

# Chapitre 5 : Rendu projectif en OpenGL

## Vertex et Fragment shaders

Modélisation 3D et Synthèse

Fabrice Aubert  
fabrice.aubert@lifl.fr



IEEA - Master Info - Parcours IVI

2012-2013

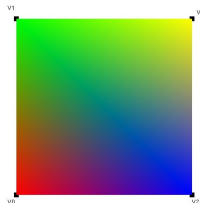
- ▶ Introduction aux techniques dites modernes des bibliothèques de prog3D (OpenGL/Direct3D) : buffer objects, shaders.
- ▶ On introduit également, dans le contexte OpenGL, des concepts fondamentaux pour la visualisation 3D : l'éclairage et les textures.

# 1 Tracé des polygones

# Mode immédiat

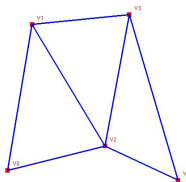
- Deprecated depuis 3.0 (compatibility profile)

```
glBegin(GL_TRIANGLE_STRIP);  
    // attributs sommet V0  
    glColor3f(1,0,0);  
    glVertex3f(-1,-1,0);  
    // attributs sommet V1  
    glColor3f(0,1,0);  
    glVertex3f(-1,1,0);  
    // attributs sommet V2  
    glColor3f(0,0,1);  
    glVertex3f(1,-1,0);  
    // attributs sommet V3  
    glColor3f(1,1,0);  
    glVertex3f(1,1,0);  
glEnd();
```



Rappel : ordre par  
TRIANGLE\_STRIP :

```
glBegin(GL_TRIANGLE_STRIP);  
    glVertex... (V0);  
    glVertex... (V1);  
    glVertex... (V2);  
    glVertex... (V3);  
    glVertex... (V4);  
    ...  
glEnd();
```



- Tracé moderne : donner les sommets par buffer.

# Array Buffer

- ▶ Affecter une zone mémoire d'OpenGL (server side memory = carte graphique si possible) avec des données => buffer objects.
- ▶ Plusieurs type de buffers : pour les attributs de sommets on utilise les array buffers (tableaux). Technique appelée anciennement Vertex Buffer Objects.
- ▶ On copie les données de l'application (client side memory) dans la mémoire OpenGL.

```
unsigned int bufferVertex;  
  
void Square::initBuffer() {  
  
    // vertex[]={x0,y0,z0,x1,y1,z1,x2,y2,z2, etc}  
    float vertex[]={-1,-1,0,-1,1,0,1,-1,0,1,1,0};  
  
    // générer un "pointeur" (ou identifiant) dans une zone mémoire OpenGL  
    glGenBuffers(1,&bufferVertex);  
  
    // le tableau de donnée actif correspond à l'identifiant bufferVertex  
    // (un seul array buffer actif à la fois)  
    glBindBuffer(GL_ARRAY_BUFFER, bufferVertex);  
  
    // affectation des données server avec les donnée clients  
    // (ici 12 float dont copiés depuis le tableau vertex).  
    glBufferData(GL_ARRAY_BUFFER,12*sizeof( GLfloat), vertex, GL_STATIC_DRAW);  
}
```

Bind -> on va travailler avec quelque chose  
GL\_ARRAY\_BUFFER: un tableau  
bufferVertex: le tableau avec lequel on va travailler

dit a carte graph de optimiser des valeurs du buffer

les données a écrire sont du type  
GL\_ARRAY\_BUFFER donc on écrit dans bufferVertex

# Tracer les données des array buffers (OpenGL 2.1)

## ► OpenGL 2.1 ? Pourquoi ?

```
void Square::drawBuffer() {  
    // activer l'alimentation de l'attribut gl_Vertex pour chaque sommet tracé  
    glEnableClientState(GL_VERTEX_ARRAY);  
  
    // le buffer courant des données :  
    glBindBuffer(GL_ARRAY_BUFFER, bufferVertex);  
  
    // les attributs gl_Vertex seront lus depuis le buffer bufferVertex (i.e. le buffer courant) :  
    glVertexPointer(3, GL_FLOAT, 0, 0);  
  
    // commande de tracé (exécution du pipeline de tracé) avec 4 sommets :  
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);  
  
    glDisableClientState(GL_VERTEX_ARRAY);  
}
```

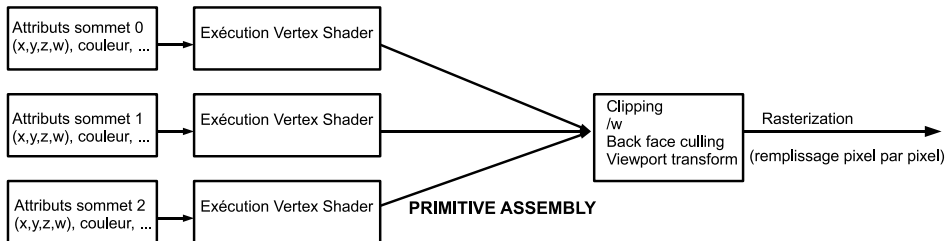
le nombre d'espace qu'il y a entre 2 sommets consecutifs  
quand commence les données a dessiner  
Ou trouvé les vertex dans la zone mémoire de OpenGL  
Et sur quoi le faire -> dernier glBindBuffer

- ▶ Toujours se rappeler qu'OpenGL "travaille" avec les valeurs courants des états (`glBindBuffer(...)` indique par exemple le buffer courant actif).
- ▶ Ne pas oublier de gérer correctement la mémoire si besoin (cf tous les `glDelete(...)`, par exemple `glDeleteBuffers`).

# Pipeline de tracé

/w : on repasse en mode normale

clipping : on limite la vision de la figure a celle de la vue



⇒ tout sommet provoque l'exécution d'un programme appelé Vertex Shader.



