OpenGL ES

La bibliothèque standard 2D/3D

Christian Nguyen

Département d'informatique
Université de Toulon

Un standard graphique :

- capacités graphiques avancées,
- modèle de programmation simple,
- spécifications adaptées aux architectures graphiques actuelles.

Portabilité : pas de pilote graphique propre.

Bibliothèque orientée uniquement vers la *manipulation* d'objets 3D et incorporable dans n'importe quel sous système fenêtré ou non.

Consortium (ATI, NVidia, Intel, Sun, SGI, ...) créé en 2000 pour définir des standards ouverts, gratuits pour la création de médias dynamiques.

- OpenGL (depuis 2006),
- OpenGL ES (OpenGL pour mobiles et systèmes embarqués),
- WebGL (binding javascript pour OpenGL ES),
- OpenVG (API vectorielle),
- EGL (abstraction entre API Khronos et système de fenêtrage),
- Collada (format de fichier 3D),
- ...

Historique

1985 GKS (*Graphical Kernel System*), 1ère bibliothèque graphique ISO,

1989 PHIGS (*Programmer's Hierarchical Interactive Graphics System*), basé sur GKS, ISO, liste d'affichage (*display list*). Inconvénients : modification très lourde, pas de rendu avancé,

1987 PEX (*PHIGS Extension to X*), rendu immédiat, différentes versions non compatibles, supprimé en 2004 (X11R6.7.0),

1992 OpenGL 1.0, créé par Silicon Graphics (1995 Direct3D),

2004 OpenGL 2.0,

2008 OpenGL 3.0 (spécifications),

2010 OpenGL 4.0 (DirectX 11),

2014 OpenGL 4.5.

2016 Vulkan, unifie OGL et OGLES (DirectX 12),

OpengL ES

Open Graphics Library for Embedded System.

Pour des plates-formes peu puissantes et avec peu de mémoire :

- nombreuses fonctionnalités d'OpenGL supprimées,
- calculs en virgule fixe,
- interface logicielle / matériel (uniformisation).

Versions :

1.0 dérivée d'OpenGL 1.3, spécification multiplate-forme,

1.1 OpenGL 1.5, adaptation à l'accélération matérielle sur plates-formes mobiles,

2.0 OpenGL 2.0, beaucoup plus léger, rendu pipeline programmable (shaders, GLSL).

3.0 OpenGL 3.3 et 4.2, août 2012.

(Wikipédia) Rappels sur la représentation des nombres :

- en virgule fixe : nombre fixe de chiffres après la virgule ; chaque $i$ème bit à droite de la virgule correspond à $1/2^i$, $\Rightarrow$ vitesse d'exécution, exactitude des calculs.
- en virgule flottante : réel $s.m.b^e$ avec $b = 2$ et m de taille fixe ; en faisant varier e, on fait "flotter" la virgule décimale. $\Rightarrow$ plus grand nombre de valeurs, moins de précision.

Types dans OpenGL ES :

- virgule fixe : suffixe `x`, sur 32 bits dont 16 bits fractionnaires,
- virgule flottante : pas de `double`, suffixe `f`,
- pas de `byte`, `ubyte`, `short` (sauf `glColor4ub`).

Fonctionnalités d'OpenGL ES 1.1

Bibliothèque graphique (*interface logicielle* d'une architecture graphique) : spécification et manipulation d'objets 2D/3D avec leurs attributs.

Toutes les opérations se font par l'intermédiaire de *commandes* (sous la forme d'appels de fonctions). Modèle d'interprétation des commandes : *client-serveur*.

Les primitives géométriques sont définies par l'intermédiaire de *sommets*. Un ensemble d'*attributs* est associé à chaque sommet : position, normale, couleur et texture. Chaque sommet est calculé *indépendamment*.

Propose des techniques de *rendu avancé*, en mode *immédiat* (plus de *display list*). Construction des *primitives* dans la mémoire image (*frame buffer*).

Fonctionnalités d'OpenGL ES 1.1

Fournit un contrôle direct sur les opérations 2D/3D fondamentales : *matrices* de transformation, coefficients des équations de la *lumière*, méthodes d'*antialiasing*, opérations de *mise à jour des pixels*, ...

Aucune fonction permettant de modéliser les *objets complexes* : description de l'apparence de l'objet et non de l'objet lui-même.

Différences fondamentales avec OpenGL 1.5 : plus de *display list*, plus d'approximation des courbes et des surfaces.

Portabilité

Les effets des commandes sont contrôlés en bout de chaîne par le système de gestion de fenêtres :

- détermine les *portions* de la mémoire image accessible à OpenGL,
- communique la *structure* de cette portion à OpenGL,
- génère l'image sur l'écran à partir de la mémoire image (incluant des transformations propres à la mémoire image telle que la correction gamma).

L'initialisation du système graphique est conditionnée par celle du *gestionnaire de fenêtre*. Les *entrées* utilisateurs sont aussi à la charge du gestionnaire de fenêtres.

EGL (eglut)

Bibliothèques développées afin d'apporter des fonctionnalités non-disponibles dans OpenGL.

EGL (Mesa) : interface entre une API Khronos (OpenGL ES, OpenVG, ...) et un environnement fenêtré ; prend en charge contexte graphique, binding, rendu.

eglut : bibliothèque très simple (glut) pour les démonstrations.

- eglutInitAPIMask(EGLUT_OPENGL_ES1_BIT), eglutInitWindowSize, eglutInit(int argc, char **argv),
- eglutCreateWindow, eglutDestroyWindow,
- eglutGet, eglutGetWindowWidth, eglutGetWindowHeight
- eglutDisplayFunc, eglutReshapeFunc, eglutKeyboardFunc, eglutSpecialFunc, eglutIdleFunc.
- eglutPostRedisplay
- eglutMainLoop

Politique de SGI pour OpenGL

SGI met à chaque fois dans le domaine public la version N-1 de GL, bibliothèque graphique de GL. Cette approche marketing

- décourage la concurrence (OpenGL étant gratuit et ouvert, pourquoi développer autre chose ?)
- dissuade la modification d'OpenGL (car tout ajout serait à recommencer dans la version d'OpenGL suivante)
- donne aux stations SGI un avantage concurrentiel substantiel, puisque GL a toujours une longueur d'avance sur OpenGL.

Implantations

Des implantations d'OpenGL (exploitant l'accélération 3D fournie par le matériel) existent pour les OS les plus courants.

Il existe une implantation libre de cette bibliothèque, nommée Mesa, créée en 1993 par Brian Paul et qui utilise la même API, ce qui permet :

- de se passer de la licence OpenGL dans la plupart des cas,
- de faire tourner des applications OpenGL sur des terminaux X.

A noter que Vulkan se base sur les travaux de Mesa 3D, en particulier le projet Gallium3D (pilotes graphiques modernes).

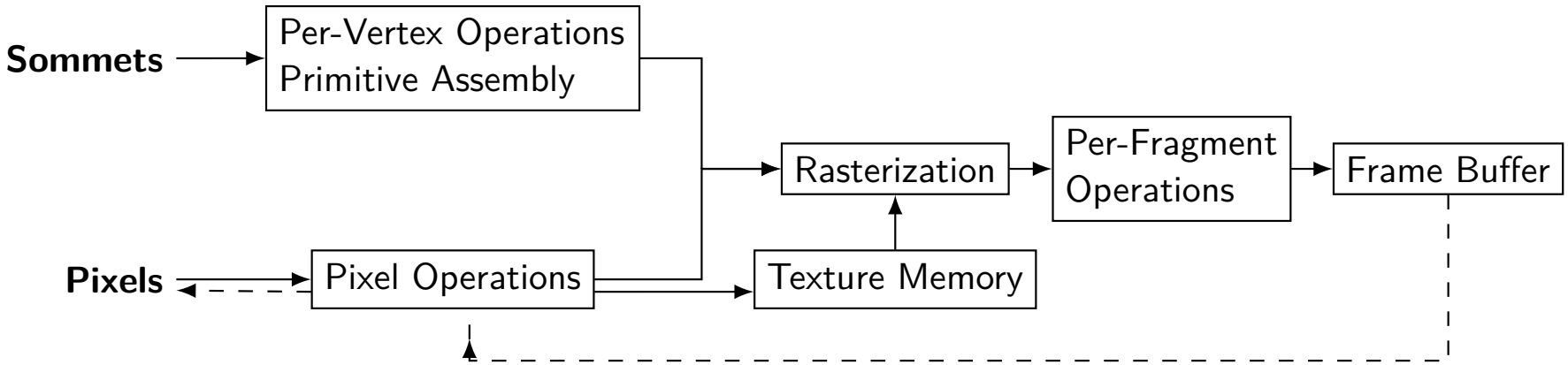
Utilisations

Liste non-exhaustive de programmes utilisant OpenGL :

- applications : Adobe After Effects, Adobe Photoshop, Adobe Premiere, 3D Studio Max, Autodesk Maya, Blender, Google Earth, Google SketchUp, Scilab, ...
- jeux vidéos : Angry Birds, Counter-Strike, Doom 3, Half-Life, Hitman, Medal of Honor, Minecraft, Neverwinter Nights, Portal, Quake, Unreal Tournament, Warcraft 3, World of Warcraft, ...

Certains logiciels utilisent OpenGL pour gérer l'ensemble de leur interface, même 2D, comme *Blender*, ou la version SGI de X11.

Le pipeline graphique d'OpenGL ES



Des opérations élémentaires

Le pipeline graphique d'OpenGL ES

1. *per-vertex operations* et *primitive assembly* : primitives géométriques (points, segments, polygones) décrites par des sommets (transformés et éclairés) fenêtrées au volume de visualisation.
2. *rasterization* : production d'une série de couples (adresse, valeurs) utilisant la description 2D des primitives pour donner des fragments.
3. *per-fragment operations* : opérations individuelles (mise à jour conditionnelle : *depth buffering*, *blending*, masques, ...) avant d'instancier la mémoire image.

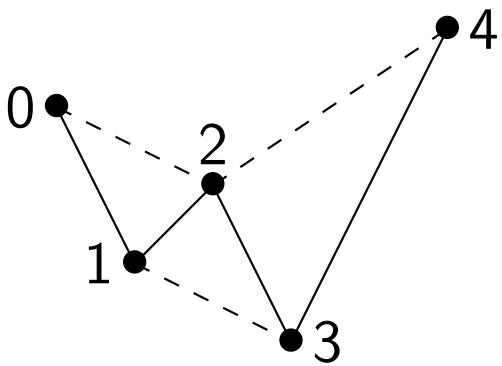
Les pixmaps évitent cette chaîne de processus et sont directement injectées dans le traitement des fragments au travers de l'étape de discrétisation (*rasterization*). Des valeurs peuvent aussi être lues dans la mémoire image ou copiées d'un bloc à l'autre. Ces transferts peuvent inclure des étapes de codage et de décodage.

Plan

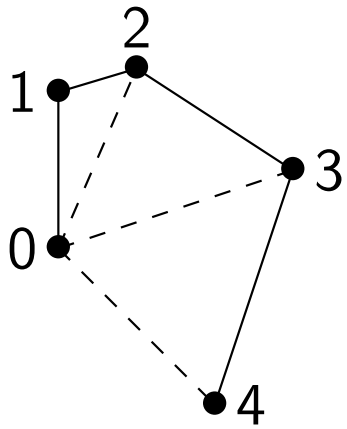
1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

Sommets et primitives

Objets	Sommets générés
point	chaque sommet donne la position d'un point
line strip	liste de segments jointifs
line loop	contour orienté fermé automatiquement
separate line	chaque paire de sommets décrit un segment
triangle strip	chaque sommet $s_{i>1}$ décrit un triangle avec s_{i-2} et s_{i-1}
triangle fan	chaque sommet $s_{i>1}$ décrit un triangle avec s_{i-1} et s_0
separate triangle	chaque triplet de sommets décrit un triangle



