OpenGL

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 assurée par l'absence 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),
- 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),

Fonctionnalités d'OpenGL

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* (ou programmable par shaders [1] à partir de la version 2). Construction des *primitives* dans la mémoire image (*frame buffer*).

1. GLSL : OpenGL Shading Language

Fonctionnalités d'OpenGL

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.

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 (glut)

Bibliothèques développées afin d'apporter des fonctionnalités non-disponibles dans OpenGL (contextes graphiques, tampons mémoire, synchronisation du rendu).

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

glut : bibliothèque très simple pour les démonstrations.

- glutInit, glutInitDisplayMode, ...
- glutCreateWindow, glutDestroyWindow, ...
- glutGet, glutGetWindowWidth, glutGetWindowHeight, ...
- glutDisplayFunc, glutReshapeFunc, glutKeyboardFunc, glutSpecialFunc, glutIdleFunc, ...
- glutPostRedisplay,
- glutMainLoop.

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.

Mesa est une implantation libre de cette bibliothèque, 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).

Remarque : sous Linux, la commande `glxinfo -B` permet de connaître la version supportée.

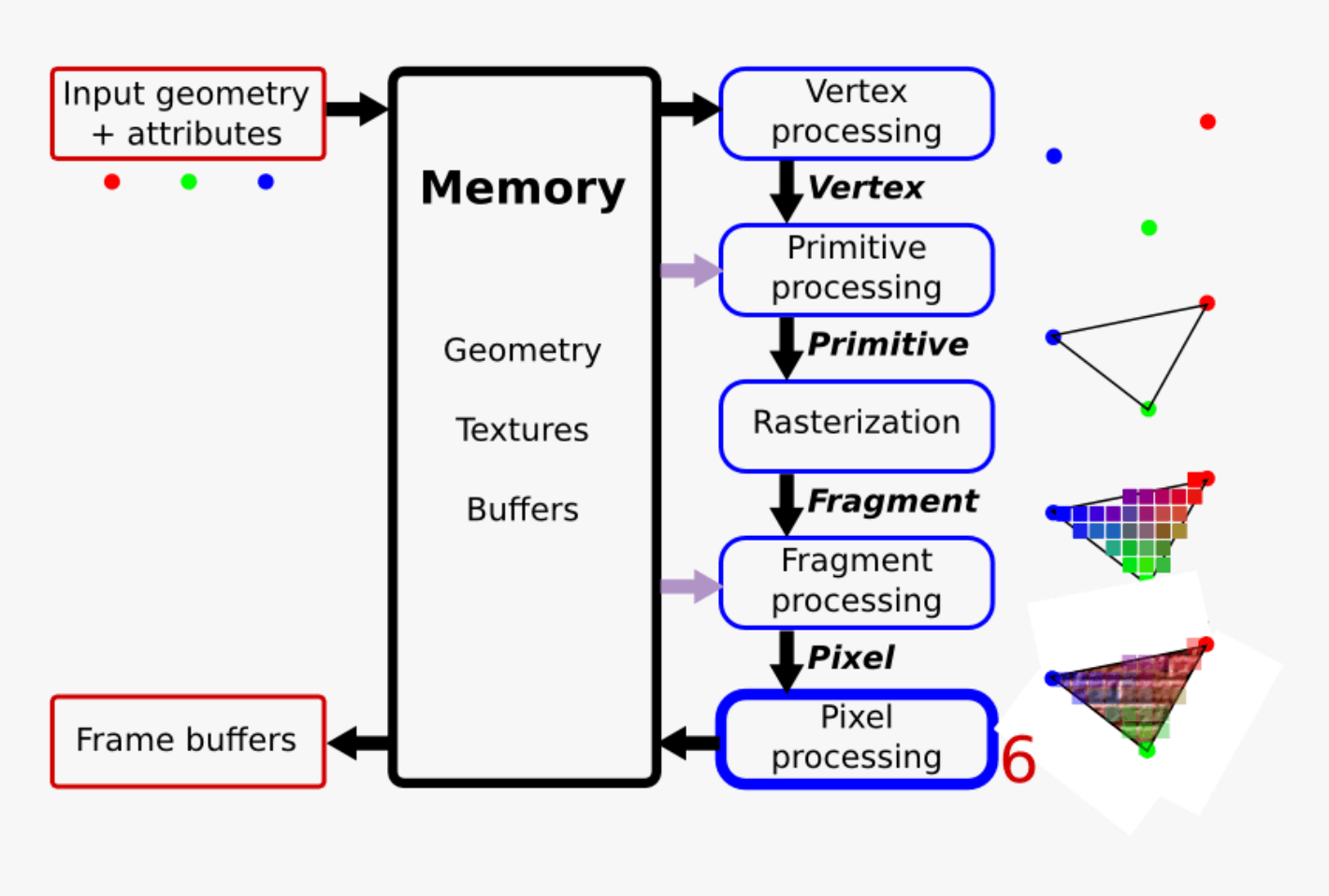
Utilisations

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



Des opérations élémentaires

Le pipeline graphique d'OpenGL

1. Per-vertex operations et *primitive assembly* : traitement des sommets qui définissent les primitives géométriques (points, segments, polygones).
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) avant d'instancier la mémoire image.

Les textures 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.

Per-vertex operations

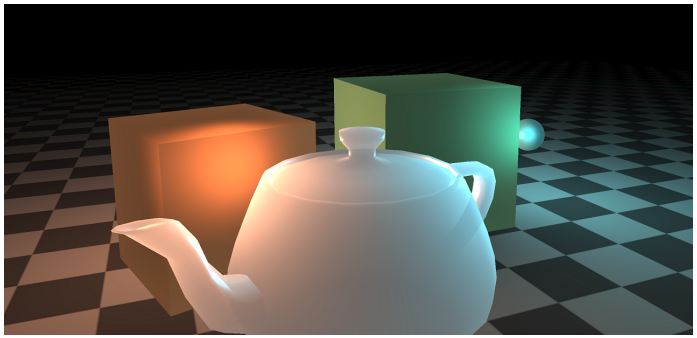
Unités chargées de traiter les sommets :

1. *backface culling* (élimination des faces arrières),
2. *clipping* (élimination des portions d'objets hors du volume visualisé),
3. *perspective divide* (déformation des objets dûe à la projection perspective),
4. *viewport transform* (adaptation de la taille des objets au périphérique d'affichage)
5. *setup engine* (prise en compte de la réflexion diffuse et spéculaire, *texture mapping* et flous).