# Vertex Shader

- ▶ **Chaque** sommet subit le vertex shader.
- ▶ = programme exécuté (par la carte graphique si possible).
- ▶ = programme écrit dans un langage particulier : GLSL (OpenGL Shading Language).
- ▶ OpenGL permet de compiler/linker et activer des programmes GLSL

```
#version 110

void main() {
    gl_Position=gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
}
```

- ▶  $\Rightarrow$  langage inspiré de C/C++
- ▶ Tous les états sont accessibles par des "variables" identifiées par `gl_`.
  - `gl_Vertex` : de type `vec4` correspond aux coordonnées  $(x, y, z, w)$  (fourni par les données lues dans le `ARRAY_BUFFER`).
  - `gl_ModelViewMatrix` : de type `mat4` correspond à la valeur de la `MODELVIEW`.
  - `gl_Projection` : de type `mat4` correspond à la valeur de la matrice `PROJECTION`.
- ▶ Le vertex shader doit fournir obligatoirement `gl_Position` à la suite du pipeline.

- Tout le langage GLSL est résumé sur les 4 dernières pages de  
<http://www.khronos.org/files/opengl-quick-reference-card.pdf>

## The OpenGL Shading Language 1.50 Quick Reference Card

The OpenGL® Shading Language is several closely-related languages which are used to create shaders for each of the programmable processors contained in the OpenGL processing pipeline.

[n.n.n] and [Table n.n] refer to sections and tables in the specification at [www.opengl.org/registry](http://www.opengl.org/registry)

Content shown in blue is removed from the OpenGL 3.2 core profile and present only in the OpenGL 3.2 compatibility profile.

### Types [4.1.1-4.1.10]

#### Transparent Types

<b>void</b>	no function return value
<b>bool</b>	Boolean
<b>int, uint</b>	signed and unsigned integers
<b>float</b>	floating scalar
<b>vec2, vec3, vec4</b>	floating point vector
<b>bvec2, bvec3, bvec4</b>	Boolean vector
<b>ivec2, ivec3, ivec4</b>	signed and unsigned integer vector
<b>uvec2, uvec3, uvec4</b>	
<b>mat2, mat3, mat4</b>	2x2, 3x3, 4x4 float matrix
<b>mat2x2, mat2x3, mat2x4</b>	2-column float matrix with 2, 3, or 4 rows
<b>mat3x2, mat3x3, mat3x4</b>	3-column float matrix with 2, 3, or 4 rows
<b>mat4x2, mat4x3, mat4x4</b>	4-column float matrix with 2, 3, or 4 rows

#### Floating-Point Sampler Types (Opaque)

<b>sampler1D, 1D</b>	access a 1D, 2D, or 3D texture
<b>samplerCube</b>	access cube mapped texture
<b>sampler2DRect</b>	access rectangular texture
<b>sampler1DShadow</b>	access 1D or 2D depth texture/comparison
<b>sampler2DRectShadow</b>	access rectangular texture/comparison
<b>sampler1DArray</b>	access 1D or 2D array texture
<b>sampler1DArrayShadow</b>	access 1D or 2D array depth texture/comparison
<b>samplerBuffer</b>	access buffer texture
<b>sampler2DMS</b>	access 2D multi-sample texture
<b>sampler2DMSArray</b>	access 2D multi-sample array texture

#### Integer Sampler Types (Opaque)

<b>isampler1D, 1D</b>	access integer 1D, 2D, or 3D texture
-----------------------	--------------------------------------

### Preprocessor [3.3]

#### Preprocessor Operators

Preprocessor operators follow C++ standards. Preprocessor expressions are evaluated according to the behavior of the host processor, not the processor targeted by the shader.

#### #version 150

#version 150 compatibility

#extension extension\_name : behavior

#extension all : behavior

"#version 150" is required in shaders using version 1.50 of the language. #version must occur in a shader before anything else other than white space or comments. Use "compatibility" to access features in the compatibility profile.

- behavior: require, enable, warn, disable
- extension\_name: the extension supported by the compiler, or "all"

### Preprocessor Directives

Each number sign (#) can be preceded in its line only by spaces or horizontal tabs.

#	#define	#undef	#if	#ifdef
#ifndef	#else	#elif	#endif	#error
#pragma	#extension	#version	#line	

### Predefined Macros

__LINE__	__FILE__	Decimal integer constants	__VERSION__	Decimal integer, e.g.: 150
----------	----------	---------------------------	-------------	----------------------------

### Qualifiers

#### Storage Qualifiers [4.3]

Variable declarations may have one storage qualifier.

<b>none</b>	(default) local read/write memory, or input parameter
<b>const</b>	compile-time constant, or read-only function parameter
<b>in</b>	linkage into a shader from previous stage (copied in)
<b>centroid in</b>	linkage with centroid based interpolation
<b>out</b>	linkage out of a shader to subsequent stage (copied out)
<b>centroid out</b>	linkage with centroid based interpolation
<b>uniform</b>	linkage between a shader, OpenGL, and the application

#### Uniform [4.3.5]

Use to declare global variables with the same values across the entire primitive being processed. Uniform variables are read-only. Use uniform qualifiers with any basic data types or array of these, or when declaring a variable whose type is a structure, e.g.:

```
uniform vec4 lightPosition;
```

#### Layout Qualifiers [4.3.8]

**layout(layout-qualifiers) block-declaration**  
**layout(layout-qualifiers) in/out/uniform**  
**layout(layout-qualifiers) in/out/uniform declaration**

Input Layout Qualifiers

#### Interpolation Qualifier [4.3.9]

Quality outputs from vertex shader and inputs to fragment shader.

<b>smooth</b>	perspective correct interpolation
<b>flat</b>	no interpolation
<b>noperspective</b>	linear interpolation

The following predeclared variables can be redeclared with an interpolation qualifier:

<b>Vertex language:</b>	<b>gl_FrontColor</b> <b>gl_BackColor</b> <b>gl_FrontSecondaryColor</b> <b>gl_BackSecondaryColor</b>
<b>Fragment language:</b>	<b>gl_Color</b> <b>gl_SecondaryColor</b>

#### Parameter Qualifiers [4.4]

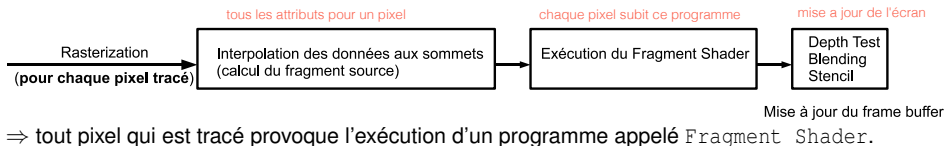
Input values are copied in at function call time, output values are copied out at function return time.