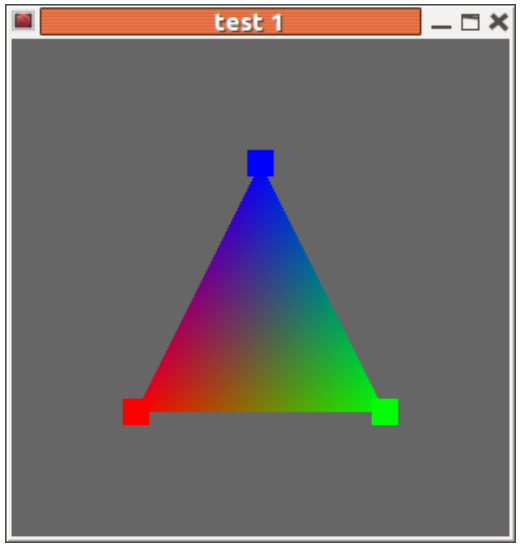
triangle strip



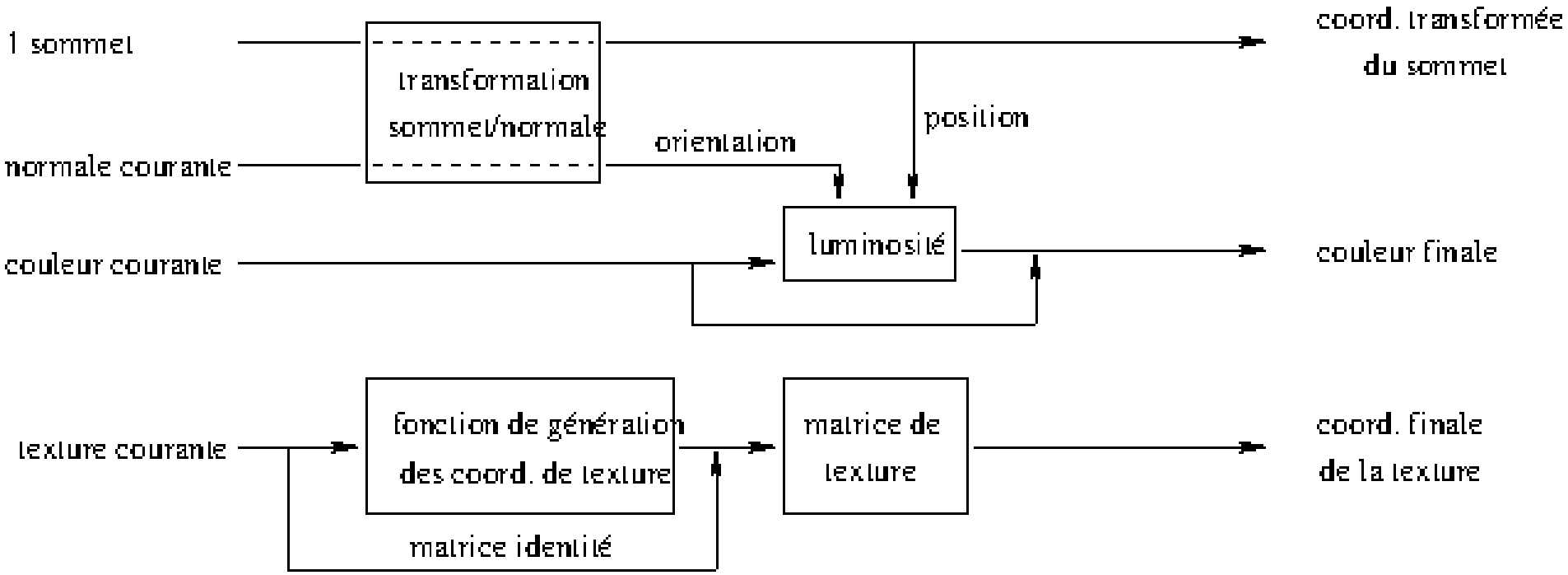
triangle fan

Un triangle

```
vertices = [-1.0,-1.0, 1.0,-1.0, 0.0,1.0]

glEnableClientState(GL_VERTEX_ARRAY)
   glVertexPointer(2, GL_FLOAT, 0, vertices)
   glDrawArrays(GL_TRIANGLES, 0, 3)
glDisableClientState(GL_VERTEX_ARRAY)
```



Sommets et primitives



Association des valeurs courantes et d'un sommet

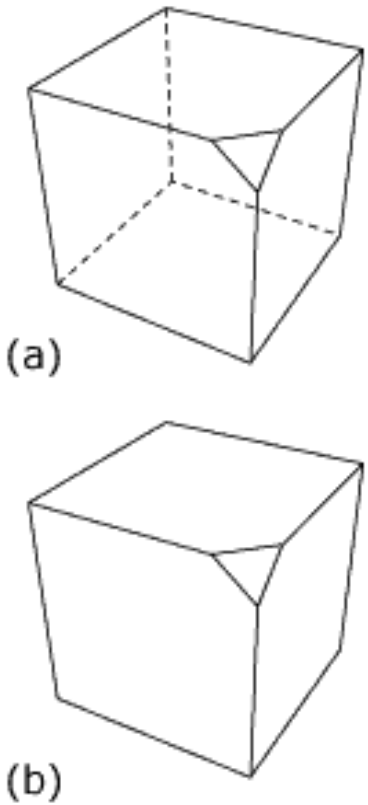
Les triangles

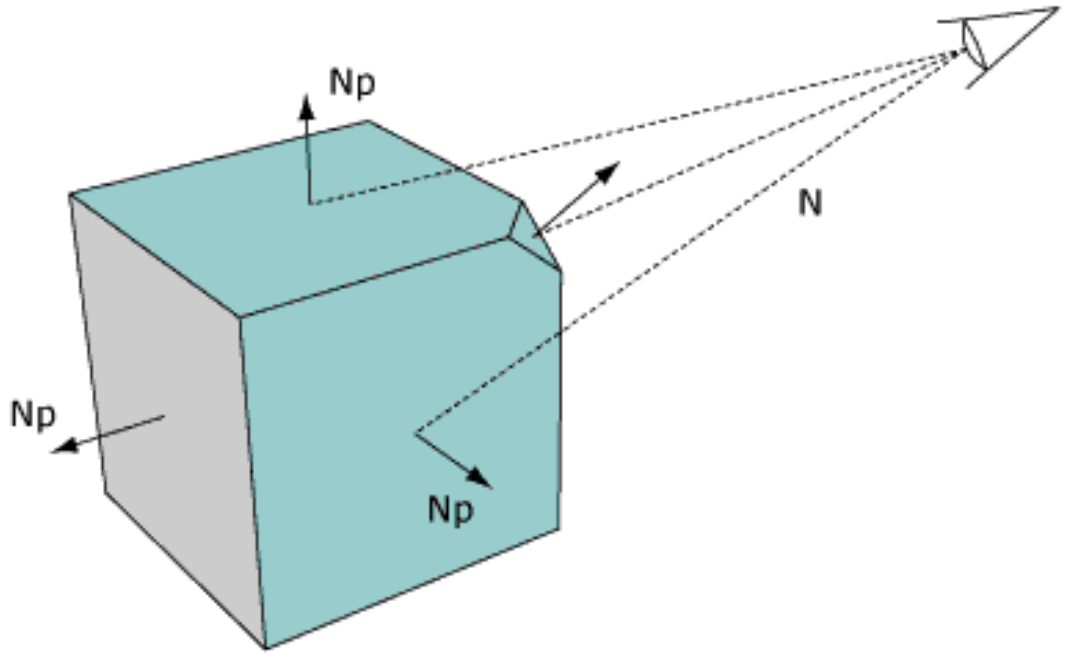
Définition par utilisation exclusive de tableaux (de couleurs, de sommets, de normales et de textures) pour des raisons d'efficacité. Exemple : `glEnableClientState(GL_VERTEX_ARRAY)`.

Chaque normale doit être orientée vers l'extérieur de la surface (triangulaire) et doit être normalisée (voir `glEnable(GL_NORMALIZE)`).

Orientation des faces (*counter clockwise* ou CCW) : *glFrontFace(mode)* avec mode = (GL_CCW, GL_CW).

Elimination des faces arrières par *glCullFace(mode)* avec mode = (GL_FRONT, GL_BACK, GL_FRONT_AND_BACK).


(a)
(b)


Np
N
Np
Np

Plan

1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

La couleur

Définie par un quadruplet (R, V, B, A) avec chaque composante dans $[0.0, 1.0]$:

- soit associée à chaque sommet (tableau et `glEnableClientState(GL_COLOR_ARRAY)`),
- soit contextuel (exemple : `glColor4f(1.0, 0.0, 0.0, 1.0)`).

RVBA : texture, lumière, ombrage, flou, antialiasing, mélange (*blending*), ...

Nombre de bits alloué à chaque composante couleur : `glGetIntegerv(compcoul)` avec compcoul = (`GL_RED_BITS`, ..., `GL_ALPHA_BITS`).

Tramage (*dithering*) : `glEnable(GL_DITHER)` (cf. double-buffer).

Affichage

OpenGL ES travaille en mode immédiat.

L'initialisation se fait par *glClear** :

```
glClearColor(0, 0, 0, 0)  # couleur de fond
glClearDepth(0)  # profondeur par defaut
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
```

ATTENTION

Réinitialiser le Z-buffer entre chaque changement de vue par : `glClear(GL_DEPTH_BUFFER_BIT)`

Forcer la fin d'un dessin : `glFlush()` (modèle client-serveur).

Exemple de scène

```
#include <GLES/gl.h>  /* use OpenGL ES 1.x */
#include <GLES/glext.h>
#include <EGL/egl.h>
#include <eglut.h>

static void draw(void);
static void reshape(int width, int height);
static void init(void);
static void key(unsigned char key);
static void special_key(int special);

int main(int argc, char *argv[])
{
   eglutInitWindowSize(300, 300);
   eglutInitAPIMask(EGLUT_OPENGL_ES1_BIT);
   eglutInit(argc, argv);

   win = eglutCreateWindow("solides basiques");

   eglutReshapeFunc(reshape);
   eglutDisplayFunc(draw);
   eglutKeyboardFunc(key);
   eglutSpecialFunc(special_key);

   init();

   eglutMainLoop();

   return EXIT_SUCCESS;
}
```

Exemple de scène

draw()

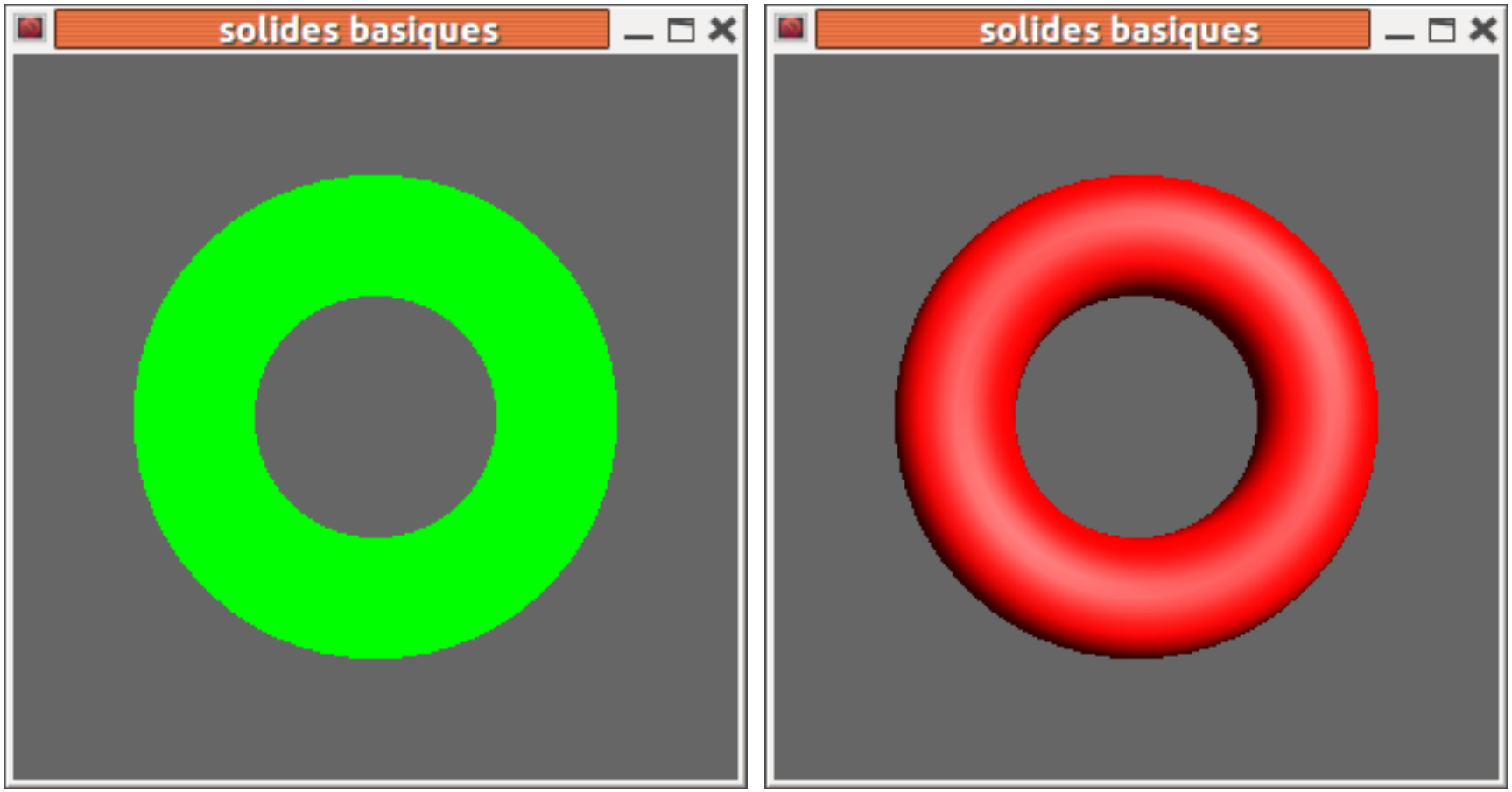
```
static GLfloat view_rotx = 0.0, view_roty = 0.0, view_rotz = 0.0;

static void draw(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix();
   glRotatef(view_rotx, 1, 0, 0);
   glRotatef(view_roty, 0, 1, 0);
   glRotatef(view_rotz, 0, 0, 1);
   glScalef(0.5, 0.5, 0.5);

   glColor4f(0,1,0,1);

   draw_torus(1.0, 3.0, 30, 60);

   glPopMatrix();
}
```

Exemple de scène

init(), key(), special_key()

```
static void init(void)
{
   glClearColor(0.4, 0.4, 0.4, 0.0);
   glEnable(GL_DEPTH_TEST);
}

static void key(unsigned char key)
{
  switch (key) {
  case 't':
    solide_courant = TORE;
    break;
  ...
  }
}

static void special_key(int special)
{
   switch (special) {
   case EGLUT_KEY_LEFT:
      view_roty += 5.0;
      break;
   ...
   }
}
```

