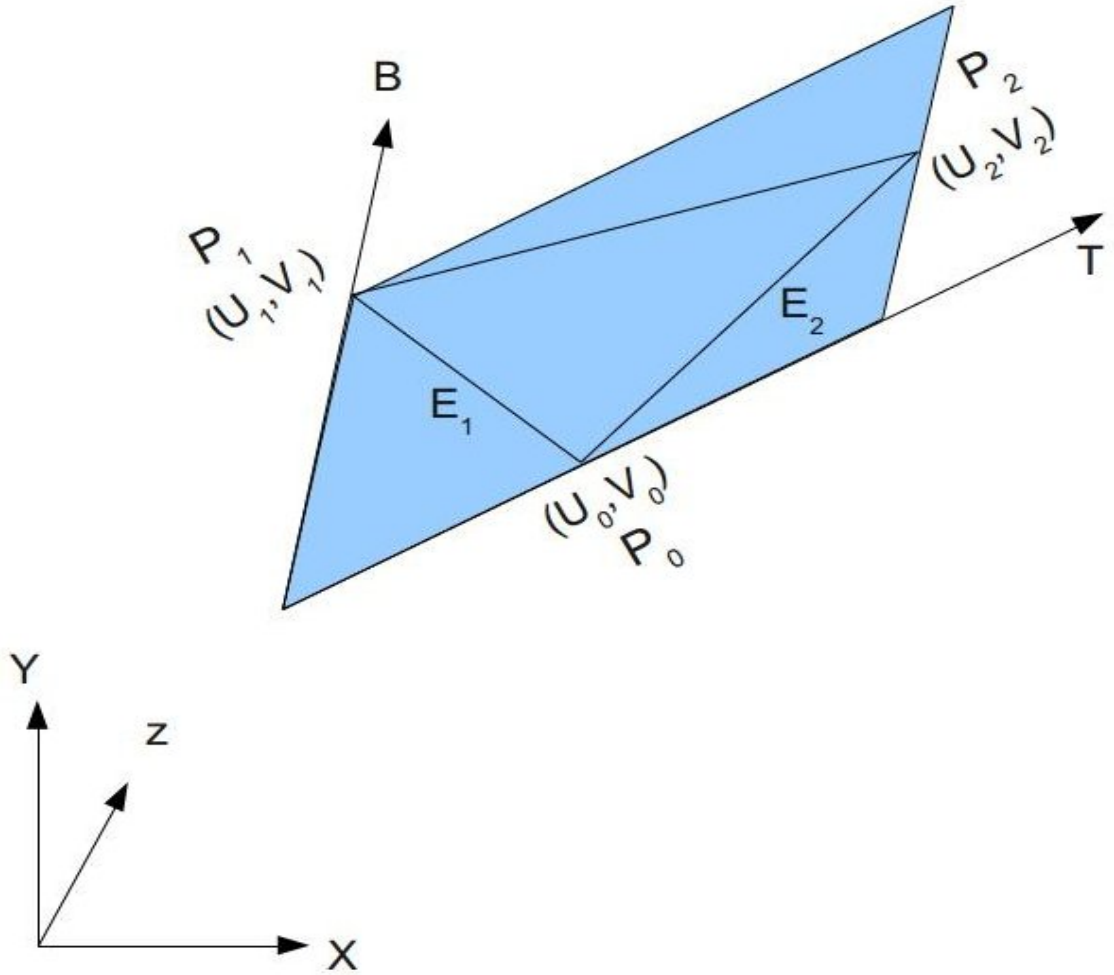


Système de coordonnées



Normal map

Le *Bump Mapping*



Détermination de l'espace Tangent

$$E_1 = (U_1 - U_0)T + (V_1 - V_0)B$$
$$E_2 = (U_2 - U_0)T + (V_2 - V_0)B$$

$$(E_{1_x}, E_{1_y}, E_{1_z}) = \Delta U_1 (T_x, T_y, T_z) + \Delta V_1 (B_x, B_y, B_z)$$
$$(E_{2_x}, E_{2_y}, E_{2_z}) = \Delta U_2 (T_x, T_y, T_z) + \Delta V_2 (B_x, B_y, B_z)$$