<b>none</b>	(default) same as in
<b>in</b>	for function parameters passed into a function
<b>out</b>	for function parameters passed back out of a function, but not initialized for use when passed in
<b>inout</b>	for function parameters passed both into and out of a function

#### Precision and Precision Qualifiers [4.5]

Precision qualifiers have no effect on precision; they aid code portability with OpenGL ES. They are:  
 highp, mediump, lowp

# Rasterization



# Fragment shader

- ▶ Même langage que vertex shader : GLSL
- ▶ Doit fournir une couleur pour le pixel qui est en train d'être tracé (affecter `gl_FragColor` de type `vec4`).

Exemple de base (listing du fichier `essai.frag`) :

```
#version 110

void main() {
    // affectation avec du vert (i.e. vec4 interprété comme (rouge,vert,bleu,alpha)).
    gl_FragColor=vec4(0.0,1.0,0.0,0.0);
}
```

R G B Transparence

# Compilation et activation d'un shader

## ► Compilation/link par l'application OpenGL :

```
void create() {
    programId=glCreateProgram();

    vertexId=glCreateShader(GL_VERTEX_SHADER);
    fragmentId=glCreateShader(GL_FRAGMENT_SHADER);

    glAttachShader(programId, vertexId);
    glAttachShader(programId, fragmentId);

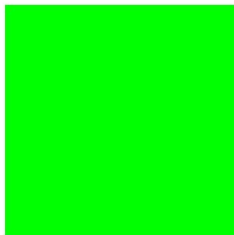
    char *source=readFile("essai.vert");
    glShaderSource(vertexId,1,&source,NULL);
    char *source=readFile("essai.frag");
    glShaderSource(fragmentId,1,&source,NULL);

    glCompileShader(vertexId);
    glCompileShader(fragmentId);

    glLinkProgram(programId);
}
```

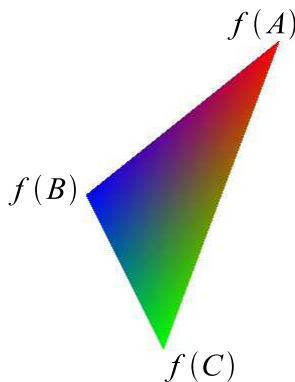
## ► Utilisation (activation pour tous les tracés effectués par draw() par exemple) :

```
void drawScene() {
    ...
    glUseProgram(programId);
    draw(); correspond au draw du carré précédent
    ...
}
```



# Interpolation de variables : `varying`

- ▶ Le vertex shader peut calculer des valeurs qui seront fournies à la suite du pipeline (variables en sortie).
- ▶ Le fragment shader peut récupérer ces variables dont les valeurs ont subi une interpolation bilinéaire par rapport aux valeurs de chacun des sommets (variables en entrée).
- ▶ Ces variables, qui apparaissent avec le même nom dans le vertex et le fragment, sont qualifiées de `varying`.



# Exemple : interpolation des couleurs

## Vertex Shader :

```
#version 110

varying vec4 couleur;

void main() {
    couleur=gl_Color; // gl_Color est alimenté par le buffer d'attribut GL_COLOR_ARRAY
    gl_Position=gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
}
```

## Fragment Shader :

```
#version 110

varying vec4 couleur;

void main() {
    gl_FragColor=couleur;
}
```

## Tracé OpenGL :

```
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, bufferVertex);
glVertexPointer(3, GL_FLOAT, 0, 0);

glEnableClientState(GL_COLOR_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, bufferColor);
glColorPointer(3, GL_FLOAT, 0, 0);

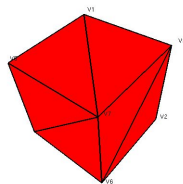
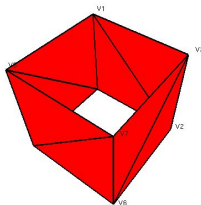
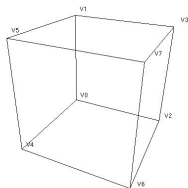
glDrawArrays(GL_TRIANGLE, 0, 3);
```



# Tracé de faces avec indices

```
float cube[]={-1,-1,-1, // Vertex 0
              -1,1,-1,   // Vertex 1
              1,-1,-1,   // Vertex 2
              1,1,-1,    // Vertex 3
              -1,-1,1,   // Vertex 4
              -1,1,1,    // Vertex 5
              1,-1,1,    // Vertex 6
              1,1,1,     // Vertex 7
              };

unsigned int indice[]={0,1,2,3,6,7,4,5,0,1,
                      5,7,1,3, // "couvercle dessus"
                      6,4,2,0}; // "couvercle dessous"
```



# Exemple du cube : initialisation buffers

```
GLuint vertexCube, indiceCube;
void initBufferCube() {
    float cube[]={-1,-1,-1,-1,1,-1,1,-1,-1,1,1,-1,-1,-1,1,1,-1,1,1,1};

    unsigned int indice[]={0,1,2,3,6,7,4,5,0,1,
                           5,7,1,3,
                           6,4,2,0};

    glGenBuffers(1,&vertexCube);
    glBindBuffer(GL_ARRAY_BUFFER, vertexCube);
    glBufferData(GL_ARRAY_BUFFER, 24*sizeof(GLfloat), cube, GL_STATIC_DRAW);

    glGenBuffers(1,&indiceCube);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indiceCube);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, 18*sizeof(GLuint), indice, GL_STATIC_DRAW);
}
```


# Exemple du cube : tracé

```
glEnableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, vertexCube);
glVertexPointer(3, GL_FLOAT, 0, 0);

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indiceCube);

// style de primitives, nombre de sommets, type des indices, début de lecture dans les indices (en unit machine)
glDrawElements(GL_TRIANGLE_STRIP, 10, GL_UNSIGNED_INT, (void*)(sizeof(GLuint)*0));
glDrawElements(GL_TRIANGLE_STRIP, 4, GL_UNSIGNED_INT, (void*)(sizeof(GLuint)*10));
glDrawElements(GL_TRIANGLE_STRIP, 4, GL_UNSIGNED_INT, (void*)(sizeof(GLuint)*14));

glDisableClientState(GL_VERTEX_ARRAY);
```

 Le type des indices

Pour le cube, le `GL_TRIANGLE_STRIP` n'est pas nécessairement bien adapté.