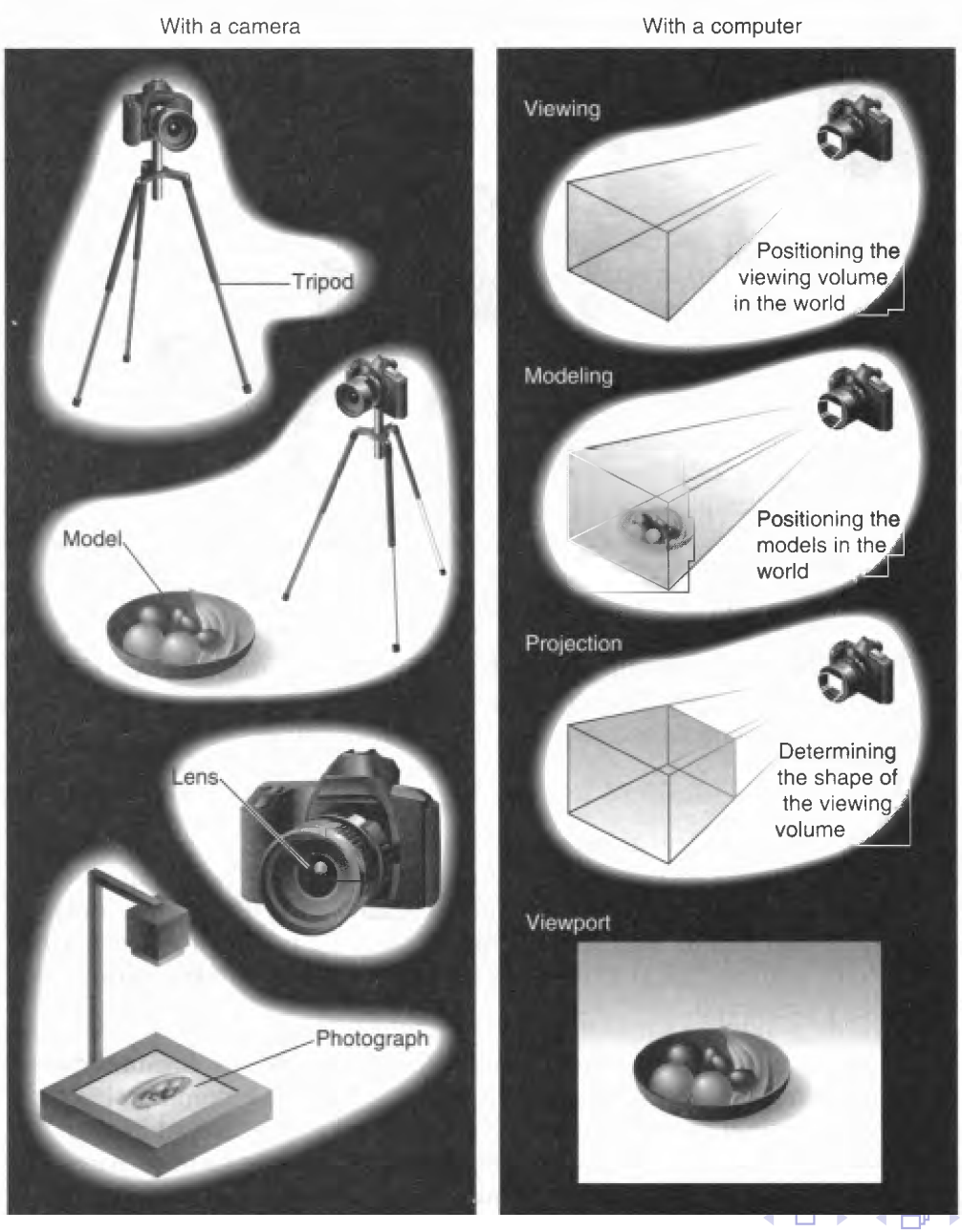
Copie d'écran



Plan

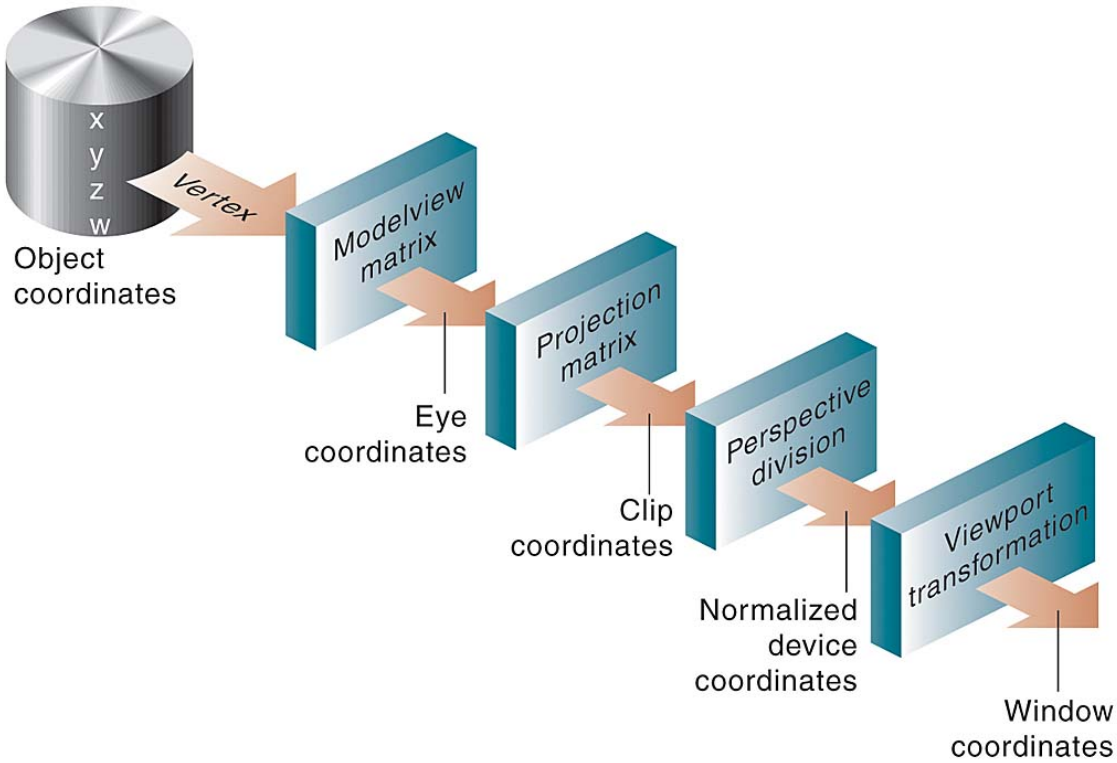
1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

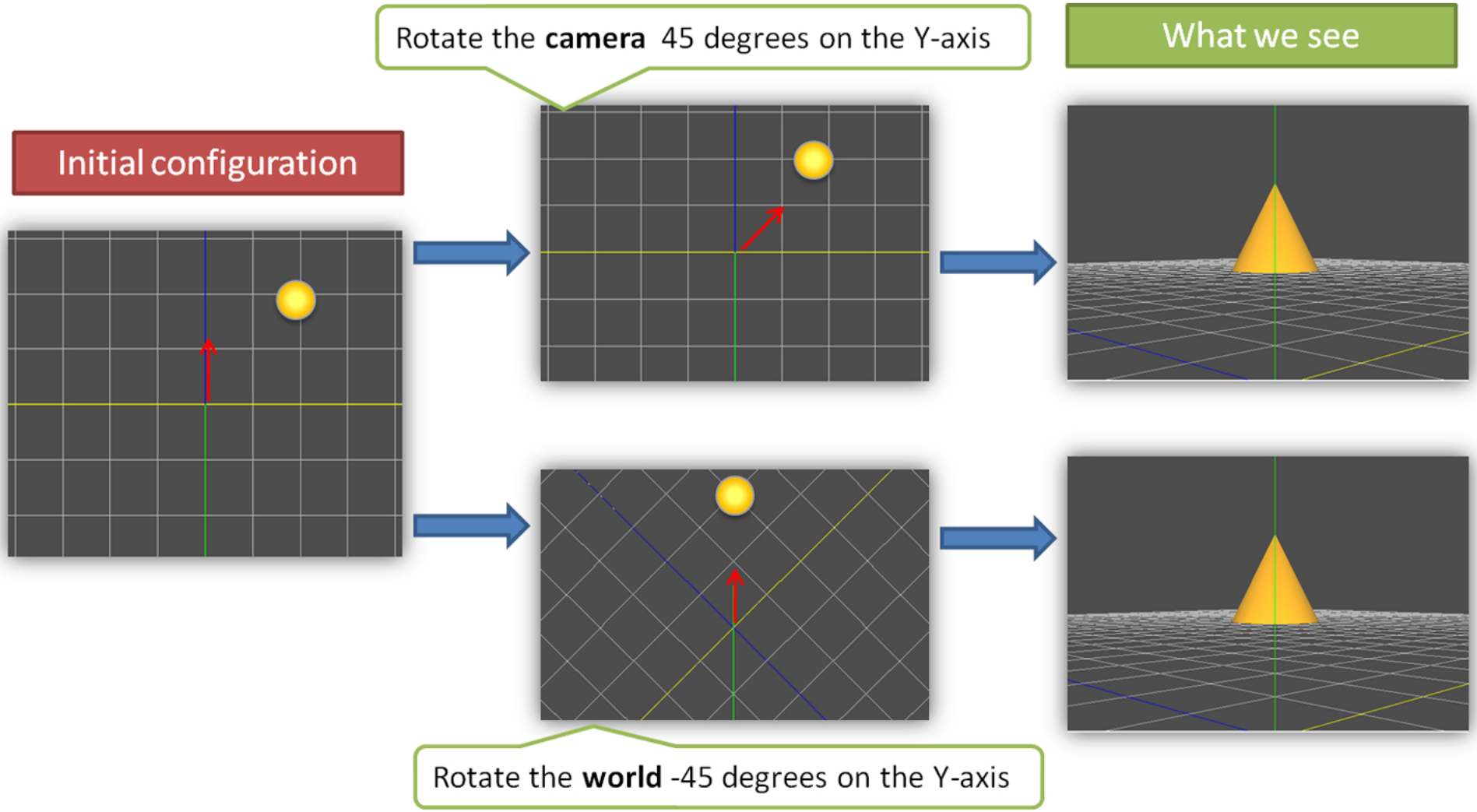
Visualisation



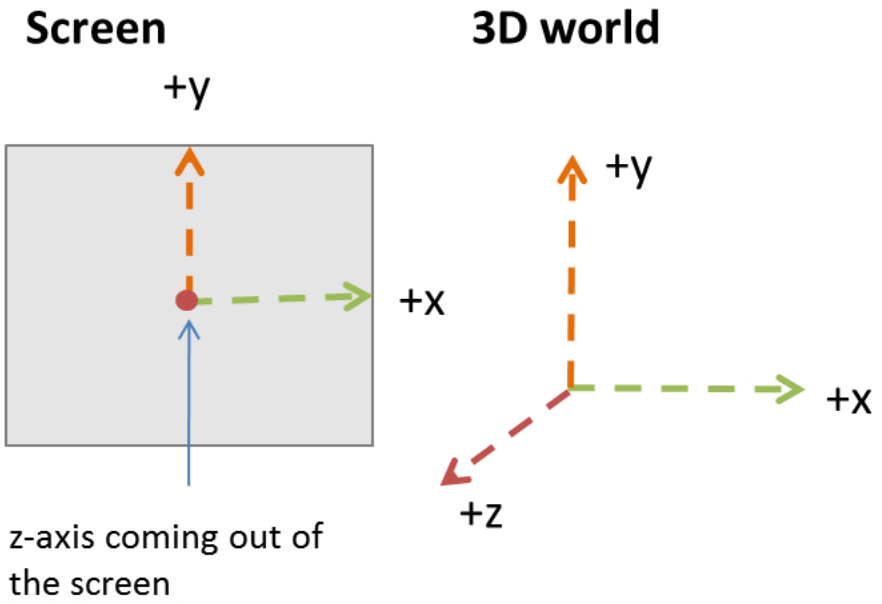
Visualisation

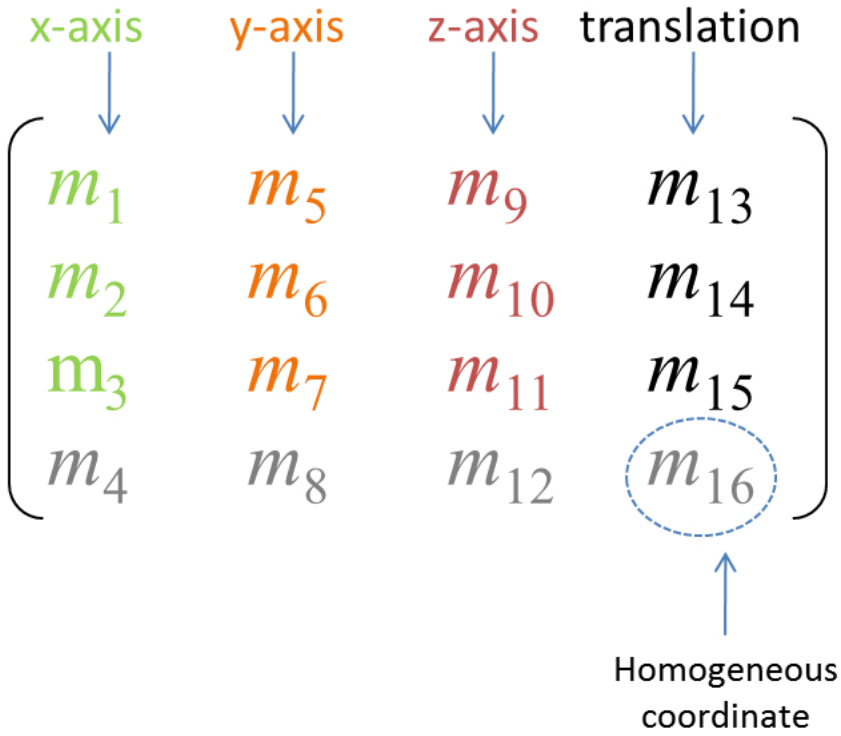
1. positionner, orienter la caméra et agencer la scène : une seule transformation (`glMatrixMode(GL_MODELVIEW);`),
2. choix de l'objectif (`glMatrixMode(GL_PROJECTION);`),
3. développement, choix de la taille de l'image papier (*viewport transformation*).