$$\begin{pmatrix} E_{1_x} & E_{1_y} & E_{1_z} \\ E_{2_x} & E_{2_y} & E_{2_z} \end{pmatrix} = \begin{pmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{pmatrix} \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix}$$

$$\begin{pmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{pmatrix}^{-1} \begin{pmatrix} E_{1_x} & E_{1_y} & E_{1_z} \\ E_{2_x} & E_{2_y} & E_{2_z} \end{pmatrix} = \begin{pmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{pmatrix} \begin{pmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{pmatrix}^{-1} \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix}$$

$$\begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix} = \begin{pmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{pmatrix}^{-1} \begin{pmatrix} E_{1_x} & E_{1_y} & E_{1_z} \\ E_{2_x} & E_{2_y} & E_{2_z} \end{pmatrix}$$

Détermination de l'espace Tangent

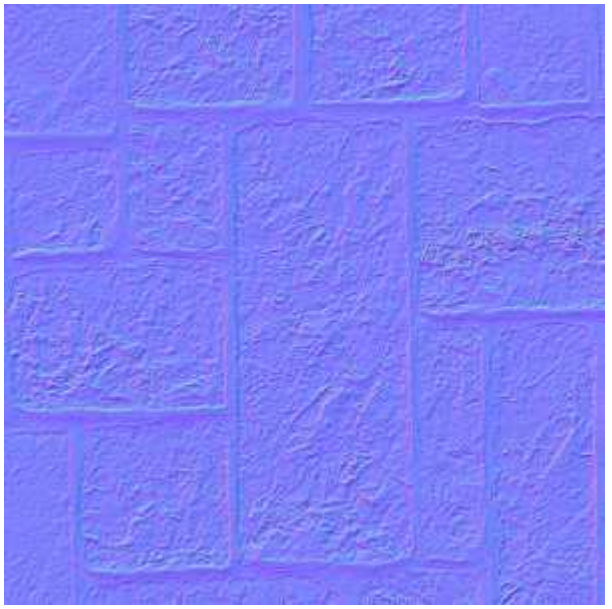
$$\begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{pmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{pmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{pmatrix} \begin{pmatrix} E_{1_x} & E_{1_y} & E_{1_z} \\ E_{2_x} & E_{2_y} & E_{2_z} \end{pmatrix}$$

Le *Bump Mapping*

Etape #1 : générer une *Normal Map* à partir d'une *Height Map*

Texture de base

Height Map



Normal Map

Des utilitaires existent (plugin dans *Gimp*)...

Le *Bump Mapping*

Etape #2 : constituer l'**espace tangente** associé à un vertex P pour déterminer la direction L' de la lumière relativement au vertex P de la surface

$$\begin{pmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{pmatrix}$$

T : tangente à la surface
B : bitangente (B = NxT)
N : normale

En plus de spécifier une normale par vertex, il est donc nécessaire de spécifier également une tangente...

Le *Bump Mapping*

Etape #3 : Calculer la normale perturbée N' en appliquant la matrice TBN sur la normale BN extraite de la *Normal Map* :

$$\begin{pmatrix} N'_x \\ N'_y \\ N'_z \end{pmatrix} = \begin{pmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{pmatrix} \begin{pmatrix} BN_x \\ BN_y \\ BN_z \end{pmatrix}$$

ou

$$N'_x = T.BN$$
$$N'_y = B.BN$$
$$N'_z = N.BN$$

Faire ensuite les calculs d'illumination avec N'.