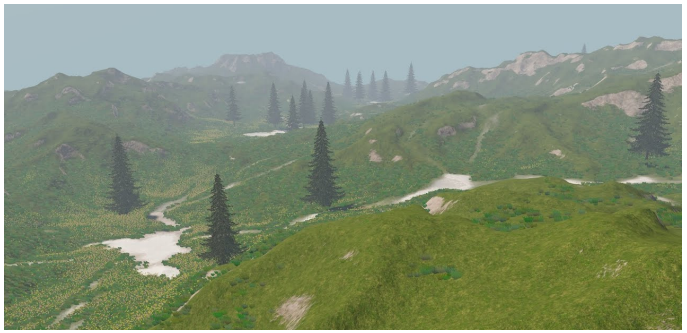
Per-fragment operations

Unités chargées du mélange de textures et d'éclairages :

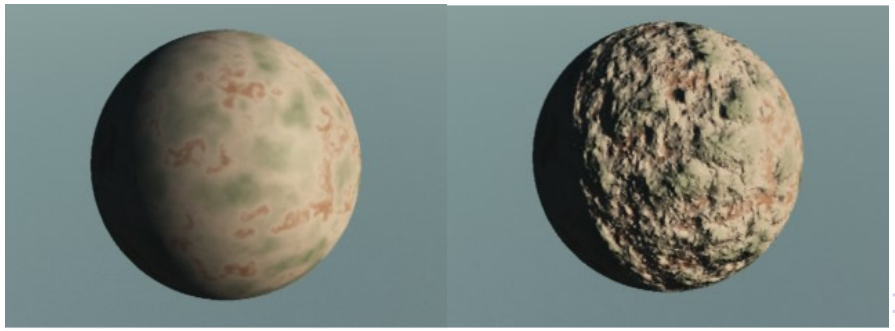
- modèles d'ombrage particuliers,



- textures procédurales,



- *bump mapping*.

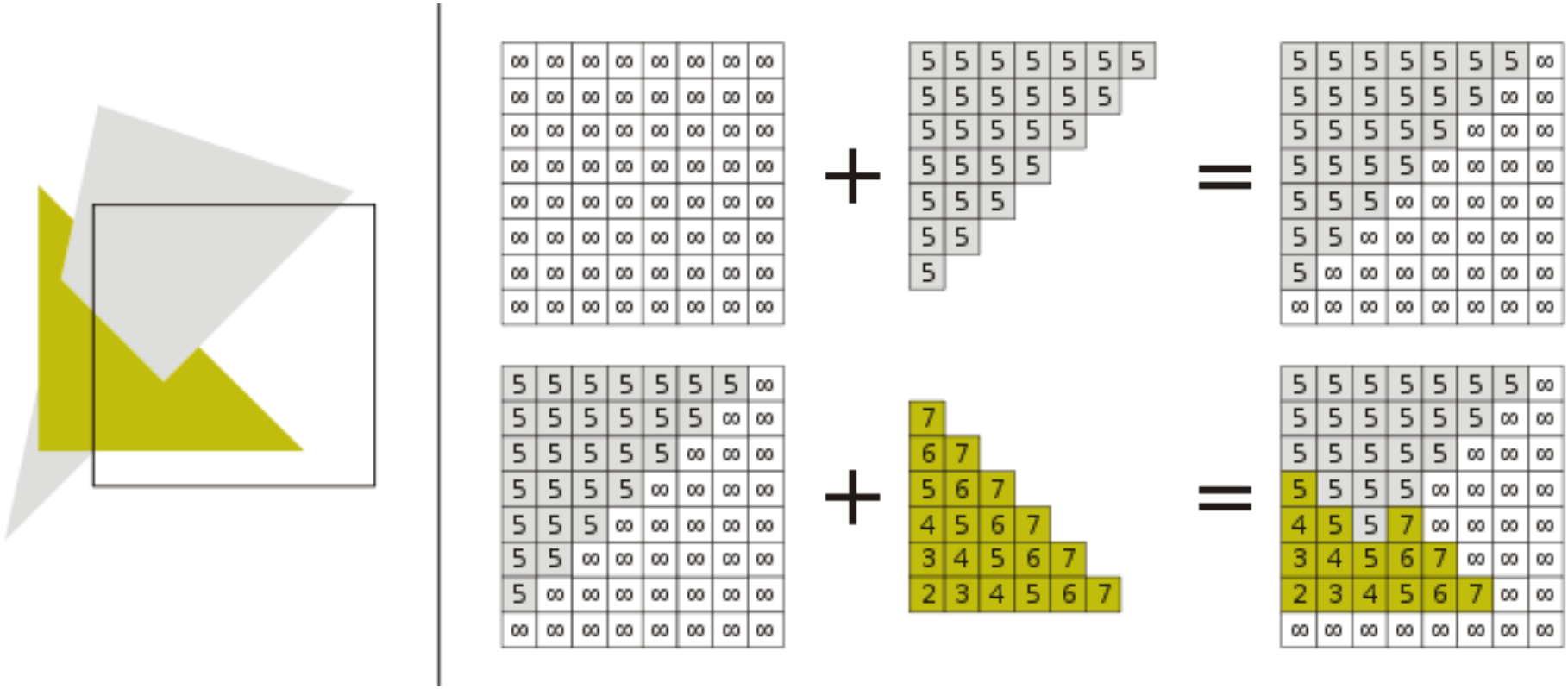


Mémoire vidéo

Elle est compartimentée et chaque zone a une utilisation bien définie :

- *frame buffer* (image définitive de ce qui va être affiché à l'ecran),
- *back buffer* (zone de mémoire stockant le rendu d'une scène en cours de création),
- *depth*, *stencil* et *accumulation buffers* (resp. Z-Buffer, ombrage volumique, flou en temps réel, etc.),
- *vertex buffer* (stocke les sommets des objets 3D),
- *texture buffer* (remarque : une seule texture de 2048×2048 en 24 bits occupe environ 12 Mo), occupation la plus importante de la VRAM.

Z-buffer

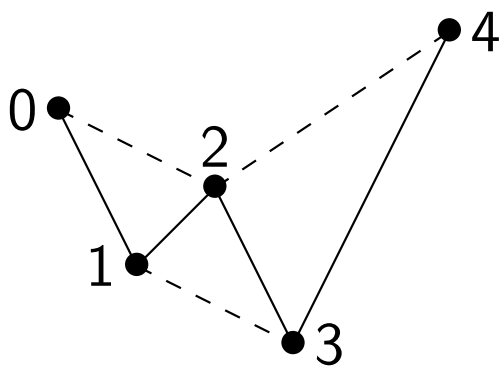

∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
5 5 5 5 5 5 5
5 5 5 5 5 5
5 5 5 5 5
5 5 5 5
5 5 5
5 5
5
5 5 5 5 5 5 5 ∞
5 5 5 5 5 5 ∞ ∞
5 5 5 5 5 ∞ ∞ ∞
5 5 5 5 ∞ ∞ ∞ ∞
5 5 5 ∞ ∞ ∞ ∞ ∞
5 5 ∞ ∞ ∞ ∞ ∞ ∞
5 ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
5 5 5 5 5 5 5 ∞
5 5 5 5 5 5 ∞ ∞
5 5 5 5 5 ∞ ∞ ∞
5 5 5 5 ∞ ∞ ∞ ∞
5 5 5 ∞ ∞ ∞ ∞ ∞
5 5 ∞ ∞ ∞ ∞ ∞ ∞
5 ∞ ∞ ∞ ∞ ∞ ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞
7
6 7
5 6 7
4 5 6 7
3 4 5 6 7
2 3 4 5 6 7
5 5 5 5 5 5 5 ∞
5 5 5 5 5 5 ∞ ∞
5 5 5 5 5 ∞ ∞ ∞
5 5 5 5 ∞ ∞ ∞ ∞
4 5 5 7 ∞ ∞ ∞ ∞
3 4 5 6 7 ∞ ∞ ∞
2 3 4 5 6 7 ∞ ∞
∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞

Plan

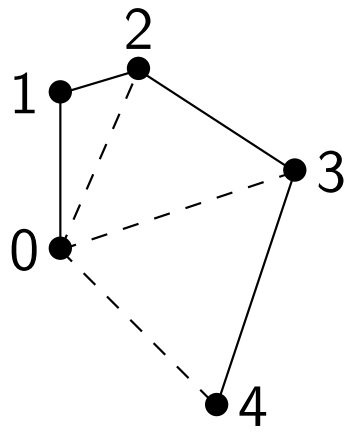
1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

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



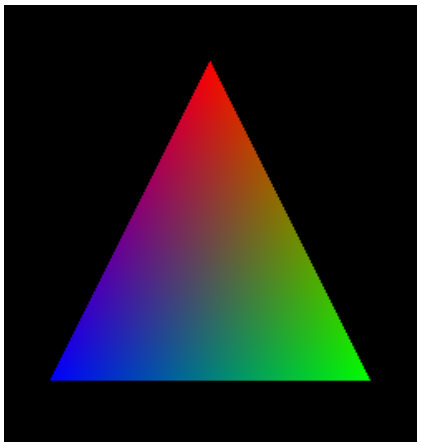
triangle strip



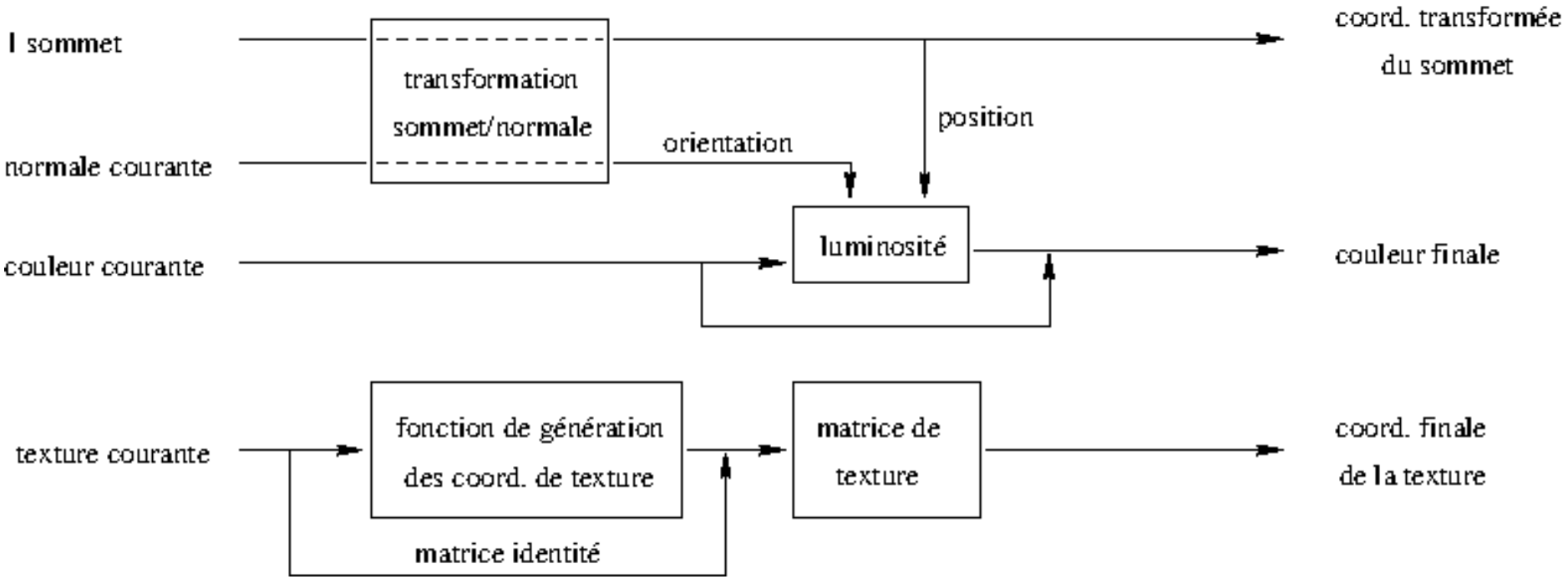
triangle fan

Un triangle

```
glBegin(GL_POLYGON)                     # Start drawing a polygon
    glColor3f(1.0, 0.0, 0.0)            # Red
    glVertex3f(0.0, 1.0, 0.0)           # Top
    glColor3f(0.0, 1.0, 0.0)            # Green
    glVertex3f(1.0, -1.0, 0.0)          # Bottom Right
    glColor3f(0.0, 0.0, 1.0)            # Blue
    glVertex3f(-1.0, -1.0, 0.0)         # Bottom Left
glEnd()                                 # We are done with the polygon
```



Sommets et primitives



Association des valeurs courantes et d'un sommet

Les triangles

Définition par :

- `glBegin(GL_TRIANGLES)`, `glColor3f()`, `glNormal3f()`, `glVertex3f()`, `glEnd()`,
- utilisation de tableaux (de couleurs, de sommets, de normales et de textures) pour des raisons d'efficacité.