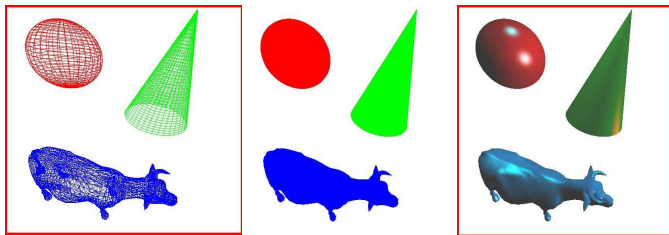
- ▶ On peut également spécifier tous les triangles dans le tableau d'indices.
- ▶ Puis tracer avec un seul

```
glDrawElements(GL_TRIANGLES, ??, GL_UNSIGNED_INT, (void *)0).
```

Exercice : combien de triangles à tracer ? quel est alors le tableau d'indices (indiquez le début pour les 3 premiers triangles) ?

## 2 Eclaircement

- Pour ajouter du réalisme aux scènes 3D, OpenGL propose de calculer des couleurs pour simuler un éclairage de la scène par des sources lumineuses.



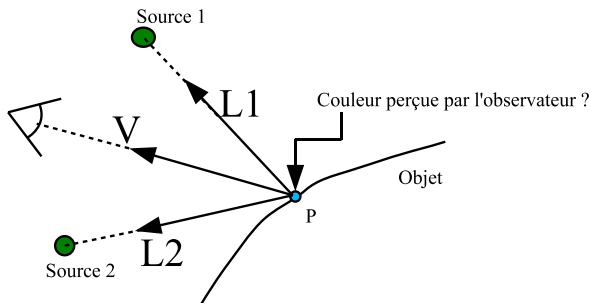
⇒ calcul de la couleur provenant de 2 contributions : réflexion diffuse (couleur mat) et réflexion spéculaire (couleur brillante : tâche spéculaire).

Les données nécessaires au calcul d'éclairage sont :

- ▶ Les sources (position, caractéristiques d'éclairage).
- ▶ Le matériel des objets (caractéristiques qui traduisent comment est réfléchi la lumière des sources).
- ▶ On peut avoir jusqu'à 8 sources simultanées en OpenGL. Elles sont identifiées par les constantes `GL_LIGHT0`, `GL_LIGHT1`, ..., `GL_LIGHT7`.
- ▶ L'instruction qui permet de donner les caractéristiques des sources : `glLight`.
- ▶ Il existe **deux seuls** matériels (un pour les faces FRONT et un pour les BACK). Il faut donc changer les caractéristiques dès qu'on veut tracer un objet avec un matériel différent de l'objet précédemment tracé.
- ▶ L'instruction qui permet de donner les caractéristiques des matériels : `glMaterial`.
- ▶ Les composantes (rouge, vert, bleu) qui apparaissent dans la suite sont comprises entre 0 et 1.

# Modèle d'éclairage

- ▶ Dans ce chapitre, pour illustrer l'éclairage on choisit le modèle empirique de Phong.
- ▶ Très simple, mais aussi très éloigné de la réalité.

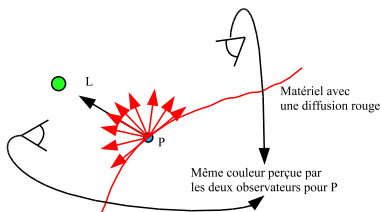


- ▶ Le vecteur  $V$  est appelé vecteur d'observation, le vecteur  $L$  est appelé vecteur d'éclairage.
- ▶ La position de la source 0 est donnée par  
`glLightfv(GL_LIGHT0, GL_POSITION, <float *pos>)` où  $pos=(x,y,z,w)$ .



# Réflexion diffuse 1/2

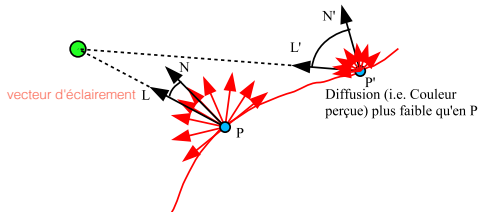
- ▶ On suppose qu'un objet (un matériel) diffuse la lumière reçue de manière uniforme dans toutes les directions.
- ▶ La lumière (i.e. la couleur) perçue ne dépend donc pas de la position de l'observateur.



- ▶ L'intensité diffusée dépend des caractéristiques du matériel (matériel rouge = « beaucoup » de rouge diffusé, matériel noir = aucune intensité diffusée, etc).
- ▶  $\Rightarrow$  Définition d'un coefficient de matériel  $k_d = (\text{rouge}, \text{vert}, \text{bleu})$  pour définir cette caractéristique (toujours  $\in [0, 1]$ ).

`glmMaterial()`

- ▶ L'intensité diffusée dépend de l'angle d'incidence des rayons lumineux sur la surface de l'objet
  - un éclairage direct (i.e. la lumière arrive orthogonalement à la surface) donne une diffusion maximale.
  - un éclairage fuyant (i.e tangent à la surface) donne un éclairage nul.
  - on souhaite que la diffusion varie « continuellement » entre ces 2 positions de la manière la plus réaliste possible.
  - $\Rightarrow$  prise en compte de la normale (i.e. vecteur orthogonal) à la surface au point  $P$ .