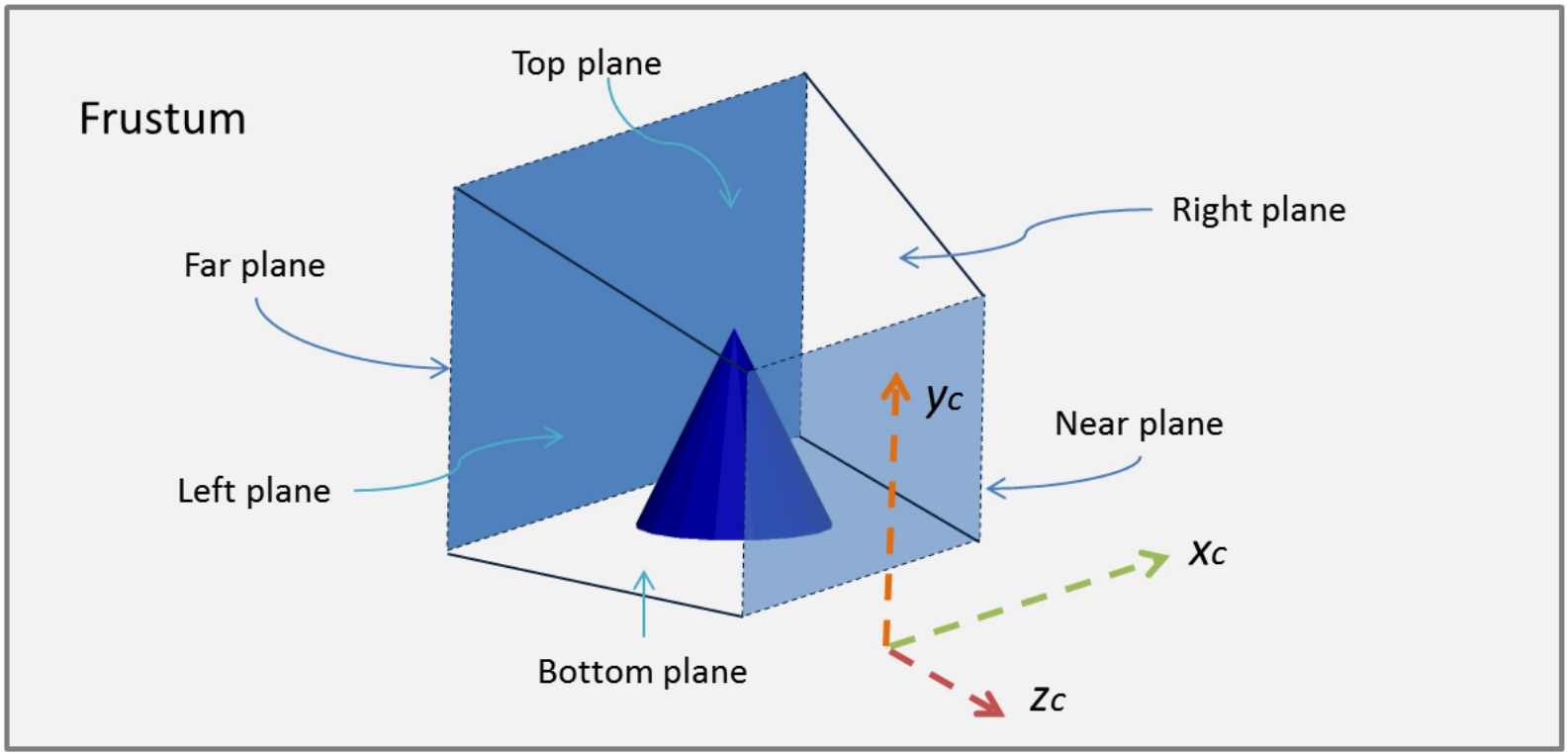



Rotate the **camera** 45 degrees on the Y-axis
What we see
Initial configuration
Rotate the **world** -45 degrees on the Y-axis

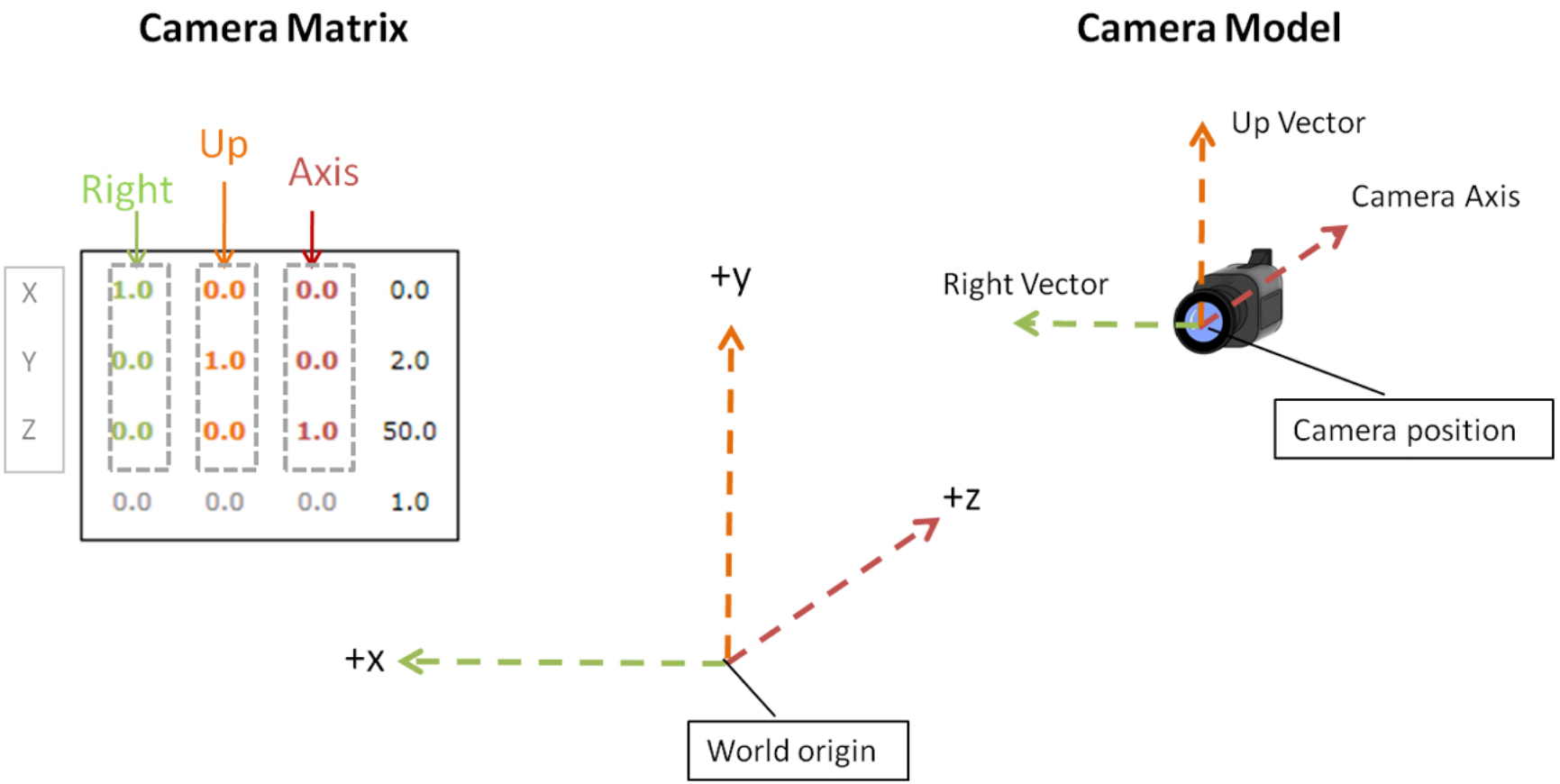
The model-view matrix

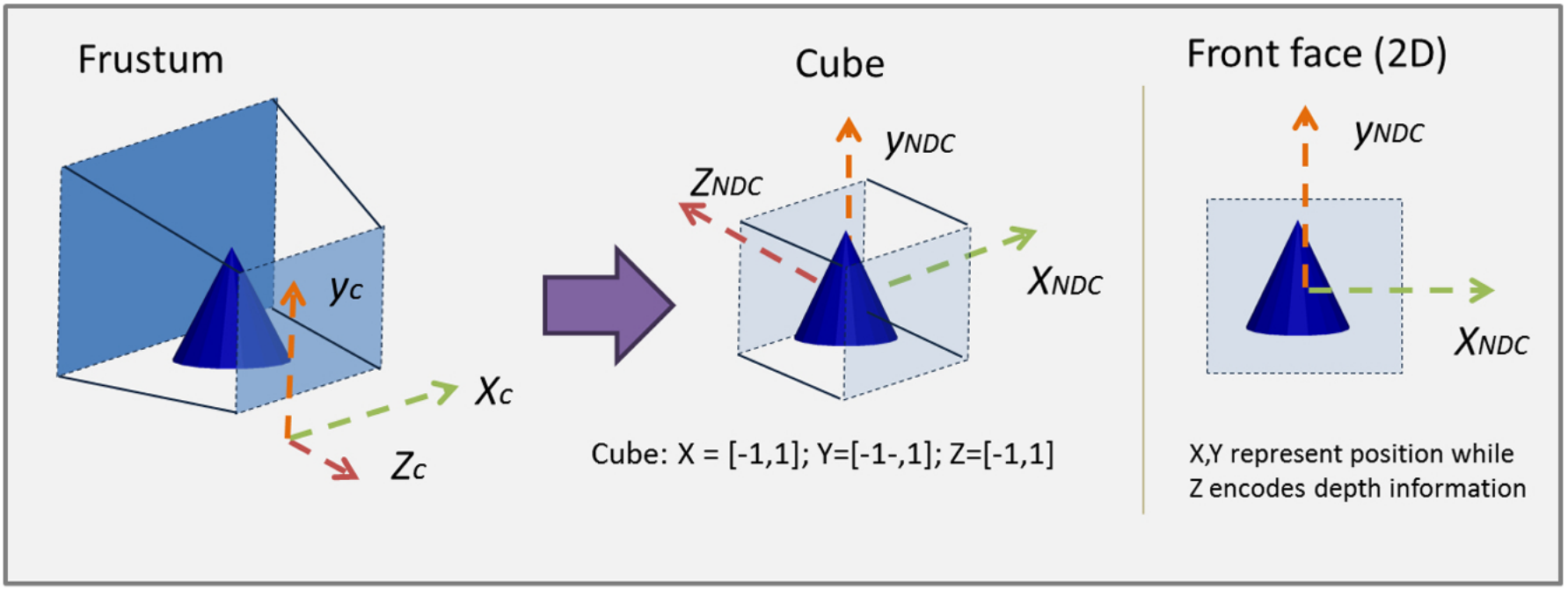




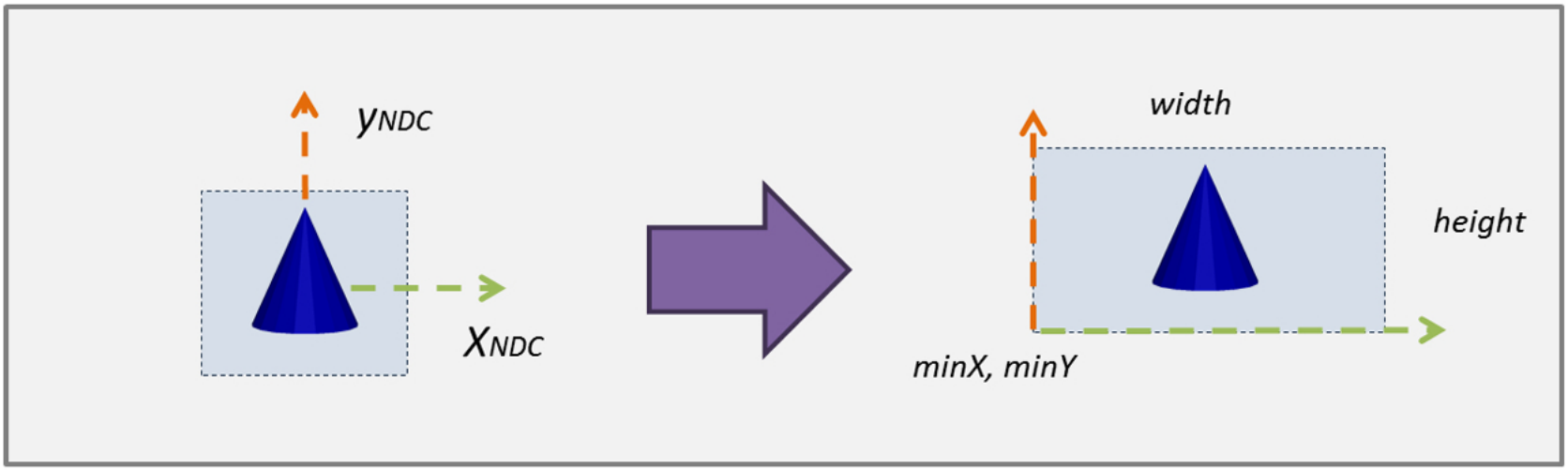

Frustum
Top plane
Right plane
Far plane
Left plane
Near plane
Bottom plane
yc
Xc
Zc


View Coordinates
Projection Transform
Clip Coordinates


Camera Matrix
Right
Up
Axis
X
Y
Z
1.0
0.0
0.0
0.0
0.0
1.0
0.0
2.0
0.0
0.0
1.0
50.0
0.0
0.0
0.0
1.0
Camera Model
Up Vector
Camera Axis
Right Vector
Camera position
+y
+z
+x
World origin


Frustum
Cube
Front face (2D)
yc
Xc
Zc
yNDC
ZNDC
XNDC
yNDC
XNDC
Cube: X = [-1,1]; Y=[-1-,1]; Z=[-1,1]
X,Y represent position while
Z encodes depth information