Un triangle

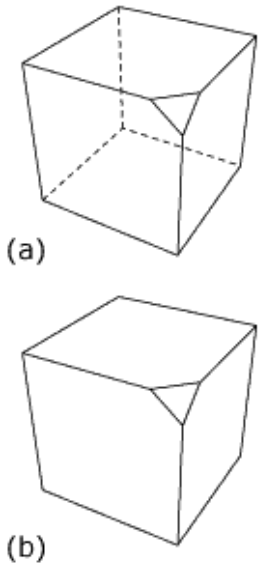
```
vertices = [-1.0,-1.0, 1.0,-1.0, 0.0,1.0]

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

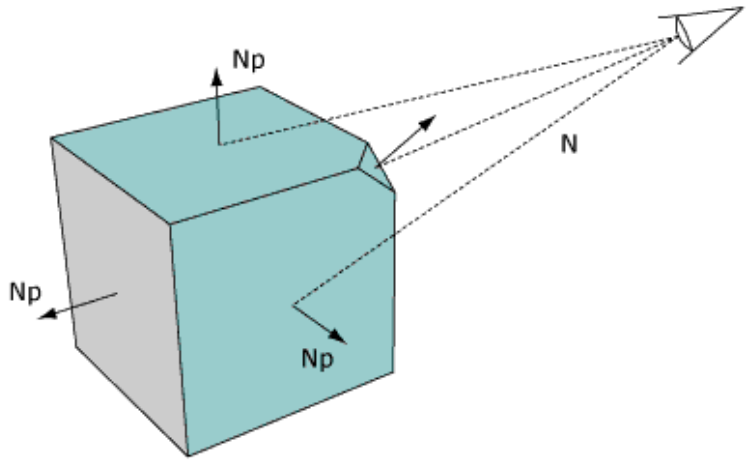
Les triangles

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





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 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

solides de base

```
from OpenGL.GL import *  # car prefixe systematique
from OpenGL.GLU import *
from OpenGL.GLUT import *

...

glutInit(sys.argv)
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH)

glutCreateWindow('solides basiques')
glutReshapeWindow(300, 300)

glutReshapeFunc(reshape)
glutDisplayFunc(display)
glutKeyboardFunc(keyboard)
glutSpecialFunc(special_key)

init()

glutMainLoop()
```

Exemple de scène

display()

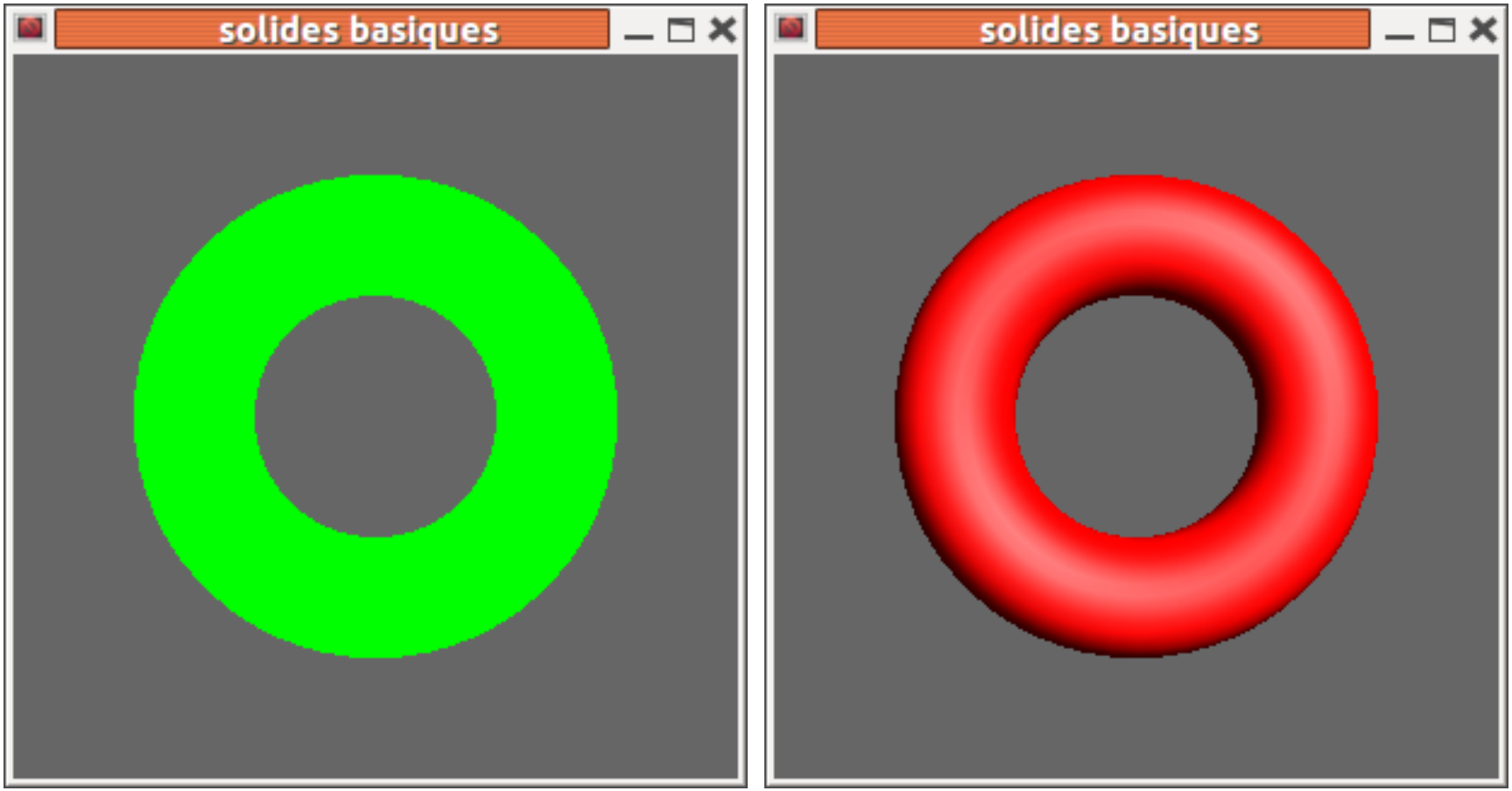
```
view_rotx, view_roty, view_rotz = 0.0, 0.0, 0.0

def display():
    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(), keyboard(), special_key()

```
def init():
   glClearColor(0.4, 0.4, 0.4, 0.0)
   glEnable(GL_DEPTH_TEST)


def keyboard(*args):
   global solide_courant
   key = args[0].decode('utf8')  # indispensable en Python 3
   if key == 't':
      solide_courant = TORE
   elif ...


def special_key(*args):
   if args[0] == EGLUT_KEY_LEFT:
      view_roty += 5.0;
   elif ...
```