- ▶ On suppose  $N$  et  $L$  sont normés ( $\|N\| = \|L\| = 1$ ).
- ▶ Le calcul de l'intensité du diffus par  $\cos(N, L) = L \cdot N$  est un « bon » choix.
- ▶ On donnera une couleur de réflexion diffuse  $K_d$  pour indiquer la couleur réfléchie par le matériel.

$$\Rightarrow \boxed{\text{Couleur}_{\text{diffus}}(P) = K_d(N \cdot L)}$$

- ▶ seul le « coté » dirigé par la normale est éclairé (si  $N \cdot L < 0$  alors éclairage nul).
- ▶  $\Rightarrow$  important de spécifier correctement les normales.
- ▶ Composante  $K_d$ =(rouge,vert,bleu) couleur souhaitée pour le matériel.
- ▶ `glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, <GLfloat Kd[3]>);`

- ▶ En OpenGL, on se contente souvent de calculer l'éclairement diffus uniquement aux sommets.
- ▶  $\Rightarrow$  obtention d'une couleur en chaque sommet.
- ▶ La couleur des pixels lors de la rasterization est alors simplement obtenue par interpolation linéaire des couleurs.
- ▶ Le résultat est satisfaisant par rapport à un calcul sur chacun des points du triangle 3D (bien que ce calcul d'éclairement diffus ne soit pas linéaire sur l'espace écran !  $\Rightarrow$  approximation convenable).

$\Rightarrow$  l'interpolation linéaire des couleurs, dans le cadre de l'éclairement est appelée **interpolation de Gouraud**.

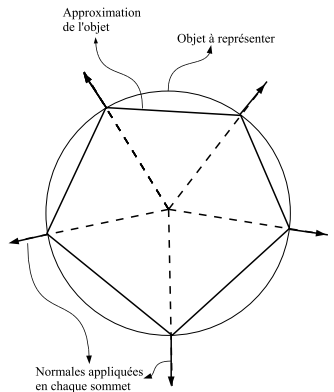
- Pour le calcul d'éclairement il faut spécifier les normales.
- Si le calcul s'effectue en chaque sommet, il faut fournir à chaque sommet une normale.

```
glEnableClientState(GL_VERTEX_ARRAY);  
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);  
glVertexPointer(3, GL_FLOAT, 0, 0);  
  
glEnableClientState(GL_NORMAL_ARRAY);  
glBindBuffer(GL_ARRAY_BUFFER, normalBuffer); // attribué auparavant  
glNormalPointer(GL_FLOAT, 0, 0);  
  
glDrawArrays(GL_TRIANGLES, ...);
```

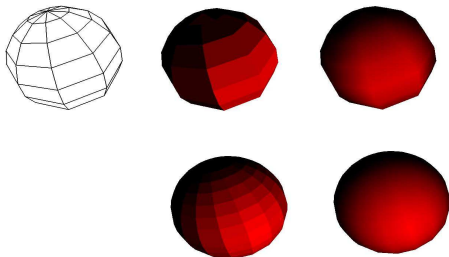
- Fournir le vecteur orthogonal au polygone tracé ?  $\Rightarrow$  pas nécessairement ! (liberté totale de donner la normale qu'on souhaite).

## Normales 2/3

- Pouvoir spécifier une normale différente en chaque sommet permet d'obtenir le calcul d'éclairage qui correspond au mieux à la forme souhaitée.
- $\Rightarrow$  permet d'obtenir une perception d'un objet lisse en « jouant » uniquement avec les normales !

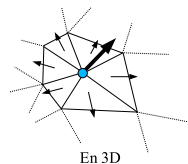
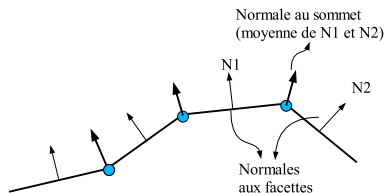


- Résultat pour le diffus :



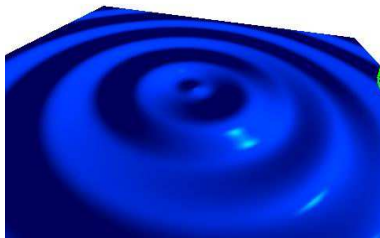
# Exemple de spécification des normales pour un objet complexe

- ▶ La normale de l'objet à représenter peut ne pas être connue (surface « réelle » inconnue).
- ▶  $\Rightarrow$  Prendre la moyenne des normales aux facettes incidentes au sommet peut donner une bonne approximation de la surface lisse.





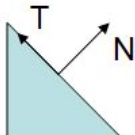
# Effets « spéciaux » avec des normales



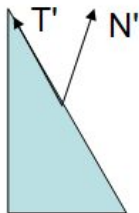
- ▶ sur l'animation on ne voit pas tous les polygones (grille plus finement subdivisée).
- ▶ il s'agit du principe appliqué par la technique dite du « Bump mapping » : spécifier les normales d'un relief sans toucher à la géométrie de l'objet. Seul le calcul d'éclaircement donne la perception de relief.

# Eclairement dans les shaders

- ▶ Tous les vecteurs apparaissant dans les calculs doivent être exprimés dans le même repère : les calculs d'éclairement se feront dans le repère Eye.
- ▶ Transformation des sommets : `vertexEye=gl_ModelViewMatrix*gl_Vertex;`
- ▶ Transformation des normales : `nEye=gl_NormalMatrix*gl_Normal;`. **Attention :** `gl_NormalMatrix` est une matrice 3x3 et `gl_Normal` est un `vec3`.
- ▶ Pourquoi ? A cause des éventuels scales :



avant transformation



après transformation par MODELVIEW

- ▶ La matrice correcte pour transformer les normales est  $(M^{-1})^t$  où  $M$  est la sous-matrice 3x3 (3 premières lignes, 3 premières colonnes) de MODELVIEW.