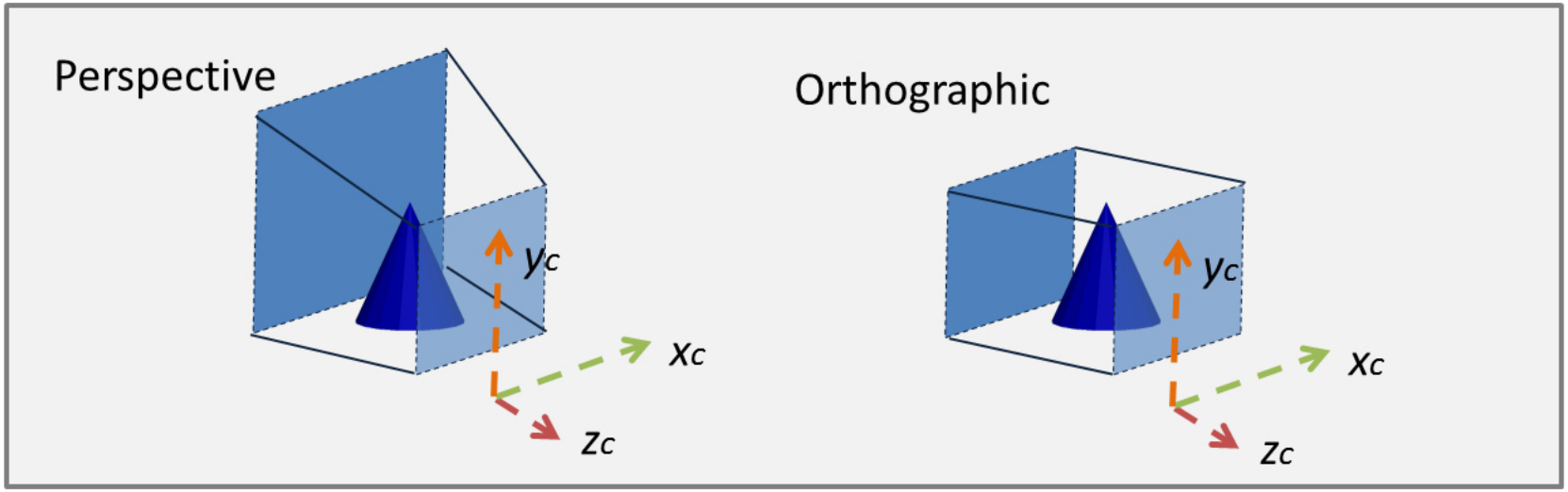

Clip Coordinates
Perspective Division
Normalized Device Coordinates


yNDC
XNDC
width
height
minX, minY


NDC
Viewport Transform
Viewport (Canvas) Coordinates

Fonctions prédéfinies (ou pas)

Définition du volume perspective : *glOrtho()* ou *glFrustum()*.



Vision perspective ou orthographique

Plus intuitif :

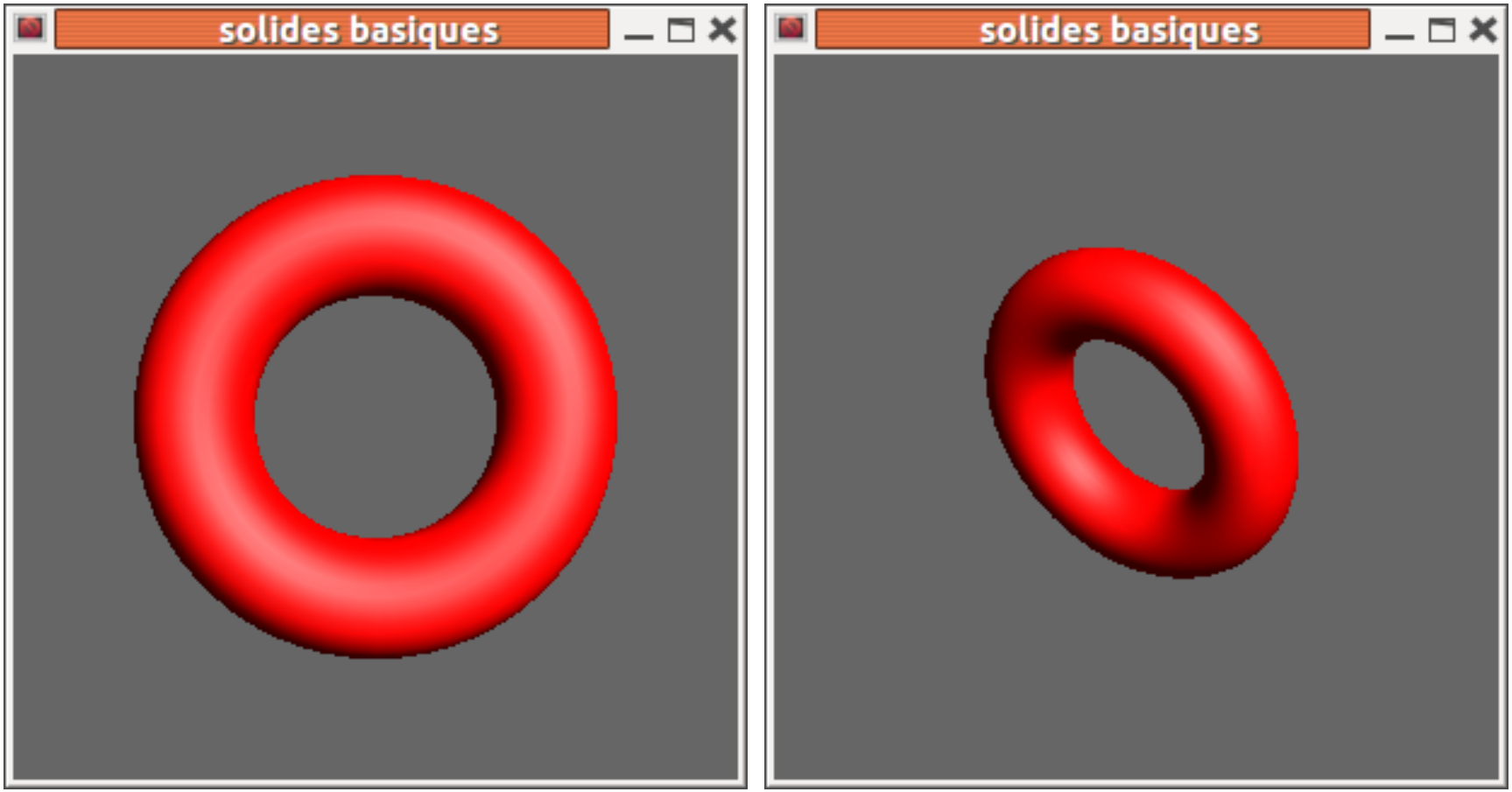
- *gluLookAt(point, refpoint, upvector)* : positionne intuitivement la caméra,
- *gluPerspective(T, aspect, near, far)* avec T : profondeur de champ.

Exemple de scène (2)

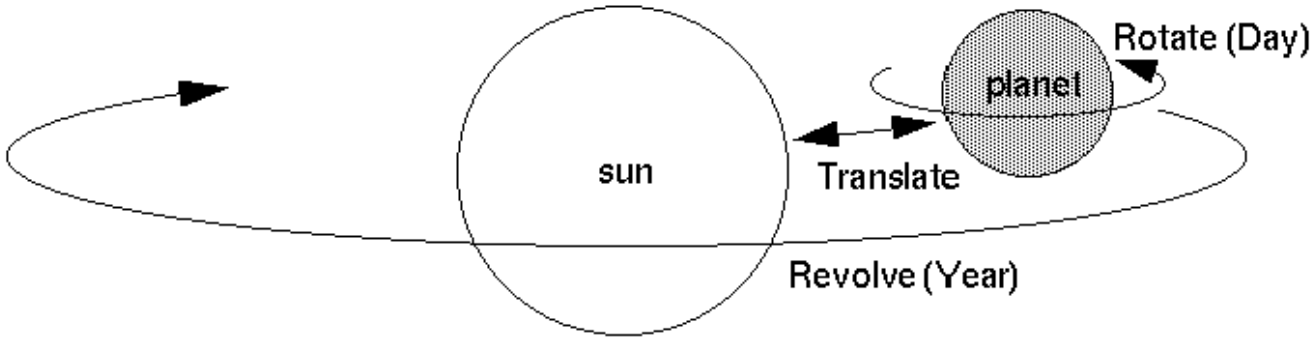
reshape()

```
static void reshape(int width, int height)
{
   GLfloat ar = (GLfloat) width / (GLfloat) height;

   glViewport(0, 0, (GLint) width, (GLint) height);

   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();

#ifdef GL_VERSION_ES_CM_1_0
   glFrustumf(-ar, ar, -1, 1, 5.0, 60.0);
#else
   glFrustum(-ar, ar, -1, 1, 5.0, 60.0);
#endif

   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glTranslatef(0.0, 0.0, -15.0);
}
```

Copie d'écran (2)



Composition de transformations

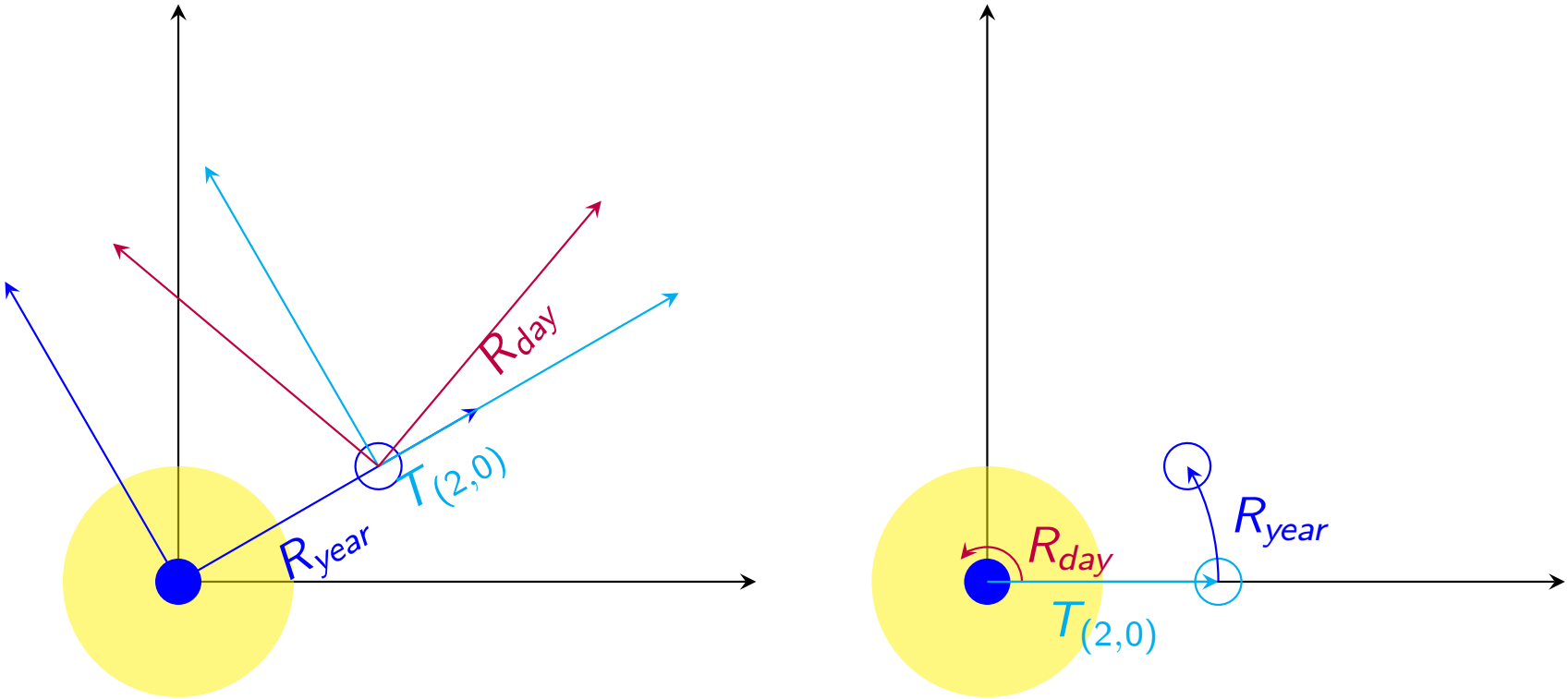


Exemple de mouvement Terre / Soleil

```
void display(void)
{
   ...
   Sphere(1.0, 20, 16); /* draw sun */
   glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
   glTranslatef (2.0, 0.0, 0.0);
   glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
   Sphere(0.2, 10, 8);  /* draw planet */
   ...
}
```

Composition de transformations

Lors de la phase de conception, il peut être avantageux de travailler par rapport à un repère fixe (figure de droite), ce que l'on fera en appliquant les transformations de bas en haut.



Les piles de matrices

Chaque type de matrice possède sa propre pile de matrices (*projection* : 2 ; *modelview* : 32).

Construction de modèles hiérarchiques avec transformations relatives (par la pile) ou indépendantes (*glLoadIdentity*).

Exemple, une voiture, 4 roues et 5 boulons par roue :

1. dessin de la voiture
2. mémorisation de son origine (`push`)
    1. translation relative appropriée pour placer la 1ère roue
    2. mémorisation de la position courante (`push`)
        1. 5 translations puis rotations pour les 5 boulons
3. retour à l'origine de la voiture (`pop`)
    1. translation relative appropriée pour placer la 2ème roue

...

Les piles de matrices

Le mécanisme de pile permet la mémorisation de la position courante (*glPushMatrix*) et le retour à la position précédente (*glPopMatrix*). Exemple :

```
void Roue_Boulons()
{
  Roue();
  for (int i = 0 ; i < 5 ; i++) {
    glPushMatrix();
      glRotatef(degre*i, 0, 0, 1); // autour de Z
      glTranslatef(Xb, 0, 0);
      Boulon(); // ATTENTION : R*T*Boulon()
    glPopMatrix();
  }
}

void Quatre_Roues()
{
  glPushMatrix();
    glTranslatef(Xr1, 0, Zr1);
    Roue_Boulons();
  glPopMatrix();
  glPushMatrix();
    glTranslatef(Xr1, 0, Zr2);
    Roue_Boulons();
  glPopMatrix();
  ...
}
```

Plan

1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

Ombre et lumière

Shading : *glShadeModel(mode)* avec mode = (GL_FLAT, GL_SMOOTH).