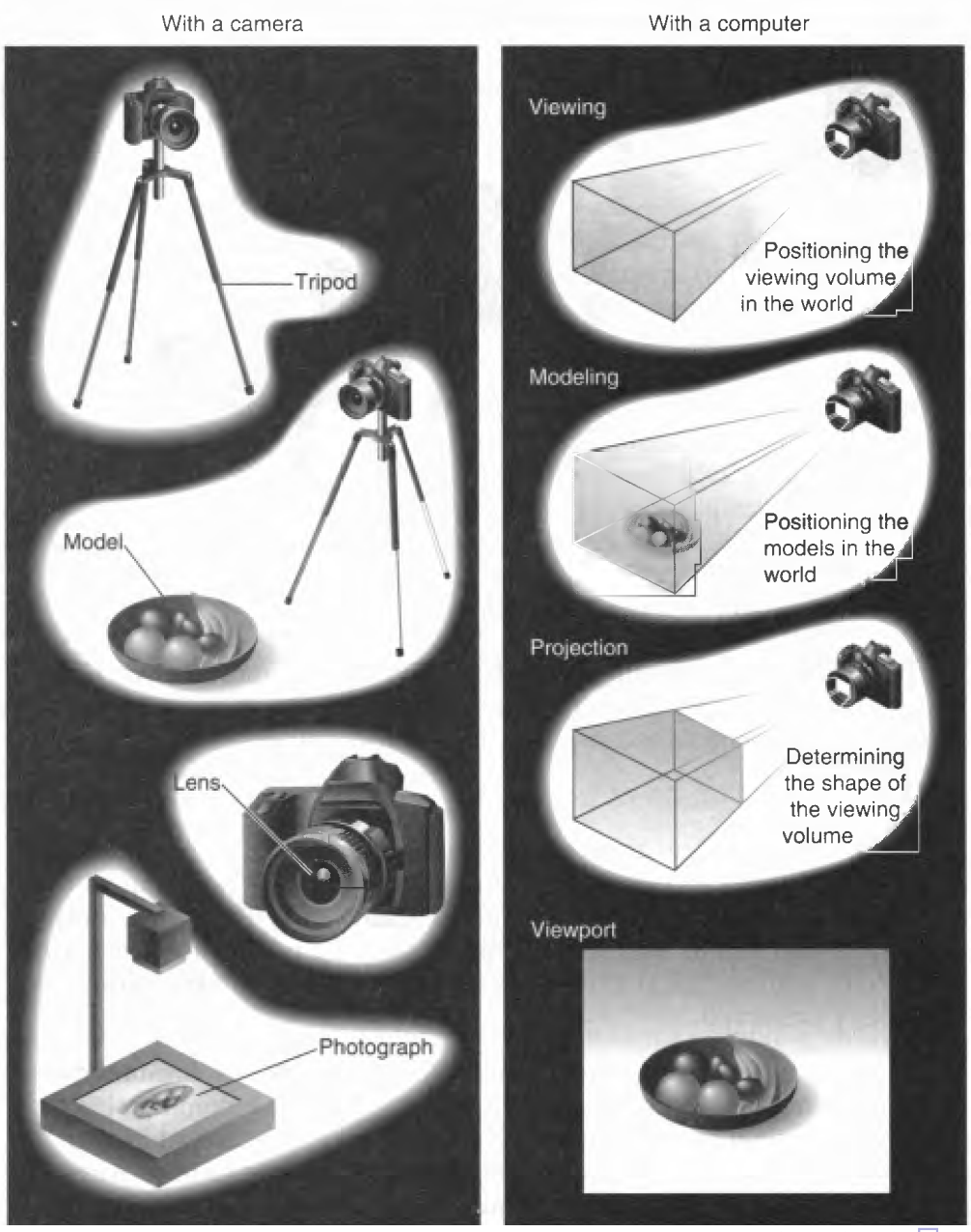
Copie d'écran



Plan

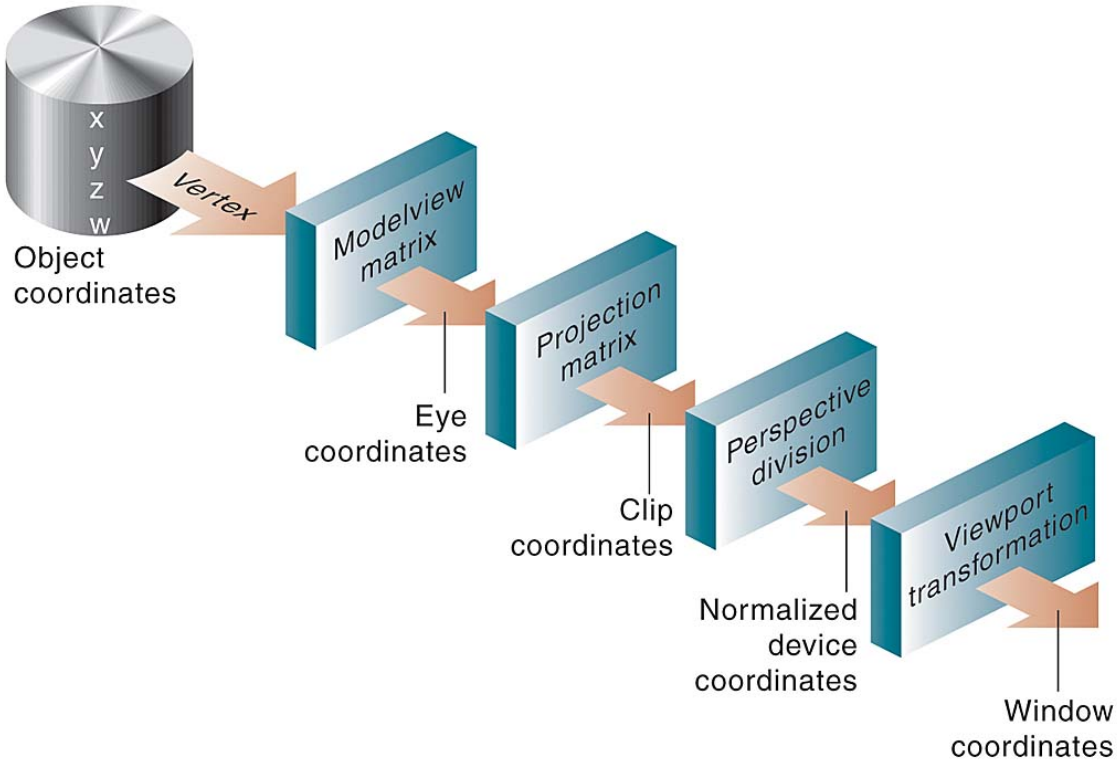
1. Sommets et primitives
2. Couleurs et affichage
3. Visualisation
4. Le réalisme
5. Mélange, antialiasing et flou
6. Manipulation d'images
7. Opérations sur les fragments