Vertex Shader de *Bump Mapping*

```
layout (location = 0) in vec3 Position;
layout (location = 1) in vec2 TexCoord;
layout (location = 2) in vec3 Normal;
layout (location = 3) in vec3 Tangent;

uniform mat4 gWVP;
uniform mat4 gLightWVP;
uniform mat4 gNM;

out vec4 LightSpacePos;
out vec2 TexCoord0;
out vec3 Normal0;
out vec3 Tangent0;

void main()
{
    gl_Position = gWVP * vec4(Position, 1.0);
    LightSpacePos = gLightWVP * vec4(Position, 1.0);
    TexCoord0 = TexCoord;
    Normal0 = (gNM * vec4(Normal, 0.0)).xyz;
    Tangent0 = (gNM * vec4(Tangent, 0.0)).xyz;
}
```

Fragment Shader de *Bump Mapping*

```
vec3 getBumpedNormal()
{
    vec3 Normal = normalize(Normal0);
    vec3 Tangent = normalize(Tangent0);
    Tangent = normalize(Tangent - dot(Tangent, Normal) * Normal);
    vec3 Bitangent = cross(Tangent, Normal);
    vec3 BumpMapNormal = texture(gNormalMap, TexCoord0).xyz;

    vec3 NewNormal;
    mat3 TBN = transpose(mat3(Tangent, Bitangent, Normal));
    NewNormal = TBN * BumpMapNormal;
    NewNormal = normalize(NewNormal);
    return NewNormal;
}

void main()
{
    vec3 Normal = getBumpedNormal();
    ...
```

Plan du cours

- *Color Conversion*
- *Traitement d'images*
- *Cel-shading*
- **Textures spécifiques**
- **Textures procédurales**

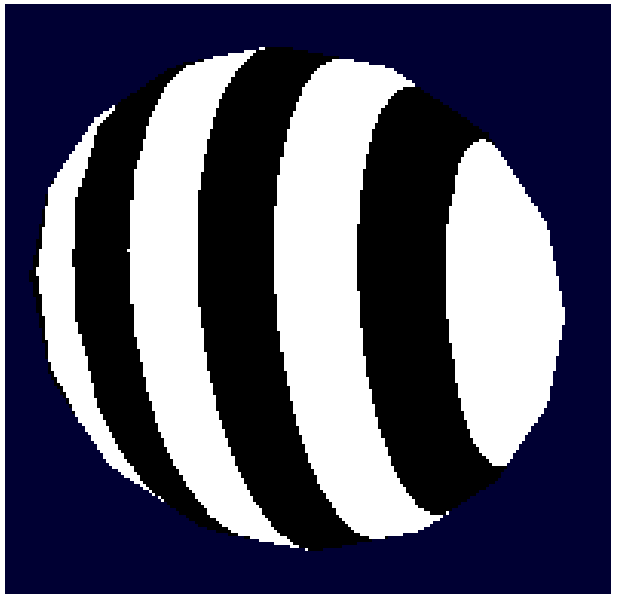
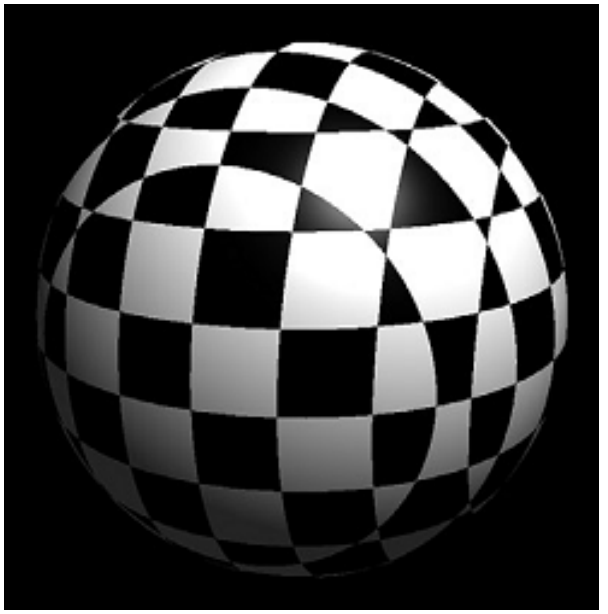
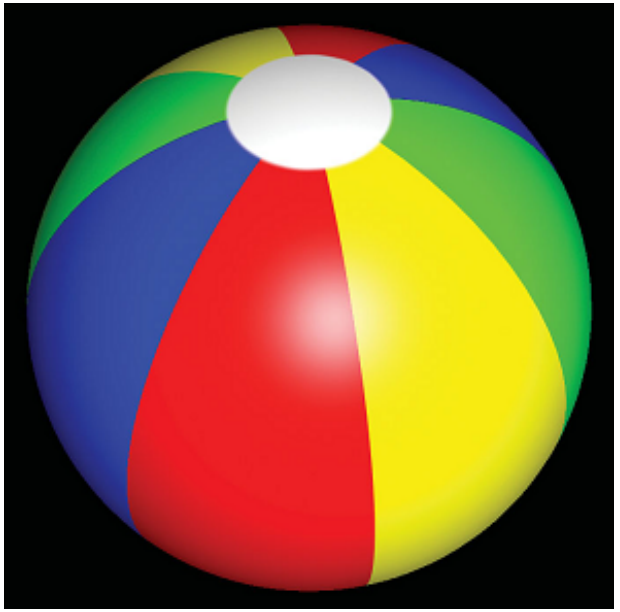
Principe, avantages et inconvénients des textures procédurales