La lumière : codage RVB pour les lumières et la matière :

- lumière : correspondent au pourcentage d'intensité de chaque couleur,
- matière : proportion réfléchie de chaque composante couleur.

Soient (l_r, l_v, l_b) les composantes de la lumière émise et (m_r, m_v, m_b) celles de la matière, la lumière parvenant à l'oeil est donnée par $(l_r * m_r, l_v * m_v, l_b * m_b)$.

Rendu d'un tore

Nécessite les étapes suivantes :

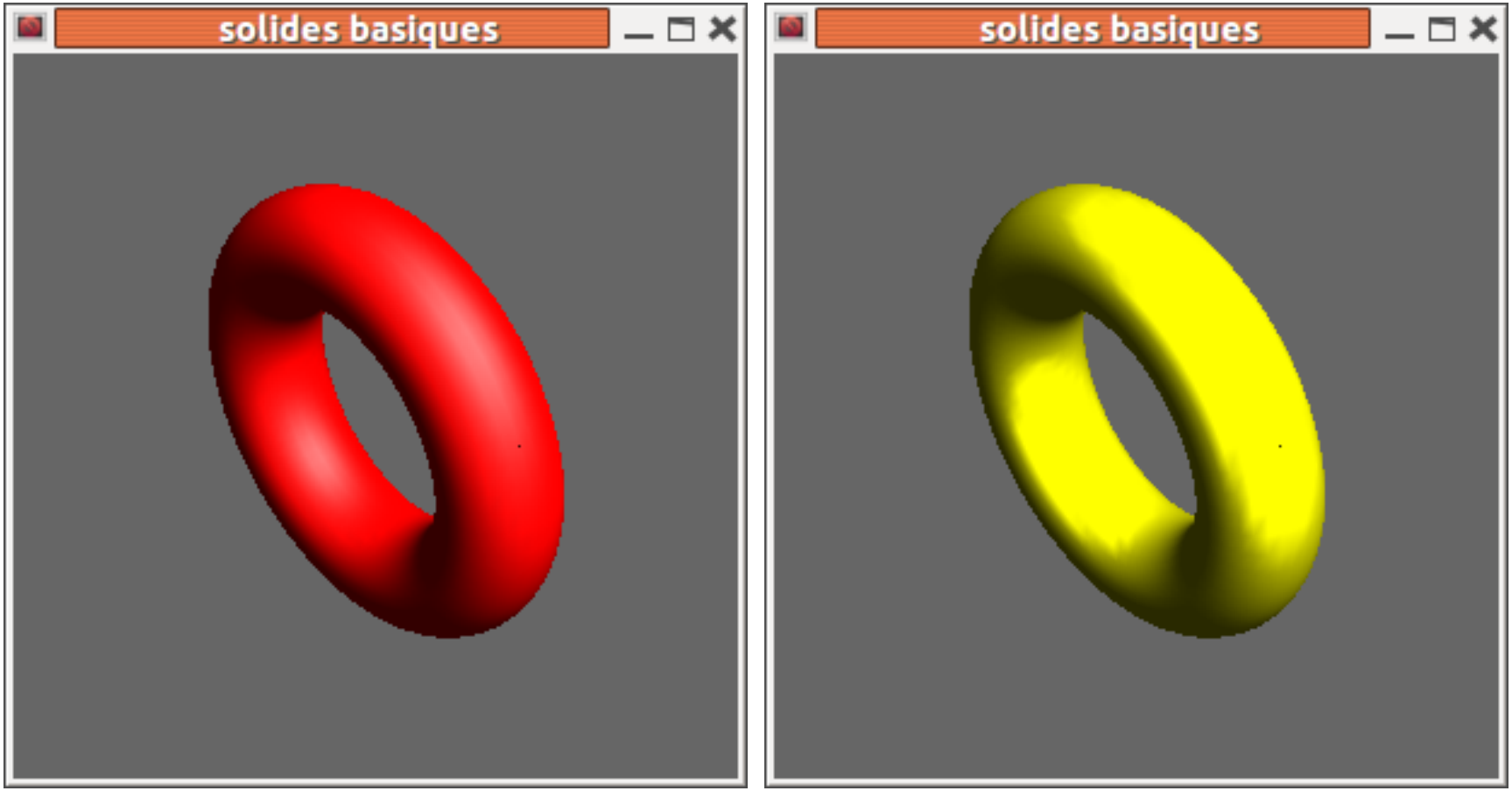
1. définition des vecteurs normaux de chaque sommet des objets,
2. création, sélection et position d'une ou de plusieurs sources lumineuses,
3. création et sélection d'un modèle de lumière, qui définit le niveau de la contribution ambiante et la position effective du point de vue (local ou à l'infini),
4. définition des propriétés de la matière pour les objets de la scène.

Exemple de scène (3)

light()

```
static void light(void)
{
   static const GLfloat red[4] = {1, 0, 0, 0};
   static const GLfloat white[4] = {1.0, 1.0, 1.0, 1.0};
   static const GLfloat diffuse[4] = {0.7, 0.7, 0.7, 1.0};
   static const GLfloat specular[4] = {0.001, 0.001, 0.001, 1.0};
   static const GLfloat pos[4] = {20, 20, 50, 1};

   glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, red);
   glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, white);
   glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 9.0);

   glEnable(GL_LIGHTING);
   glEnable(GL_LIGHT0);
   glLightfv(GL_LIGHT0, GL_POSITION, pos);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
   glLightfv(GL_LIGHT0, GL_SPECULAR, specular);
}
```

Copie d'écran



à droite : GL_AMBIENT_AND_DIFFUSE, yellow ; GL_SPECULAR, yellow ; GL_SHININESS, 20.0

Création des sources lumineuses

Paramètre	Valeur par défaut
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)
GL_DIFFUSE	(0.0, 0.0, 0.0, 1.0)
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)
GL_POSITION	(0.0, 0.0, 1.0, 0.0)
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)
GL_SPOT_EXPONENT	0.0
GL_SPOT_CUTOFF	180.0
GL_CONSTANT_ATTENUATION	1.0
GL_LINEAR_ATTENUATION	0.0
GL_QUADRATIC_ATTENUATION	0.0

Par défaut : source lumineuse directionnelle (+Z) sans attenuation.

Création des sources lumineuses

Activation : $glEnable(GL\_LIGHT_i)$ avec $i = 0..7$.

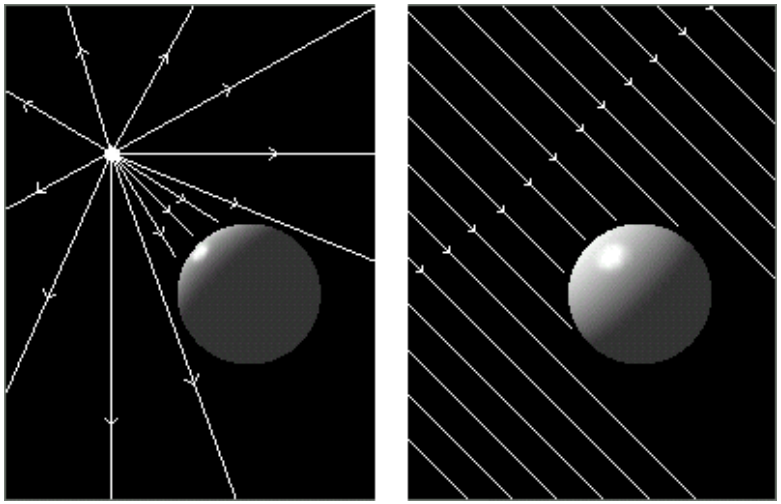
Prise en compte des calculs associés aux sources lumineuses : `glEnable(GL_LIGHTING);`

La lumière `GL_LIGHT0` est définie pas défaut par les paramètres suivants :

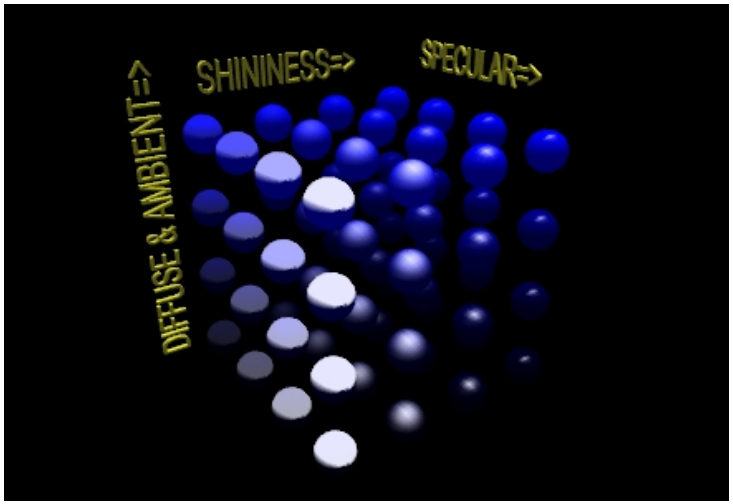
- GL_AMBIENT(0,0,0,1),
- GL_DIFFUSE(1,1,1,1),
- GL_SPECULAR(1,1,1,1),
- GL_POSITION(0,0,1,0).

Création des sources lumineuses

Lumière directionnelle ou positionnelle



Paramètres `ambient`, `diffuse` et `specular`



Mouvement d'une lumière

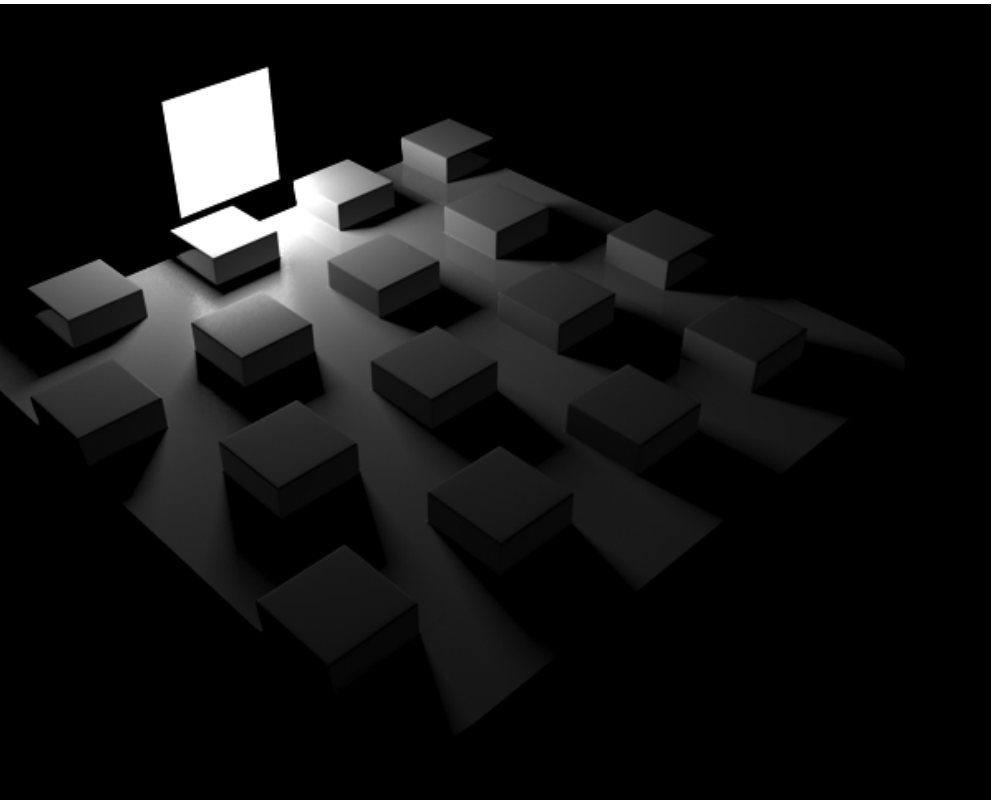
Exemple de lumière comme objet

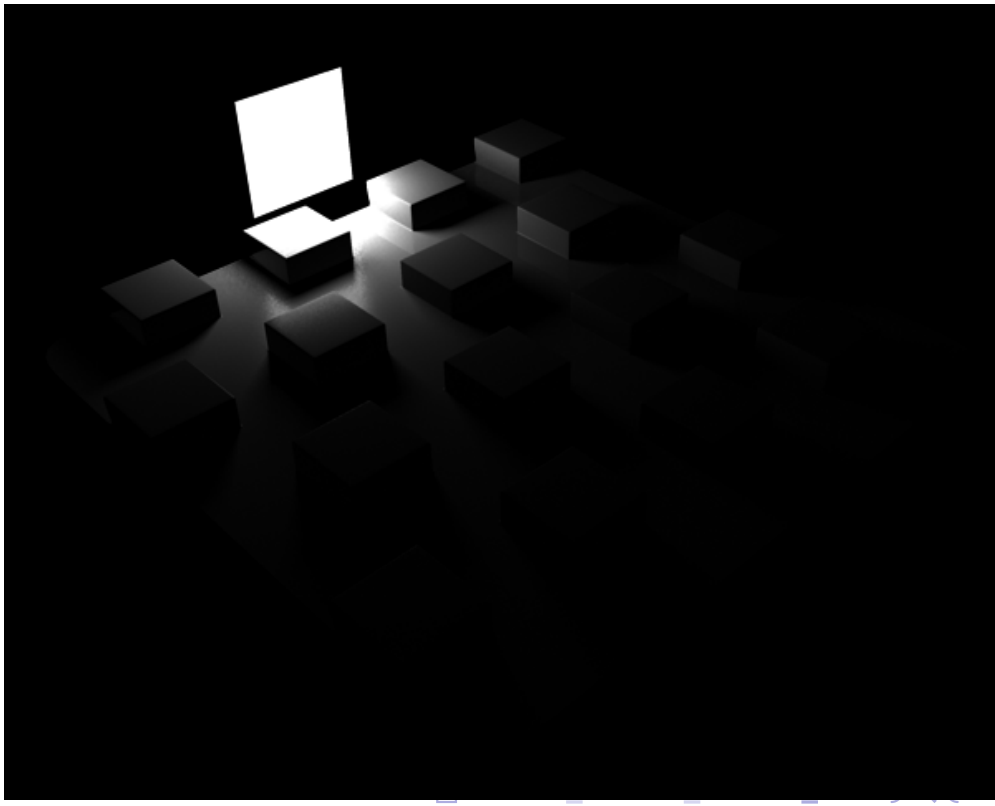
```
void display()
{
  GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
  glClear(GL_COLOR_BUFFER_BIT) | GL_DEPTH_BUFFER_BIT);

  glPushMatrix();
    glTranslatef(0.0, 0.0, -5.0);
    glPushMatrix();
      glRotated((GLdouble) angle, 1.0, 0.0, 0.0);
      glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glPopMatrix();
    draw_torus(1.0, 3.0, 30, 60);
  glPopMatrix();
  glFlush();
}
```