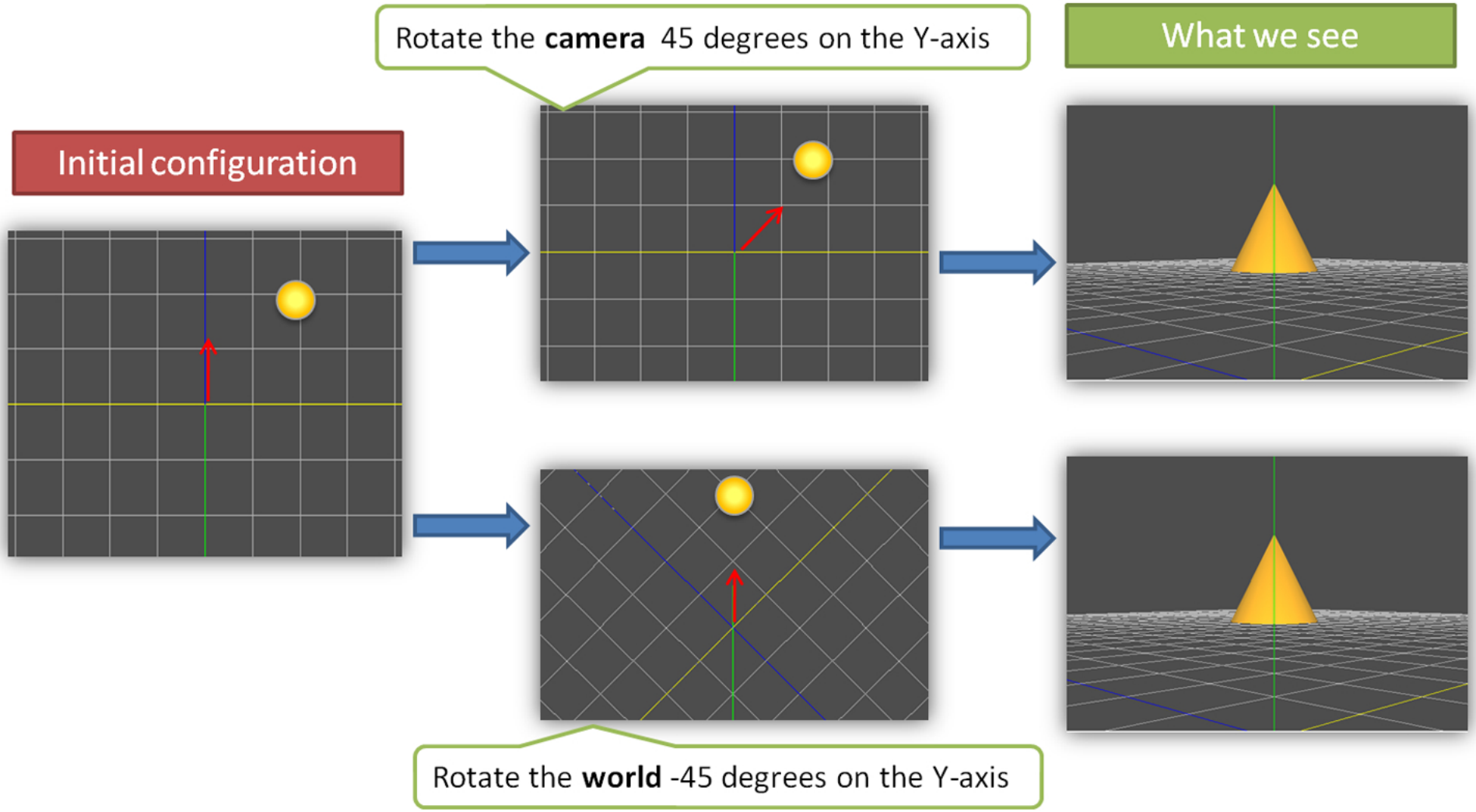
Visualisation



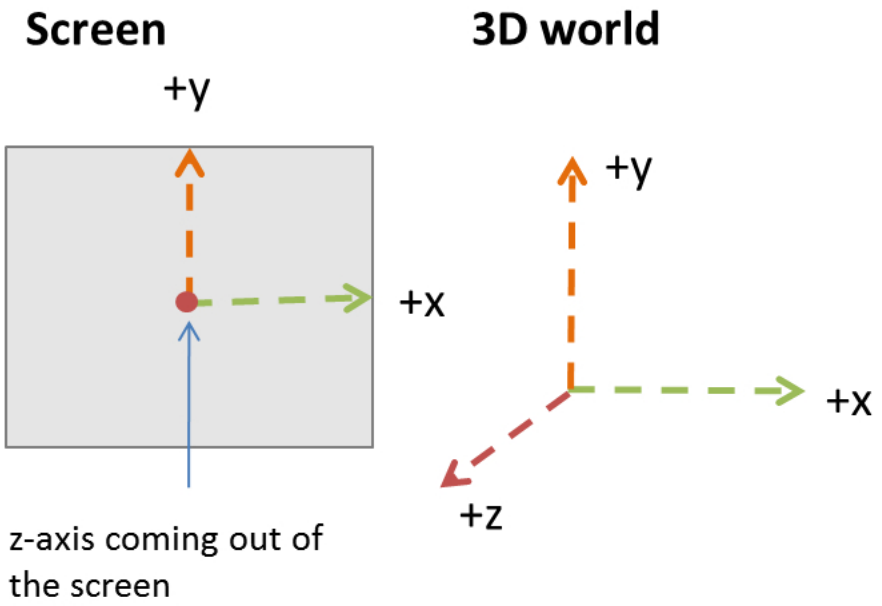
Visualisation

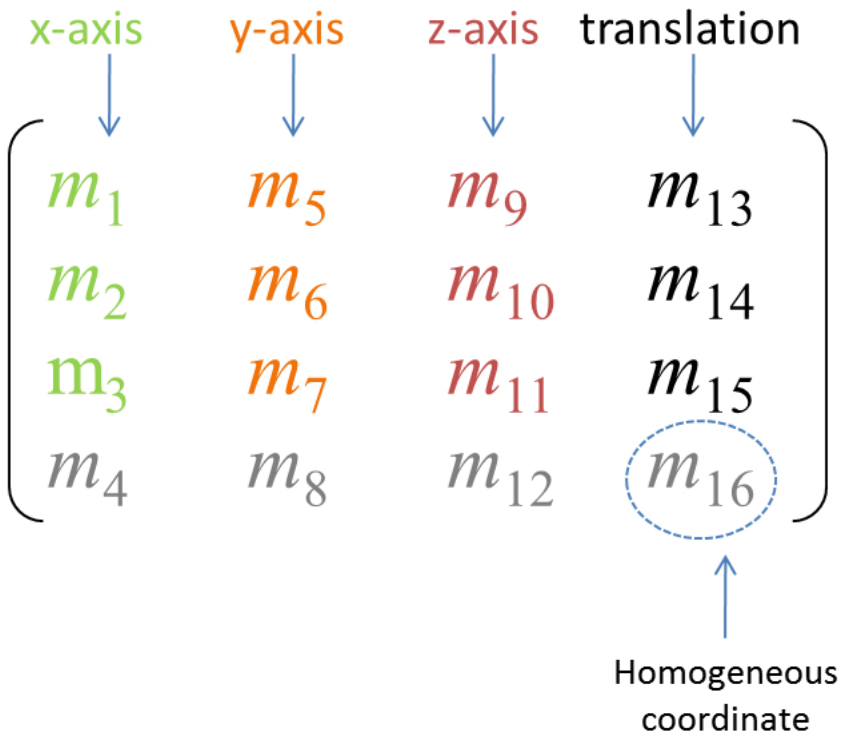
1. positionner, orienter la caméra et agencer la scène : une seule transformation (`glMatrixMode(GL_MODELVIEW);`),
2. choix de l'objectif (`glMatrixMode(GL_PROJECTION);`),
3. développement, choix de la taille de l'image papier (*viewport transformation*).