Principe :
Décrire algorithmiquement l'aspect de la texture à appliquer à une surface.

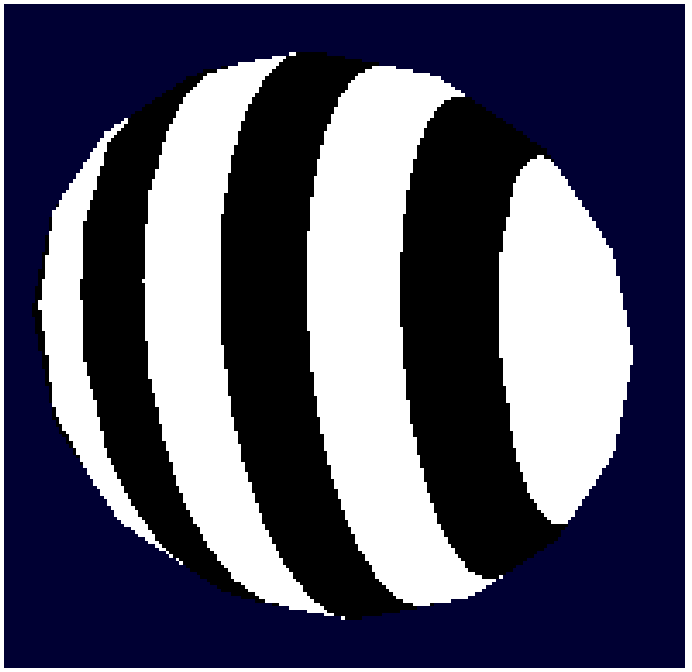
Avantages :
- Encombrement mémoire réduit
- Qualité du rendu

Inconvénient :
- Complexité éventuelle du shader associé

Exemples


Exemple #1



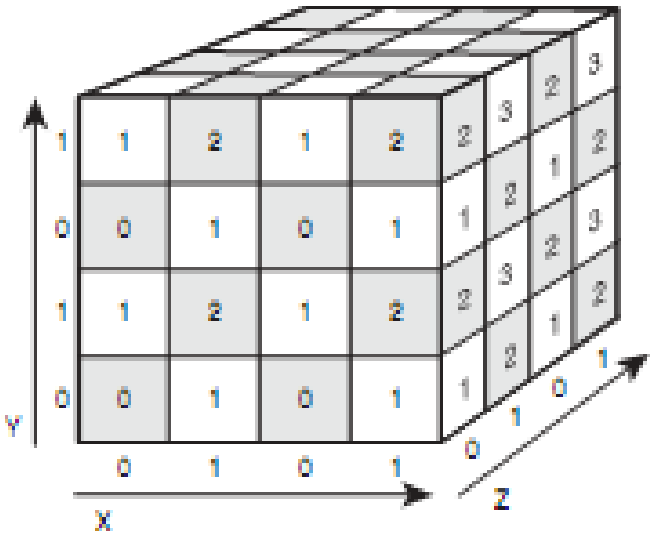
Exemple #1 : Vertex *Shader*

```
in vec3 pos;

uniform mat4 mvp ;

out vec3 V;

void main()
{
1     gl_Position = mvp*vec4(pos,1.0) ;

2     V = pos ;
}
```