Les sources lumineuses (suite)

Paramètres :

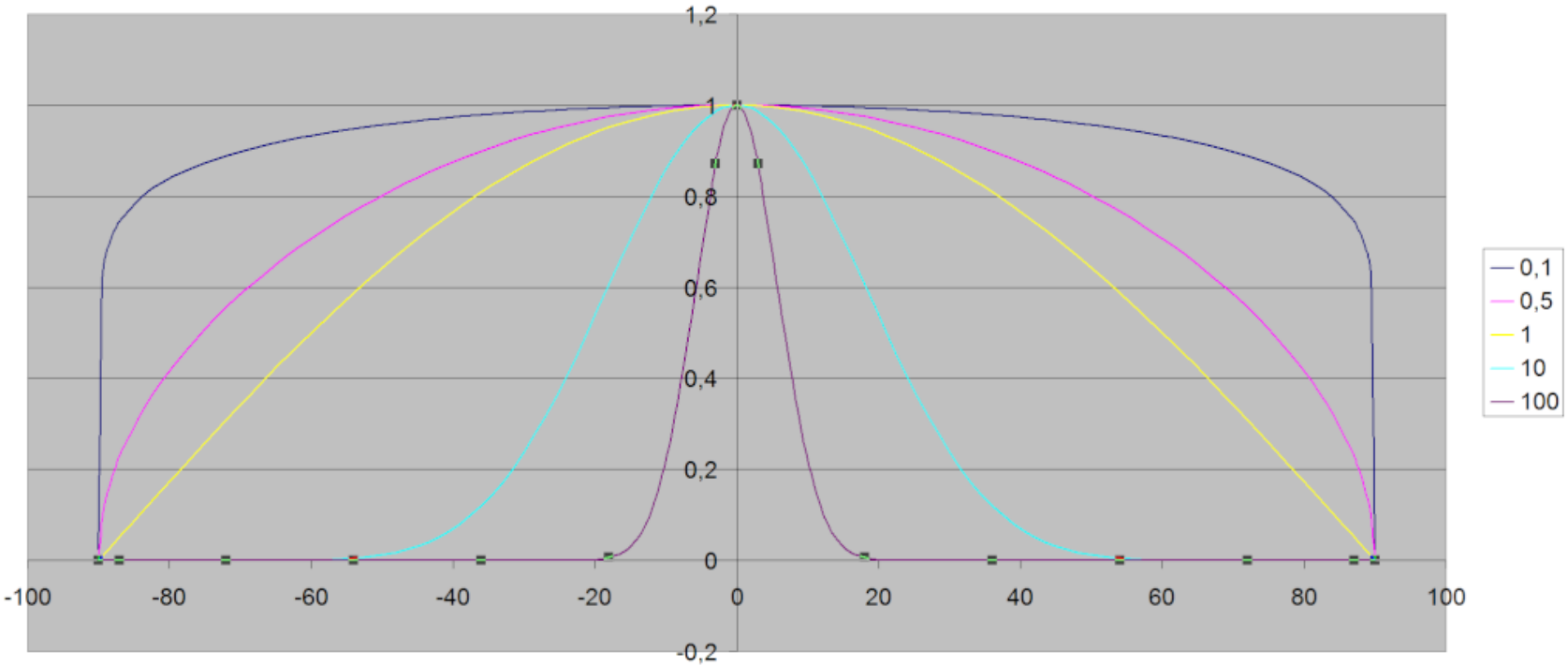
- de couleurs : `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`,
- atténuation (avec position) : $1/(k_c + k_l.d + k_q.d^2)$
  `GL_CONSTANT_ATTENUATION` (k_c),
  `GL_LINEAR_ATTENUATION` (k_l),
  `GL_QUADRATIC_ATTENUATION` (k_q).





Les spots

- `GL_SPOT_DIRECTION`,
- `GL_SPOT_CUTOFF` (angle d'ouverture) : de 0 à 90°, entre l'axe et une génératrice du cône.
- `GL_SPOT_EXPONENT` (focalisation) : l'intensité lumineuse diminue du centre à la périphérie comme le cosinus de l'angle entre l'axe et l'objet éclairé à la puissance `GL_SPOT_EXPONENT`,



Modèle d'illumination

Contribution ambiante pour toute la scène (valeurs initiales : 0.2, 0.2, 0.2, 1.0) :
glLightModel(GL_LIGHT_MODEL_AMBIENT, ...)*.

La couleur d'un sommet est la somme :

- de la couleur d'émission (cf *material*),
- du produit de la réflection ambiante (cf *material*) et de la contribution ambiante,
- de la contribution de toutes les sources lumineuses.

Plan de coupe et prise en compte des faces arrières :
glLightModelfv(LIGHT_MODEL_TWO_SIDE, bool)

Propriétés de la matière

glMaterial(face, type, param)*

Type	Valeur par défaut
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)
GL_AMBIENT_AND_DIFFUSE	
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)
GL_SHININESS	0.0
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)

Diminution du coût pour ambient et diffuse en prenant en compte la couleur courante (*glColor**), rendu actif par glEnable(GL_COLOR_MATERIAL).

Plan

1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

Mélange (*blending*)

- `glEnable(GL_BLEND);`
- calcul de la couleur finale :
  $(s_r.R_s + d_r.R_d,\ s_d.V_s + d_v.V_d,\ s_b.B_s + d_b.B_d,\ s_a.A_s + d_a.A_d)$,
- le calcul des *facteurs* est défini par la fonction *glBlendFunc(F_s, F_d)*, exemple en 2D :

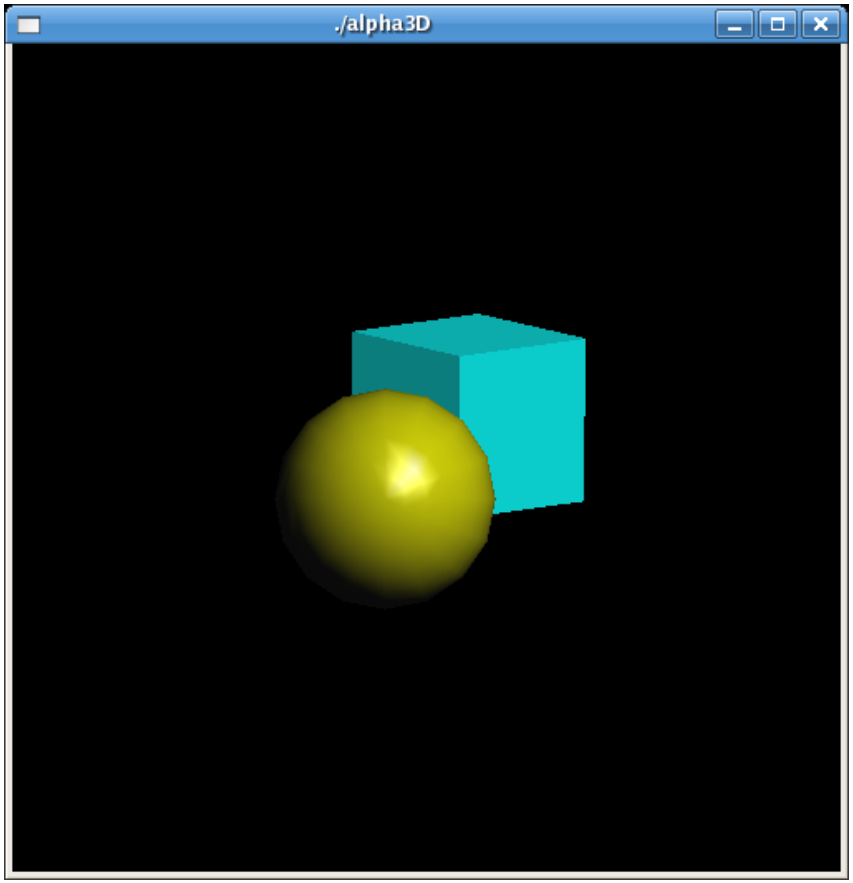
```
// 1er objet
glBlendFunc(GL_ONE, GL_ZERO); // 1xRs, 1xVs, 1xBs, 1xAs
...
// 2eme objet avec alpha = 0,25 (0,25xRs + 0,75xRd, ...)
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
// 25% opaque, 75% transparent
```

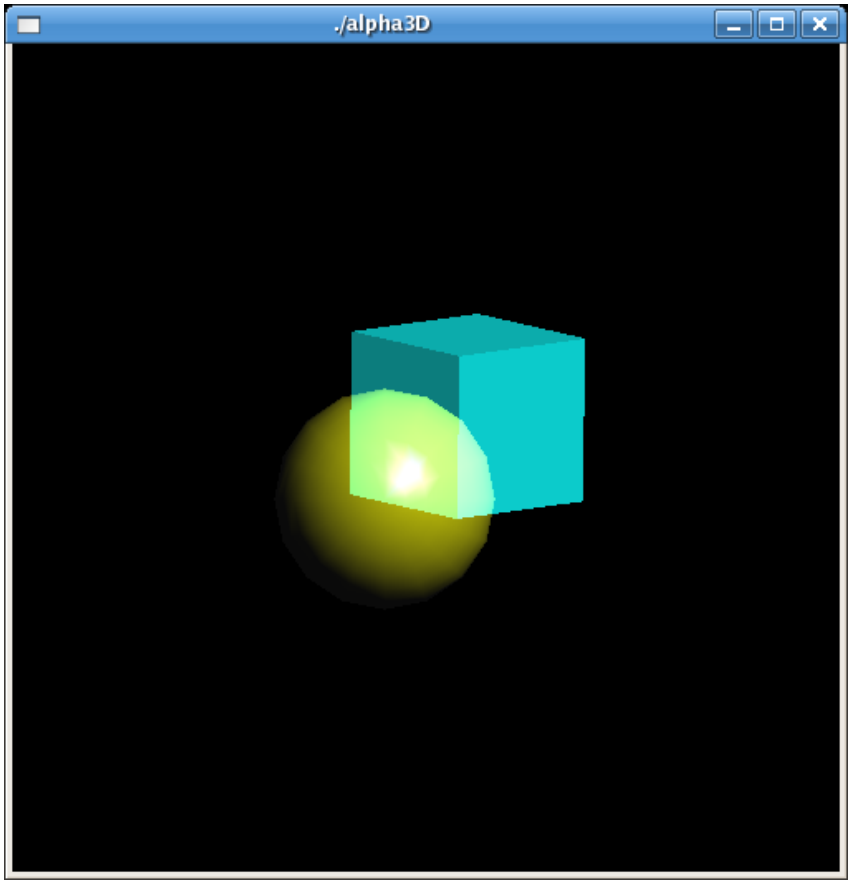
- la couleur résultante ne doit pas dépendre de l'*ordre* dans lequel on affiche les primitives graphiques : *glDepthMask(GL_FALSE)*

Blending : application à la transparence

```
void display(void)
{
   GLfloat mat_solid[] = { 0.75, 0.75, 0.0, 1.0 };
   GLfloat mat_zero[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat mat_transparent[] = { 0.0, 0.8, 0.8, 0.6 };
   GLfloat mat_emission[] = { 0.0, 0.3, 0.3, 0.6 };

   glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   glPushMatrix ();
      glTranslatef (-0.15, -0.15, solidZ);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_zero);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_solid);
      draw_sphere();
   glPopMatrix ();

   glPushMatrix ();
      glTranslatef (0.15, 0.15, transparentZ);
      glRotatef (15.0, 1.0, 1.0, 0.0);
      glRotatef (30.0, 0.0, 1.0, 0.0);
      glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
      glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_transparent);
      glEnable (GL_BLEND);
      glDepthMask (GL_FALSE);
      glBlendFunc (GL_SRC_ALPHA, GL_ONE);
      draw_cube();
      glDepthMask (GL_TRUE);
      glDisable (GL_BLEND);
   glPopMatrix ();
   ...
```

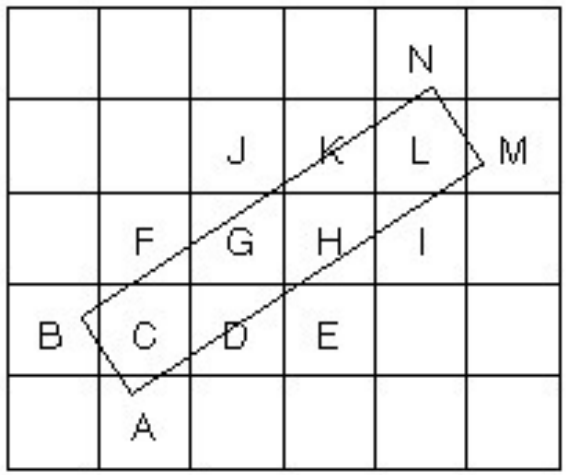
Blending : application à la transparence





Antialiasing

Multiplication de la valeur alpha du fragment par sa couverture (*glEnable(GL_BLEND)*) i.e la fraction en pourcentage de la surface du pixel recouvert par le fragment.