(cf <http://www.lighthouse3d.com/opengl/glsl/index.php?normalmatrix>)

# Shader

## Vertex shader :

```
#version 110

varying vec4 couleur;

void main() {
    float intensiteDiffus;

    vec3 N,L;

    N=gl_NormalMatrix*gl_Normal;

    // Source supposée directionnelle ici :
    L=gl_LightSource[0].position.xyz; // déjà exprimé dans le repère Eye

    L=normalize(L);
    N=normalize(N);

    intensiteDiffus=max(dot(N,L),0.0);

    couleur=intensiteDiffus*gl_FrontMaterial.diffuse;

    gl_Position=gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;
}
```

## Fragment shader :

```
#version 110

varying vec4 couleur;

void main() {
    gl_FragColor=couleur;
}
```

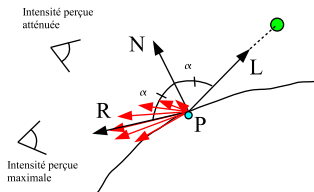
# Remarques sur GLSL

On dispose en GLSL des types propres à la 3D (vec, mat), et d'une manipulation assez souple des variables :

```
void main() {  
    vec4 a=vec4(0.1,0.2,0.3,0.4); // constructeur  
    float b=a.x; // accès au champ x  
    vec2 c=a.xy; // opération dite de sizzling retour un vector2 entre x et y  
    float d=a.w;  
    mat2 e = mat2(1.0,0.0,1.0,0.0);  
    float f=e[1][1]; // accès tableau  
    vec2 g=e[1]; // 2ieme colonne.  
    ...  
}
```

# Spéculaire

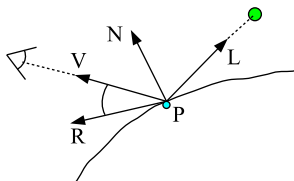
- ▶ La specularité traduit l'aspect « brillant » de l'objet.
- ▶ La réflexion spéculaire provient de la réflexion (au sens « miroir ») des rayons lumineux sur l'objet.
- ▶  $\Rightarrow$  on considère alors la direction miroir  $R$  du vecteur d'éclairage  $L$  ( $R$  est le symétrique de  $L$  par rapport à  $N$ ).
- ▶ L'intensité de la réflexion spéculaire est maximale dans la direction  $R$  et est atténuée autour de cette direction  $R$ .



$\Rightarrow$  L'intensité perçue (i.e. la couleur) par l'observateur va donc dépendre de sa position par rapport à la direction  $R$ .

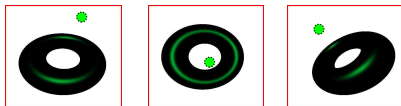
# Calcul spéculaire

- ▶ Le calcul de  $V \cdot R$  (cosinus de l'angle entre  $V$  et  $R$ ) donne une approximation correcte de l'effet spéculaire (maximal dans la direction si  $R$  dirigé directement sur l'observateur ; atténué autour).
- ▶ Comme pour le diffus : on affecte une caractéristique  $K_s = (\text{rouge}, \text{vert}, \text{bleu})$  pour le matériel.
- ▶ Ne pas oublier : tous les vecteurs normés ( $V \cdot R = \cos(V, R) \in [0, 1]$ ).



$$\Rightarrow \text{Couleur}_{\text{Spéculaire}}(P) = K_s(V \cdot R)$$

- ▶ Le calcul du spéculaire donne une « tache » lumineuse sur l'objet (conséquence de la réflexion des rayons lumineux).
- ▶ Pour accentuer ou atténuer l'effet, on donne également un coefficient de brillance  $s$  pour accentuer ou atténuer l'effet autour de la direction principale.
- ▶  $\Rightarrow \text{Couleur}_{\text{Spéculaire}}(P) = K_s (V \cdot R)^s$



Effet spéculaire (seul) :  
 $K_s=(0.0,0.7,0.2)$ ;  $I_s=(1.0,1.0,1.0)$ ;  $s=100$

# Spécifier le spéculaire en OpenGL

Ce sont les même instructions que pour l'ambient et le diffus :

- ▶ `glMaterialfv (GL_FRONT_AND_BACK, GL_SPECULAR, <GLfloat Ks[3]>);`
- ▶ **Brillance** : `glMateriali (GL_FRONT_AND_BACK, GL_SHININESS, <int brillance>);`

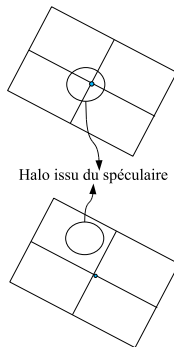
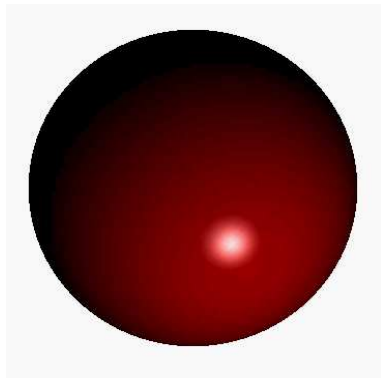
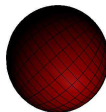
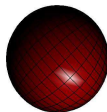
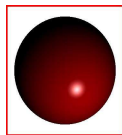
On peut récupérer ces données dans les shaders avec les built-in

`gl_FrontMaterial.specular` (de type `vec4`) et `gl_FrontMaterial.shininess` (de type `float`).



# Spéculaire calculé au sommet

- Spéculaire : très mal rendu s'il est calculé uniquement au sommet.



# Interpolation de Phong

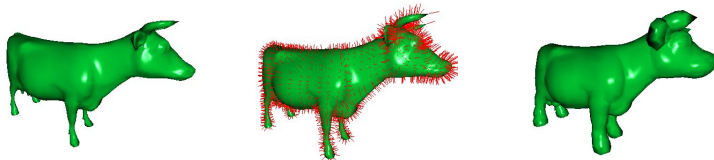
- ▶ On calcule l'éclairement spéculaire en chacun des pixels (surcote en temps de calcul).
- ▶ Avec quels vecteurs ? on prend les vecteurs  $L$ ,  $V$ ,  $N$  interpolés linéairement par rapport aux valeurs aux sommets (variables  $L$ ,  $V$ ,  $N$  définies comme *varying*).