Exemple #1 : *Fragment Shader*

```
const vec3 onColor = vec3(1.0, 1.0, 1.0);
const vec3 offColor = vec3(0.0, 0.0, 0.0);

const int numSquaresPerSide = 8;

in vec3 V; // object-space position

out vec4 fColor;

void main()
{
1     vec3 NV = normalize(V);

      // Map -1,1 to 0,numSquaresPerSide
2     float onOrOff = ((NV.z + 1.0) * float(numSquaresPerSide)) / 2.0;
      // mod 2 >= 1
3     onOrOff = step(1.0, mod(onOrOff, 2.0));

5     fColor = mix(offColor, onColor, onOrOff);
}
```

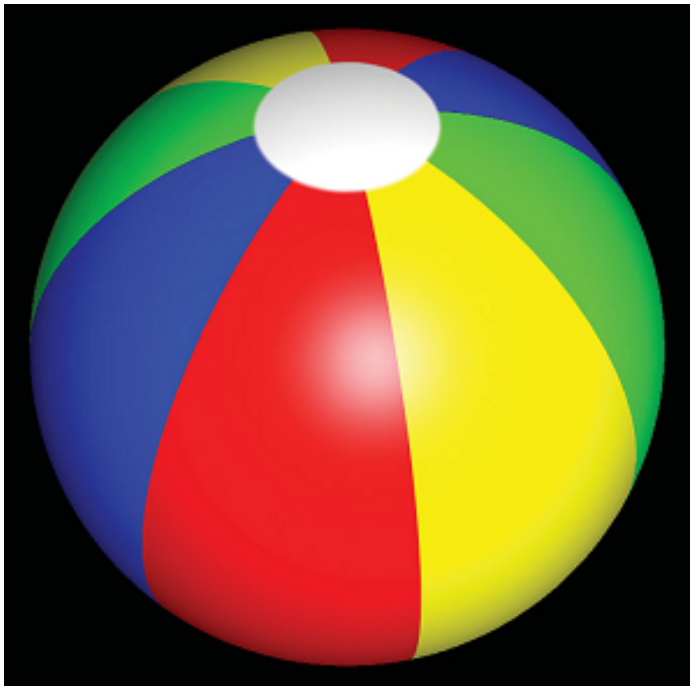
Exemple #2

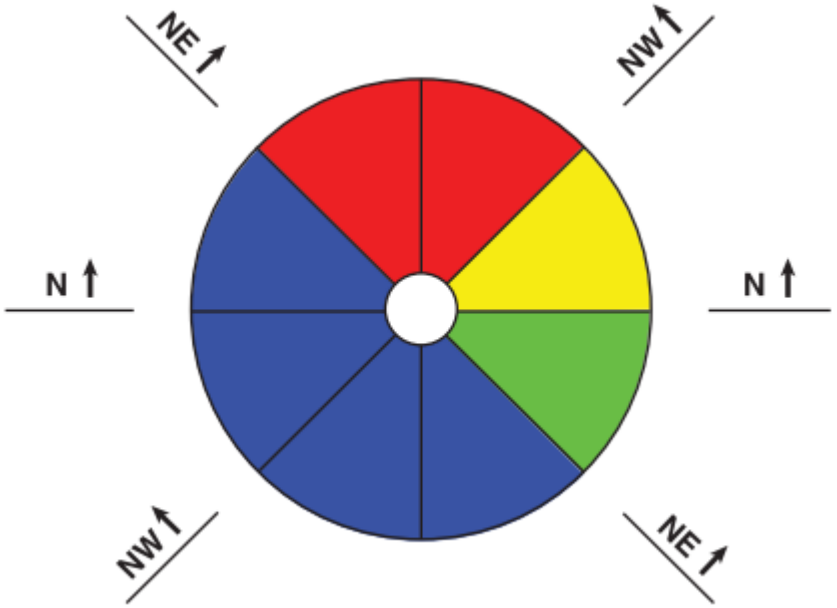


Exemple #2 : *Fragment Shader*

```
const vec3 onColor = vec3(1.0, 1.0, 1.0);
const vec3 offColor = vec3(0.0, 0.0, 0.0);

const int numSquaresPerSide = 8;

in vec3 V; // object-space position

out vec4 fColor;

void main()
{
1     vec3 NV = normalize(V);

      // Map -1,1 to 0,numSquaresPerSide
2     vec3 onOrOff = ((NV + 1.0) * float(numSquaresPerSide)) / 2.0;
      // mod 2 >= 1
3     onOrOff = step(1.0, mod(onOrOff, 2.0));
      // 3-way xor
4     onOrOff.x = step(0.5, mod(onOrOff.x + onOrOff.y + onOrOff.z, 2.0));

5     fColor = mix(offColor, onColor, onOrOff.x);
}
```