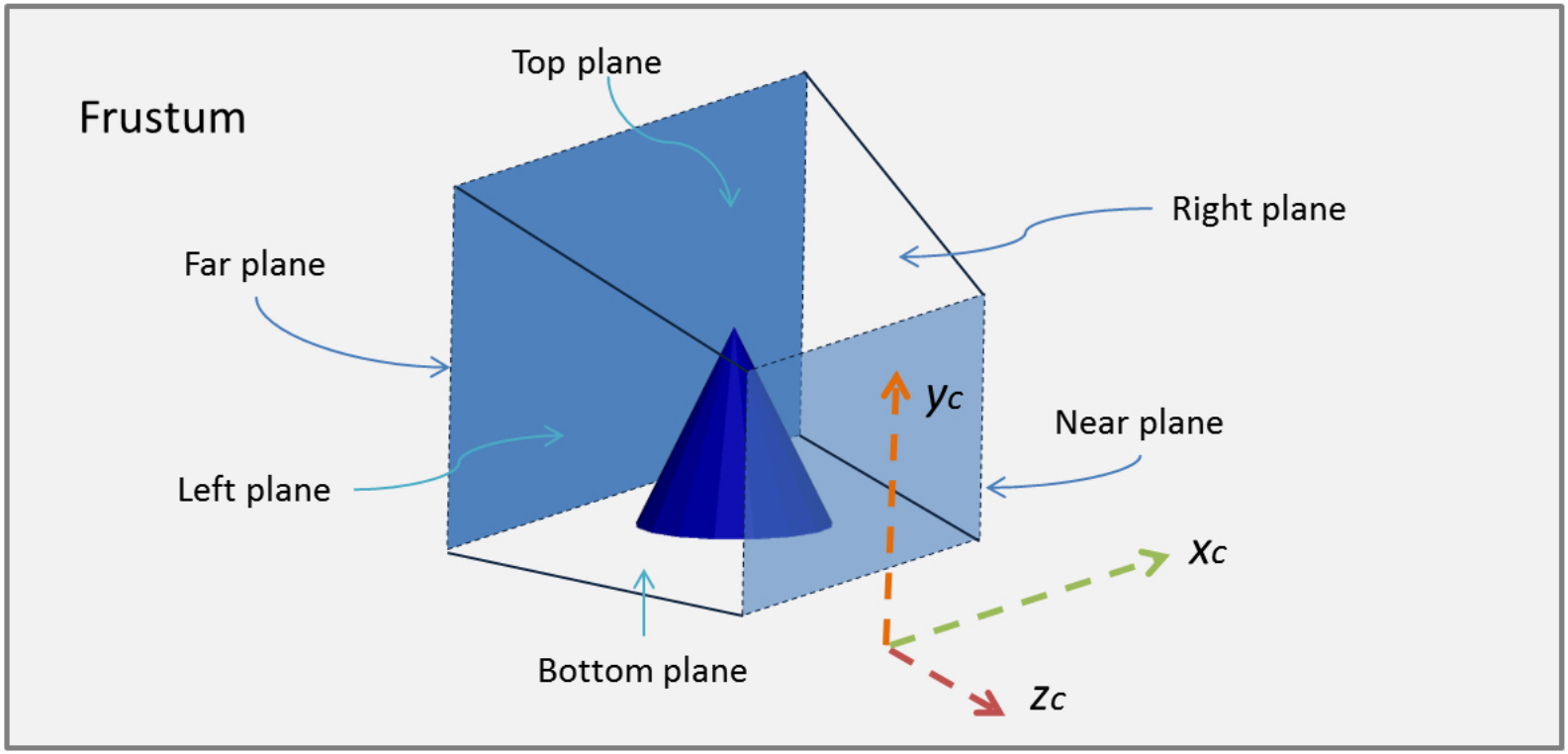



Rotate the **camera** 45 degrees on the Y-axis
What we see
Initial configuration
Rotate the **world** -45 degrees on the Y-axis