### 3 Variables uniform

# Communiquer des valeurs aux shaders

- ▶ Communiquer des valeurs spécifiques à chaque sommet  $\Rightarrow$  attributs de sommets (`glVertexPointer`, `glColorPointer` par exemple)
- ▶ Communiquer des valeurs calculées dans le vertex et transmises par interpolation aux fragments  $\Rightarrow$  variables qualifiées de `varying`
- ▶ Communiquer des valeurs pour paramétrer les shaders (vertex ou fragment)  $\Rightarrow$  variables qualifiées de `uniform`
  - Une variable uniforme peut changer entre chaque tracé (`glDraw...`) par l'application OpenGL, mais leur valeur reste constante durant l'exécution des vertex/fragment shaders (lecture seule).

Exemple : modifier les coordonnées des sommets pour effectuer une dilatation.



$\Rightarrow$  déplacer le sommet dans la direction de la normale selon un certain facteur

# Vertex shader

```
#version 110

uniform float facteur;

varying vec4 couleurDiffus;
varying vec3 N,L,V;

void main() {
    float intensite;
    vec4 vertexLocal;
    vec4 vertexEye;

    // déplacement du sommet
    vertexLocal=facteur*vec4(gl_Normal,0)+gl_Vertex;

    // transformation & lighting
    vertexEye=gl_ModelViewMatrix*vertexLocal;

    V=vec3(-gl_ModelViewMatrix*gl_Vertex);
    L=gl_LightSource[0].position.xyz-vertexEye.xyz/vertexEye.w;
    N=gl_NormalMatrix*gl_Normal;

    L=normalize(L);
    V=normalize(V);
    N=normalize(N);

    intensite=dot(N,L);

    couleurDiffus=gl_FrontMaterial.diffuse*intensite;

    gl_Position=gl_ProjectionMatrix*vertexEye;
}
```

# Affecter les variables uniform

Dans l'application OpenGL on affecte les variables uniform des shaders avec `glUniform` :

```
GLuint programId;
GLuint locationFacteur;

void init() {
    programId=readMyShader("dilatation"); // fonction utilitaire pour lire/compiler/linker
                                           // les shaders des fichiers dilatation.vert et dilatation.frag par exemple

    locationFacteur=glGetUniformLocation(programId,"facteur"); // récupérer où se trouve "facteur" dans le shader
}

void draw() {
    glUseProgram(programId);
    glUniform1f(locationFacteur,0.1); // affecte la variable facteur du shader avec la valeur 0.1

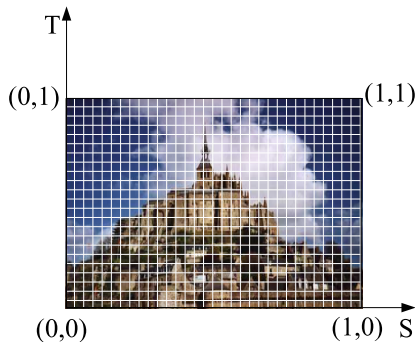
    obj.drawBuffer(); // commande de tracé

    glUseProgram(0);
}
```

# 4 Texture

# Définition (intuitive)

- ▶ Une texture est une « image », c'est à dire une grille de pixels.
- ▶ Les pixels de la texture sont appelés texels (pour les différencier des pixels de l'écran graphique).
- ▶ Chaque texel est localisé dans la texture par ses coordonnées  $s$  et  $t$ .
- ▶ Toute l'image de la texture est décrite par  $s \in [0, 1]$  et  $t \in [0, 1]$

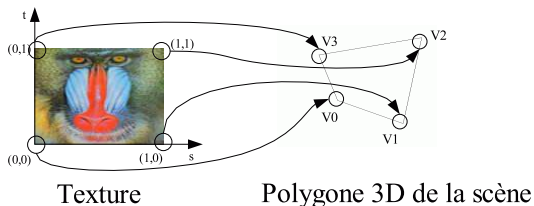


Remarque : la grille est grossie par rapport à la résolution réelle



# Plaquage une texture

- ▶ Consiste à associer à chaque point 3D des coordonnées de texture pour lui associer un texel de l'image texture
- ▶ Plaquage linéaire sur un triangle :
  - Il suffit d'associer des coordonnées de texture uniquement aux sommets.
  - Les coordonnées de texture des pixels sont alors interpolées linéairement lors du remplissage du triangle.



# Chargement d'une image de texture

- ▶ On doit attribuer une zone mémoire OpenGL pour accueillir l'image (mémoire qualifiée de texture buffer).

```
GLuint tex_id;
void initTexture() {
    // génération d'un identifiant
    glGenTextures(1,&tex_id);

    // la texture courante sera l'unité 0 à laquelle on affecte la texture tex_id
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D,tex_id); // toutes les instructions qui suivront s'adresseront à tex_id

    // affectation de l'image de la texture
    glTexImage2D(
        GL_TEXTURE_2D, // texture 2D (= image)
        0,             // niveau de mipmap (sera vu plus tard)
        3,             // le tableau est à interpréter par "paquet" de
                       // trois valeurs consécutives
        width,height,  // taille en pixels de l'image
        0,             // gestion des bords pour recollement
                       // (dans ce cours = 0)
        GL_RGB,        // interprétation par OpenGL (image sera
                       // stockée en valeurs de Rouge, Vert, Bleu)
        GL_UNSIGNED_BYTE, // format du tableau (ici unsigned byte)
        image);        // l'image (tableau contenant les texels).
```

# Attribution des coordonnées de texture

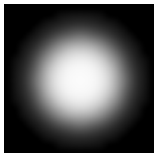
- ▶ On peut plaquer plusieurs textures simultanément lors d'un même tracé : gestion des unités de texture.
- ▶ Pour chaque unité de texture, on peut spécifier :
  - Une texture distincte (i.e. un identifiant distinct) `glBindTexture`
  - Des coordonnées de textures indépendantes et distinctes (i.e. plusieurs coordonnées de texture pour chaque sommet).