Exemple #3





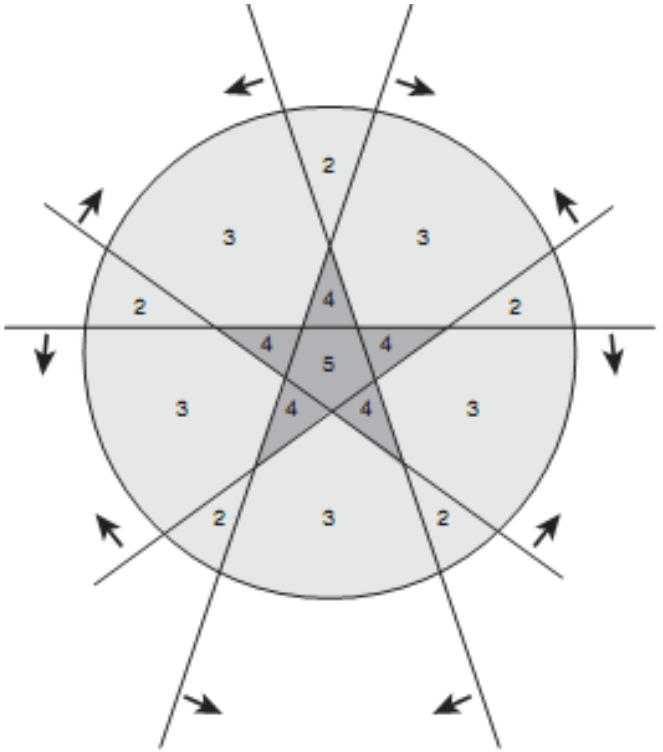
Exemple #3 : *Fragment Shader*

```
const vec3 myRed = vec3(1.0, 0.0, 0.0);
const vec3 myYellow = vec3(1.0, 1.0, 0.0);
const vec3 myGreen = vec3(0.0, 1.0, 0.0);
const vec3 myBlue = vec3(0.0, 0.0, 1.0);
const vec3 myWhite = vec3(1.0, 1.0, 1.0);
const vec3 myBlack = vec3(0.0, 0.0, 0.0);

const vec3 northHalfSpace = vec3(0.0, 0.0, 1.0);
const vec3 northeastHalfSpace = vec3(0.707, 0.0, 0.707);
const vec3 northwestHalfSpace = vec3(-0.707, 0.0, 0.707);
const float capSize = 0.03;
const float smoothEdgeTol = 0.005;

in vec3 V; // object-space position

out vec4 fColor;
```