A	.040510
B	.040510
C	.878469
D	.434259
E	.007639
F	.141435
G	.759962
H	.759962
I	.141435
J	.007639
K	.434258
L	.878469
M	.040510
N	.040510

Antialiasing

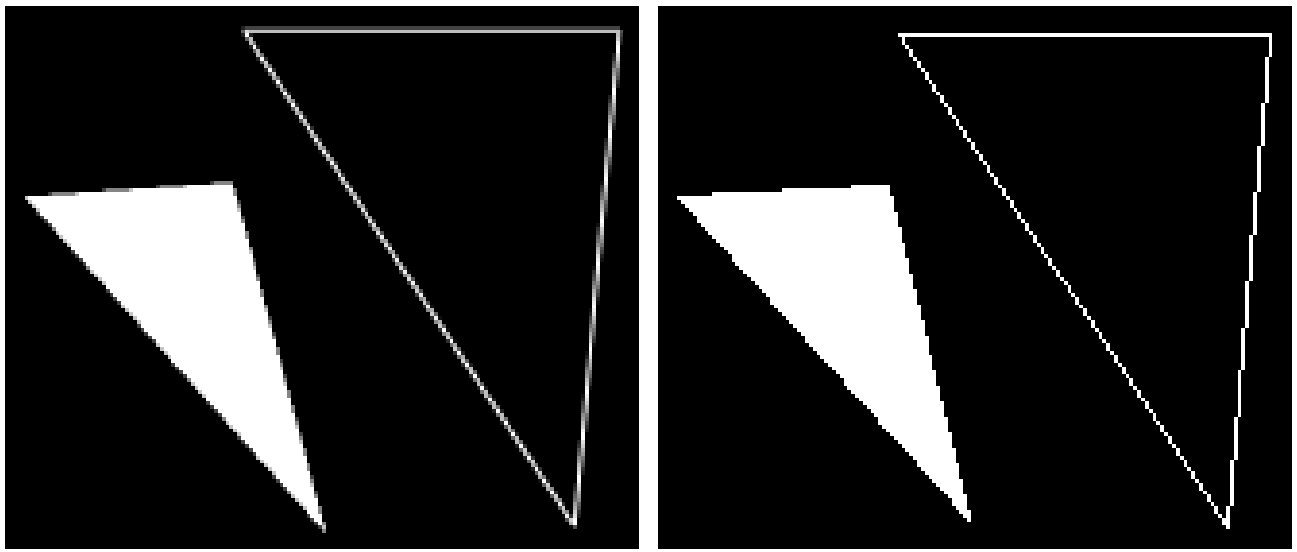
Détermination de la qualité du rendu : *glHint(cible, hint)* avec :

- cible :
  - `GL_POINT_SMOOTH_HINT`,
  - `GL_LINE_SMOOTH_HINT`,
  - `GL_FOG_HINT` (brouillard par pixel ou par sommet),
  - `GL_PERSPECTIVE_CORRECTION_HINT` (interpolation des couleurs et des textures),
- hint :
  - `GL_FASTEST` option la plus efficace,
  - `GL_NICEST` option grande qualité de rendu,
  - `GL_DONT_CARE` aucune préférence.

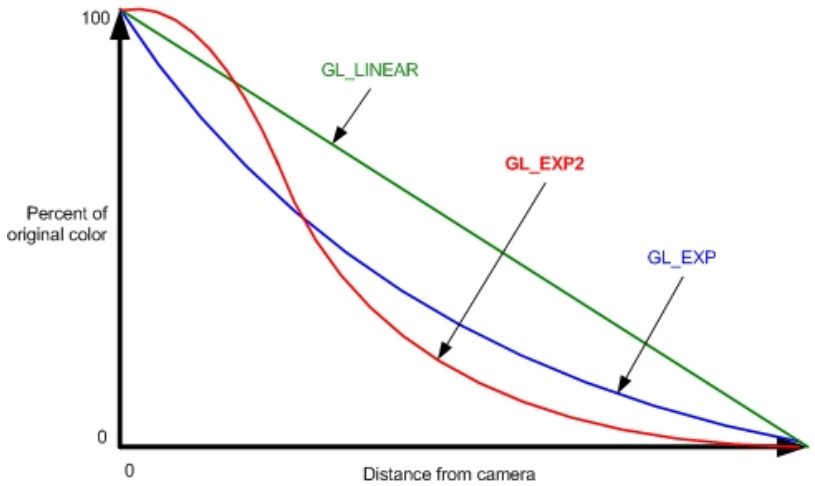
Antialiasing des points et des droites

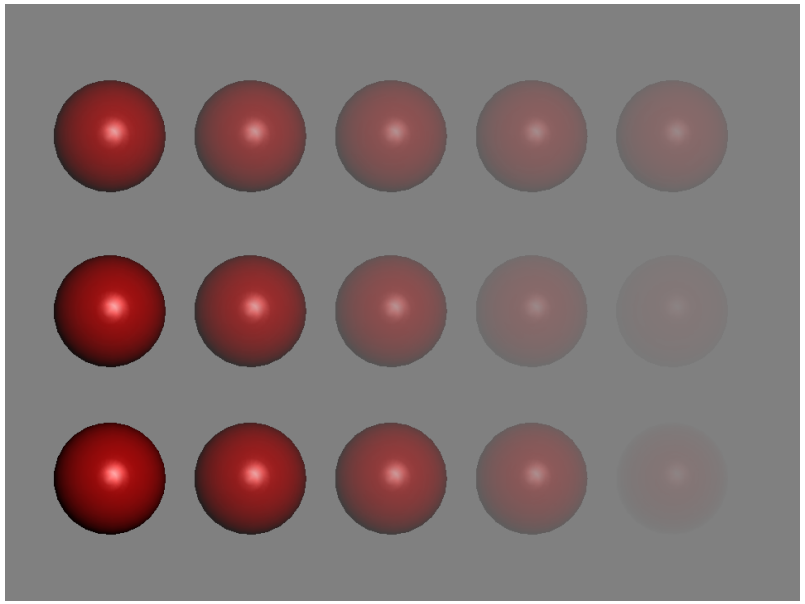
1. `glEnable(GL_POINT_SMOOTH)` ou `glEnable(GL_LINE_SMOOTH)`,
2. `glEnable(GL_BLEND)`,
3. `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.

A noter qu'effectuant des mélanges, on doit prendre en compte l'ordre du rendu.



Flou de distance (*fog*)





Flou de distance (*fog*)

Fonction appliquée à chaque fragment : mélange de la couleur courante et d'une couleur constante (de flou) suivant un facteur de poids (distance).

1. `glEnable(GL_FOG)`
2. *glFog*(nom, param)* : mélange de la couleur du brouillard (`GL_FOG_COLOR`) avec celle du fragment courant suivant un facteur f (`GL_FOG_MODE`) :
   - GL_EXP : $f = e^{-(densite*z)}$,
   - GL_EXP2 : $f = e^{-(densite*z)^2}$,
   - GL_LINEAR : $f = (end - z)/(end - start)$.

Exemple d'initialisation : *glFog*(*`GL_FOG_DENSITY`*, 0.35)*, *glFog*(*`GL_FOG_START`*, 1.0)*, *glFog*(*`GL_FOG_END`*, 5.0)*

Couleur finale $= f.C_i + (1 - f).C_f$, avec C_i couleur du fragment au bout du pipeline et C_f couleur du brouillard.

Plan

1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

Principes

Groupe de valeurs destinées à la mémoire image (*frame buffer*).

Non affectées par les opérations géométriques qui participent au pipeline graphique au cours de la discrétisation.

- de la mémoire image à la mémoire conventionnelle :

glPixelStorei(GL_PACK_ALIGNMENT, alignement)
glReadPixels(x, y, la, ha, format, type, pixels[])

avec *format* = `GL_RGBA` et *type* = `GL_UNSIGNED_BYTE` (ou `GL_IMPLEMENTATION_`)

- de la mémoire conventionnelle à la mémoire image :

glPixelStorei(GL_UNPACK_ALIGNMENT, alignement)
glTexImage2D(GL_TEXTURE_2D, level, internal_format, la, ha, bordure, format, type, pixels[])

Définition d'une texture

Etape 1 : spécification de la texture, chaque texel comporte 1, 2, 3 ou 4 élément(s) (des quadruplets RVBA) :

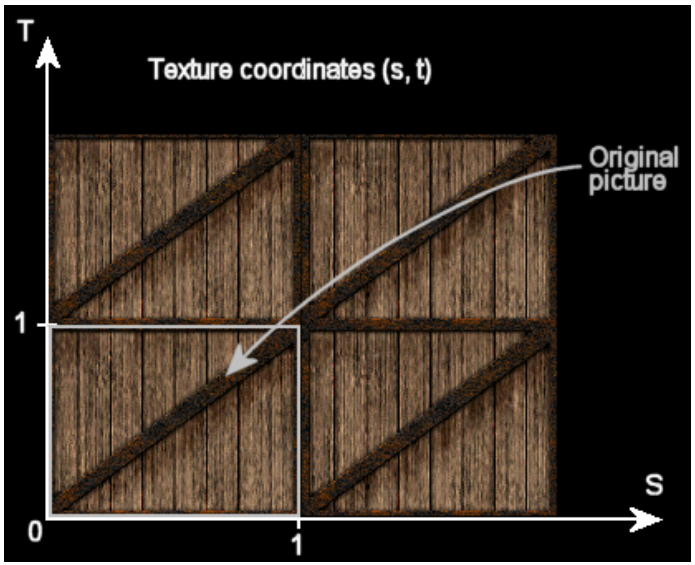
- *glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, la, ha, bordure, format, type, *pixels)* avec
  - 0 : niveau de détails (mipmaps),
  - la (2^m), ha (2^n), bordure (0),
  - format : `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE` ou `GL_LUMINANCE_ALPHA`,
  - type : `GL_UNSIGNED_BYTE` (ou `SHORT`).
- fonctions de filtrage, appliquées lors du placage sur la primitive :

  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_[ST], GL_[CLAMP, REPEAT]) ;

  glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_[MAG, MIN]_FILTER, GL_[NEAREST, LINEAR]) ;

Définition d'une texture

1. glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
2. glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);



Définition d'une texture (suite)

Etape 2 : définition de son influence sur chaque pixel

- mode DECAL (le texel remplace le pixel),
- MODULATE (la couleur du fragment est transformée),
- BLEND (mélange simple)

glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_[DECAL, MODULATE, BLEND]),

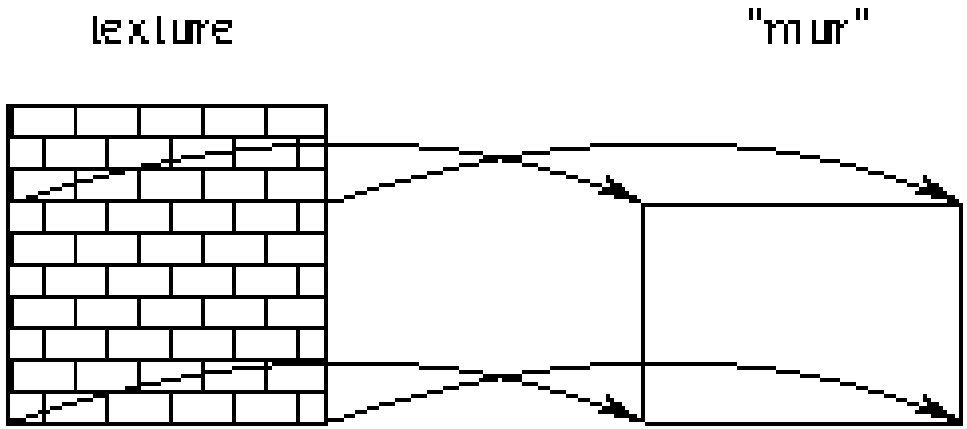
Etape 3 : autoriser le placage de texture : *glEnable(GL_TEXTURE_1D)* (2D le plus souvent),

Etape 4 : construire une scène, en spécifiant les coordonnées géométriques de texture.

Exemple

Mur de largeur la et de hauteur $ha = 2/3la$:

```
GLfloat varray[] = {-1,0, 1,0, -1,3, 1,3};
GLfloat tarray[] = {0,0, 1,0, 1,2/3, 0,2/3};

glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
   glTexCoordPointer(2, GL_FLOAT, 0, tarray);
   glVertexPointer(2, GL_FLOAT, 0, varray);
   glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);
```



Plan

1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

La mémoire image (frame buffer)

Destination des fragments, correspondant à un tableau $m \times n$ de pixels.

Les plans sont regroupés dans différents buffers logiques (couleur, profondeur, "stencil" et accumulation) :

- buffer couleur (*color buffer*) et buffer de profondeur (*depth buffer*),
- buffer "stencil" (*stencil buffer*) : valeurs mises à jour lorsqu'un fragment atteint la mémoire image (algorithmes à plusieurs passes, restriction d'écriture à certaine zone de l'affichage, ...),
- buffer d'accumulation (*accumulation buffer*) : données RVBA, accumulation d'une série d'images en une image finale composite (algorithmes multi-passes : anti-aliasing, effet de profondeur, flou de mouvement).

Opérations *per-fragment*

Un fragment est soumis à une série de tests et de modifications, avant d'être placé dans le buffer image, dans cet ordre :

1. test de découpe (*scissor test*), proche du test stencil mais portion rectangulaire uniquement,
2. test alpha (objets transparents ou textures partiellement transparentes (*billboarding*)) :
    1. objets avec $\alpha = 1$,
    2. objets avec $\alpha \neq 1$ et le test de profondeur inhibé.
3. test stencil, choix de la fonction de comparaison, de la valeur de référence, de la modification appliquée au fragment,
4. test de profondeur (*depth test*),
5. mélange (*blending*),
6. *dithering*.