GL\_TEXTURE0



GL\_TEXTURE1



GL\_TEXTURE2



placage des 3 unités

# Attribution des coordonnées de texture

Affectation de la mémoire OpenGL avec des coordonnées de textures :

```
void initBuffer() {  
    float vertex[]={-1,-1,0,-1,1,0,1,-1,0,1,1,0};  
    float texture[]={0,0,0,1,1,0,1,1};  
  
    glGenBuffers(1,&bufferVertex);  
    glBindBuffer(GL_ARRAY_BUFFER,bufferVertex);  
    glBufferData(GL_ARRAY_BUFFER,12*sizeof(GLfloat),vertex,GL_STATIC_DRAW);  
  
    glGenBuffers(1,&bufferTexCoord0);  
    glBindBuffer(GL_ARRAY_BUFFER,bufferTexCoord0);  
    glBufferData(GL_ARRAY_BUFFER,8*sizeof(GLfloat),texture,GL_STATIC_DRAW);  
}
```

Pour que le tracé soit fait avec les coordonnées de textures comme attributs de sommet :

```
...           quel unité de texture  
glClientActiveTexture(GL_TEXTURE0); // on travaille avec le premier jeu de coordonnées de texture (i.e. GL_TEXTURE0)  
glEnableClientState(GL_TEXTURE_COORD_ARRAY); // activation de l'alimentation du premier jeu de coordonnées de texture  
                                                //(i.e. alimentation de gl_MultiTexCoord0 dans le vertex shader).  
glBindBuffer(GL_ARRAY_BUFFER,bufferTexCoord0);  
glTexCoordPointer(2,GL_FLOAT,0,0); // données du premier jeu de coordonnées de texture  
  
glDrawArrays(GL_TRIANGLE_STRIP,0,4);  
  
glClientActiveTexture(GL_TEXTURE0);  
glDisableClientState(GL_TEXTURE_COORD_ARRAY); // désactive le premier jeu de coordonnées de textures
```

A noter : possibilité d'avoir plusieurs jeux de coordonnées de textures gérées indépendamment (`glClientActiveTexture(GL_TEXTUREi)`).

# Type sampler2D

- Dans les shaders, les images de texture sont accessibles grâce à des uniform de type `sampler2D`

Vertex Shader (se contente d'interpoler les coordonnées de texture) :

```
varying vec2 coordTex;  
  
void main() {  
    coordTex=gl_MultiTexCoord0.st; // accès premier jeu de coordonnees  
  
    gl_Position = gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;  
}
```

Fragment shader (récupération des coordonnées de texture interpolées et lecture de la couleur dans l'image) :

```
uniform sampler2D uneTexture; stock l'unité de texture  
// affecter par l'application OpenGL  
varying vec2 coordTex;  
  
void main() {  
    vec4 couleur;  
    couleur=texture2D(uneTexture,coordTex);  
    gl_FragColor = couleur;  
}
```

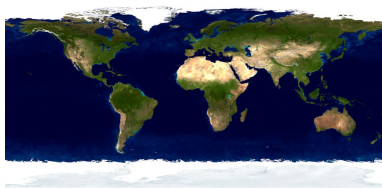
# Affectation de la texture depuis l'application

- On affecte l'uniform `uneTexture` avec l'unité de texture souhaitée.

```
glUseProgram(monShader.id());  
GLuint locationSampler=glGetUniformLocation(monShader.id(),"uneTexture"); // récupérer où se trouve "uneTexture" dans le shader  
glUniform1i(locationSampler,1); // uneTexture correspondra à l'image de l'unité de texture 1  
on affecte la variable  
draw();  
glUseProgram(0);
```

 dans le shader correspond a la texture d'unité 1

# Un exemple en multi-texture



# Dans l'application

Quelques classes utilitaires pour alléger le code :

```
class GLView : ... {  
...  
Shader _earthShader;  
Texture _earthDay, _earthNight;  
Sphere _sphere;  
...  
};
```

```
void GLView::init () {  
    earthDay.read("earthD.jpg");  
    earthNight.read("earthN.jpg");  
  
    glActiveTexture(GL_TEXTURE0);  
    earthDay.bind();  
    glActiveTexture(GL_TEXTURE1);  
    earthNight.bind();  
  
    sphere.initBuffer(); // génération sommets+coordonnées de textures  
}  
  
void draw () {  
    earthShader.enable();  
    earthShader.uniform("texJour", 0);  
    earthShader.uniform("texNuit", 1);  
  
    sphere.drawBuffer();  
  
    earthShader.disable();  
}
```



# Le vertex shader

## Vertex :

```
varying vec2 texCoord;  
varying vec3 normal;  
varying vec3 L;  
  
void main() {  
    texCoord=gl_MultiTexCoord0.st;  
  
    // Source supposée directionnelle ici :  
    L=gl_LightSource[0].position.xyz; // déjà exprimé dans le repère Eye  
  
    normal=gl_NormalMatrix*gl_Normal;  
  
    gl_Position = gl_ProjectionMatrix*gl_ModelViewMatrix*gl_Vertex;  
}
```

# Le fragment shader

## Fragment :

```
uniform sampler2D texJour, texNuit;

varying vec2 texCoord;
varying vec3 normal;
varying vec3 L;

void main() {
    vec4 couleurNuit, couleurJour;
    vec3 normal2, L2;
    normal2=normalize(normal);
    L2=normalize(L);
    float eclaire=dot(L2, normal2);

    // eclaire=-1 : completement nuit
    // eclaire=1 : completement jour
    eclaire=(eclaire+1.0)/2.0; // intervalle [0,1]

    couleurNuit=texture2D(texNuit, texCoord);
    couleurJour=texture2D(texJour, texCoord);

    // mélange texture jour+nuit (mélange trop étendu ici)
    gl_FragColor = couleurJour*(1-eclaire)+couleurNuit*eclaire;
}
```