Exemple #3 : *Fragment Shader*

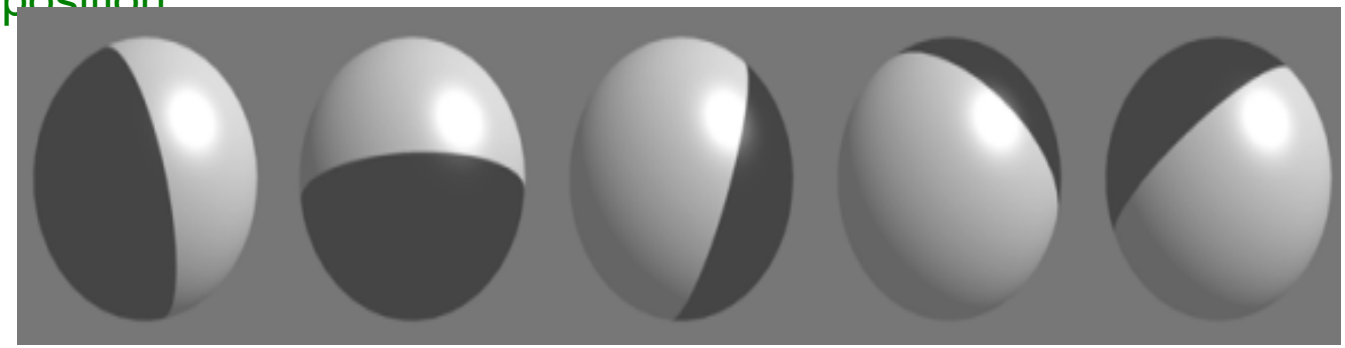
```
void main()
{
1     vec3 NV = normalize(V);
      // Mirror half of ball across X and Z axes
2     float mirror = (NV.x >= 0.0) ? 1.0 : -1.0;
3     NV.xz *= mirror;

      // Check for north/south, east/west,
      // northeast/southwest, northwest/southeast
4     vec4 distance;
5     distance.x = dot(NV, northHalfSpace);
6     distance.y = dot(NV, northeastHalfSpace);
7     distance.z = dot(NV, northwestHalfSpace);
      // set up for white caps on top and bottom
8     distance.w = abs(NV.y) - 1.0 + capSize;

9     distance = smoothstep(vec4(0.0), vec4(smoothEdgeTol), distance);

      // red, green, red+green=yellow, and blue stripes
10    vec3 surfColor = mix(myBlack, myRed, distance.x);
11    surfColor += mix(myBlack, myGreen, distance.y*(1.0-distance.z));
12    surfColor = mix(surfColor, myBlue, 1.0-distance.y);
      // white caps on top and bottom
13    fColor = mix(surfColor, myWhite, distance.w);
}
```

Exemple #4



Exemple #4 : *Fragment Shader*

```
const vec3 myRed = vec3(0.6, 0.0, 0.0);
const vec3 myYellow = vec3(0.6, 0.5, 0.0);
const vec3 myBlue = vec3(0.0, 0.3, 0.6);
const vec3 myHalfSpace0 = vec3(0.31, 0.95, 0.0);
const vec3 myHalfSpace1 = vec3(-0.81, 0.59, 0.0);
const vec3 myHalfSpace2 = vec3(-0.81, -0.59,
0.0);
const vec3 myHalfSpace3 = vec3(0.31, -0.95, 0.0);
const vec3 myHalfSpace4 = vec3(1.0, 0.0, 0.0);

const float stripeThickness = 0.4;    // 0 to 1
const float starSize = 0.2;           // 0 to ~0.3
const float smoothEdgeTol = 0.005;

in vec3 V;     // object-space position

out vec4 fColor;
```



Exemple #4 : *Fragment Shader*

```
void main()
{
1     vec4 distVector;
2     float distScalar;

3     vec3 NV = normalize(V);

4     float myInOut = -3.0;

      // We need to perform 5 dot products, one for each edge of
      // the star.  Perform first 4 in vector, 5th in scalar.
5     distVector.x = dot(NV, myHalfSpace0);
6     distVector.y = dot(NV, myHalfSpace1);
7     distVector.z = dot(NV, myHalfSpace2);
8     distVector.w = dot(NV, myHalfSpace3);
9     distScalar = dot(NV, myHalfSpace4);

      // The half-space planes all intersect the origin.  We must
      // offset them in order to give the star some size.
10    distVector += starSize;
      distScalar += starSize;
```

Exemple #4 : *Fragment Shader*

```
11   distVector = smoothstep(0.0, smoothEdgeTol, distVector);
     distScalar = smoothstep(0.0, smoothEdgeTol, distScalar);

12   myInOut += dot(distVector, vec4(1.0));
     myInOut += distScalar;
13   myInOut = clamp(myInOut, 0.0, 1.0);

     // red star on yellow background
14   vec3 surfColor = mix(myYellow, myRed, myInOut);
     // blue stripe down middle
15   myInOut = smoothstep(0.0, smoothEdgeTol, abs(NV.z) - stripeThickness);

16   fColor = mix(myBlue, surfColor, myInOut);
}
```