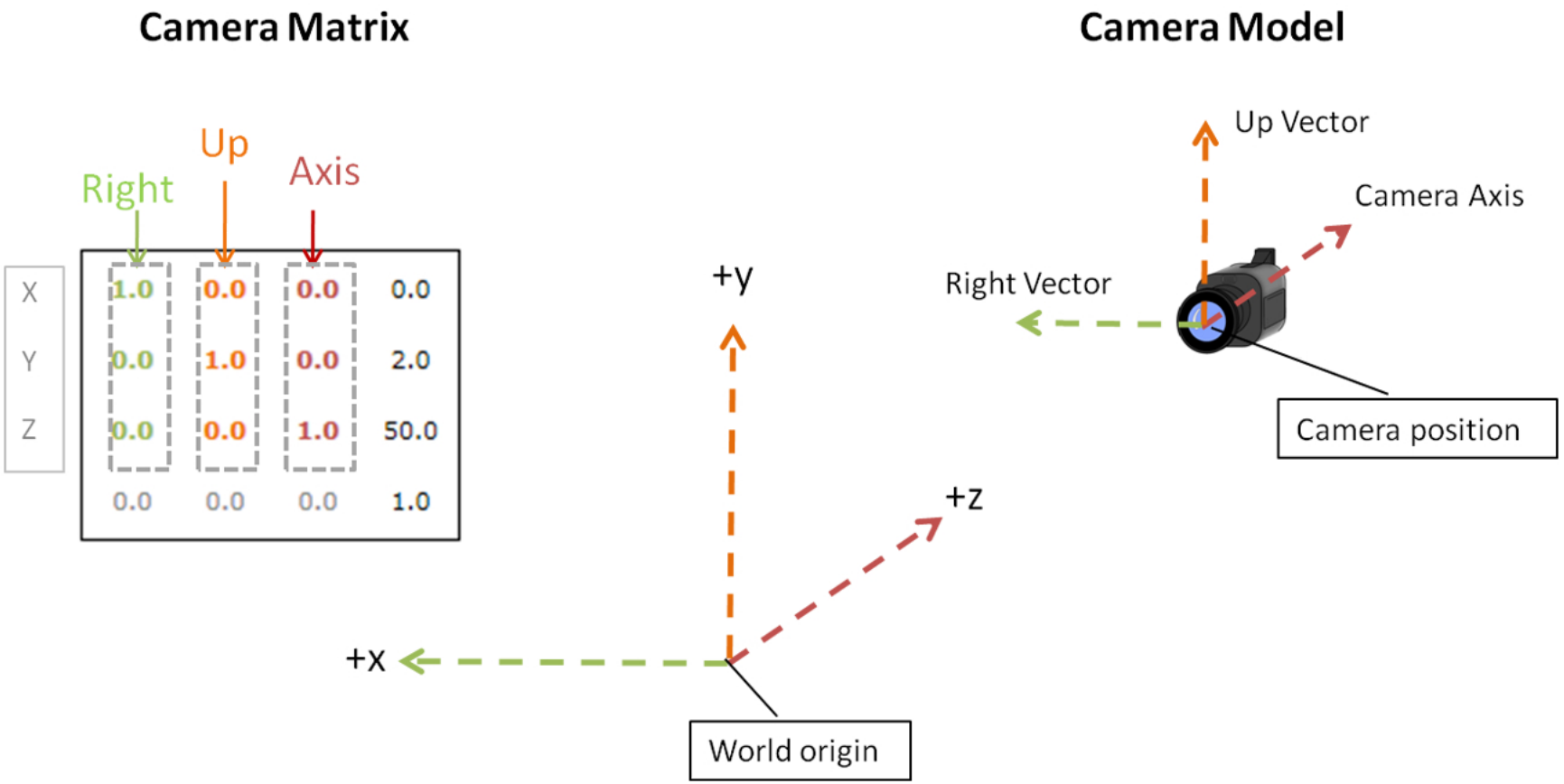
The model-view matrix

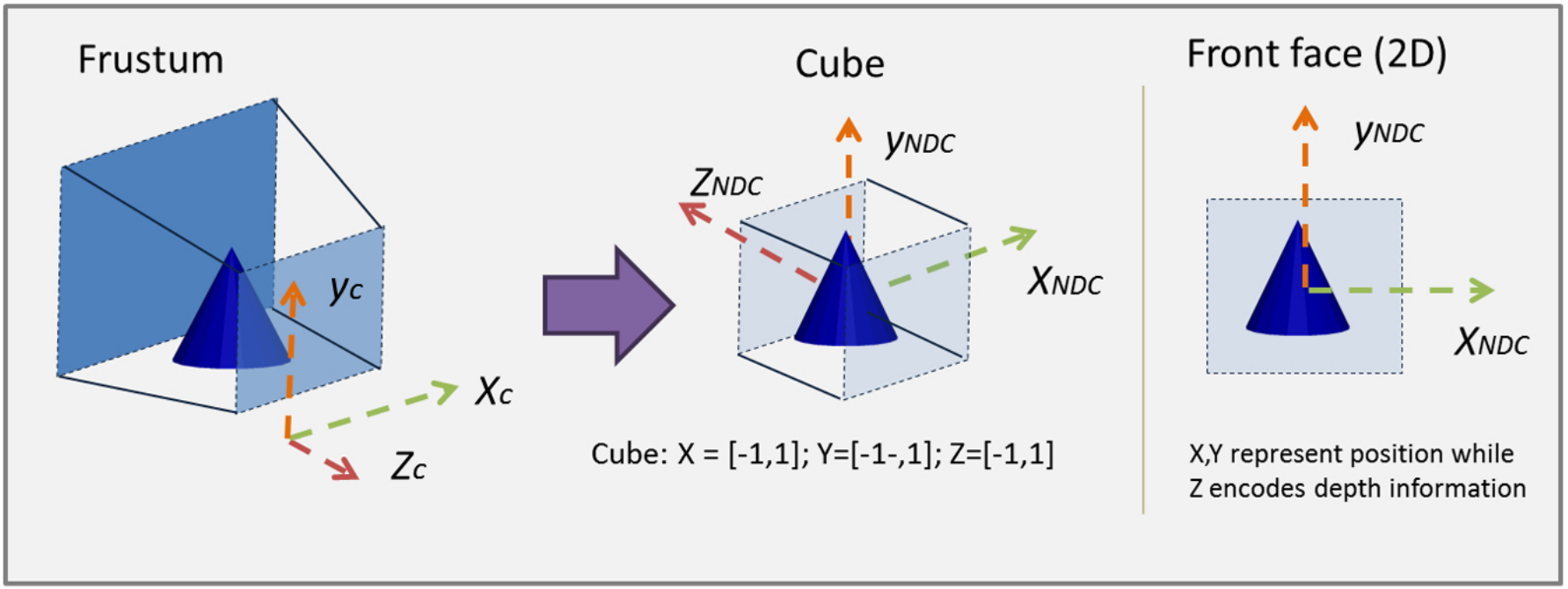




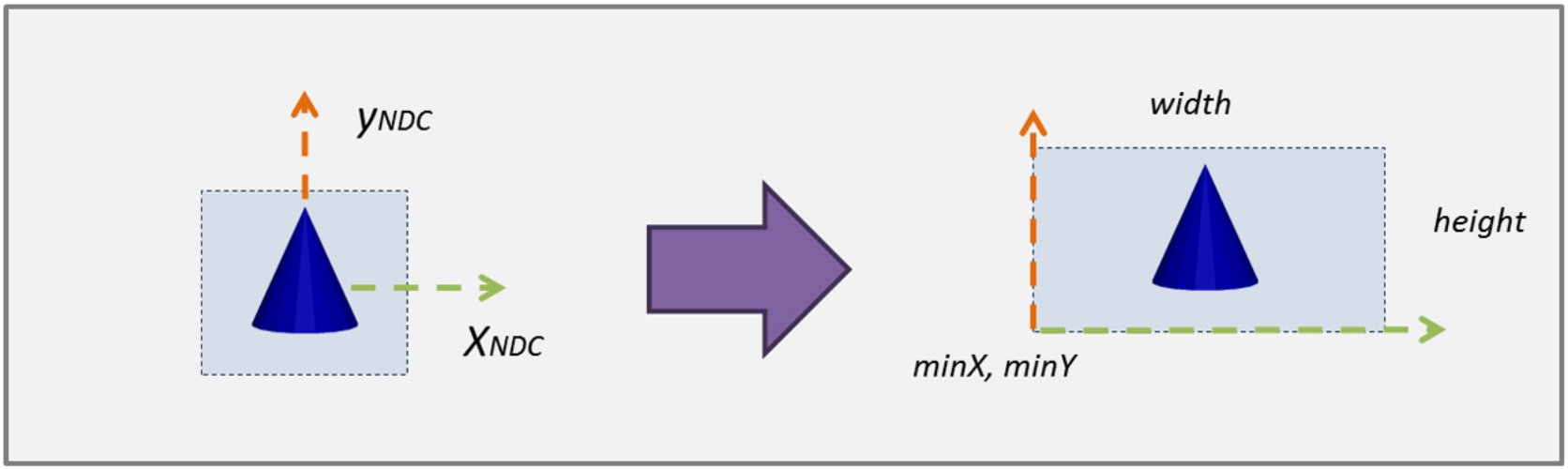

Frustum
Top plane
Right plane
Far plane
Left plane
Near plane
Bottom plane
yc
Xc
Zc

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

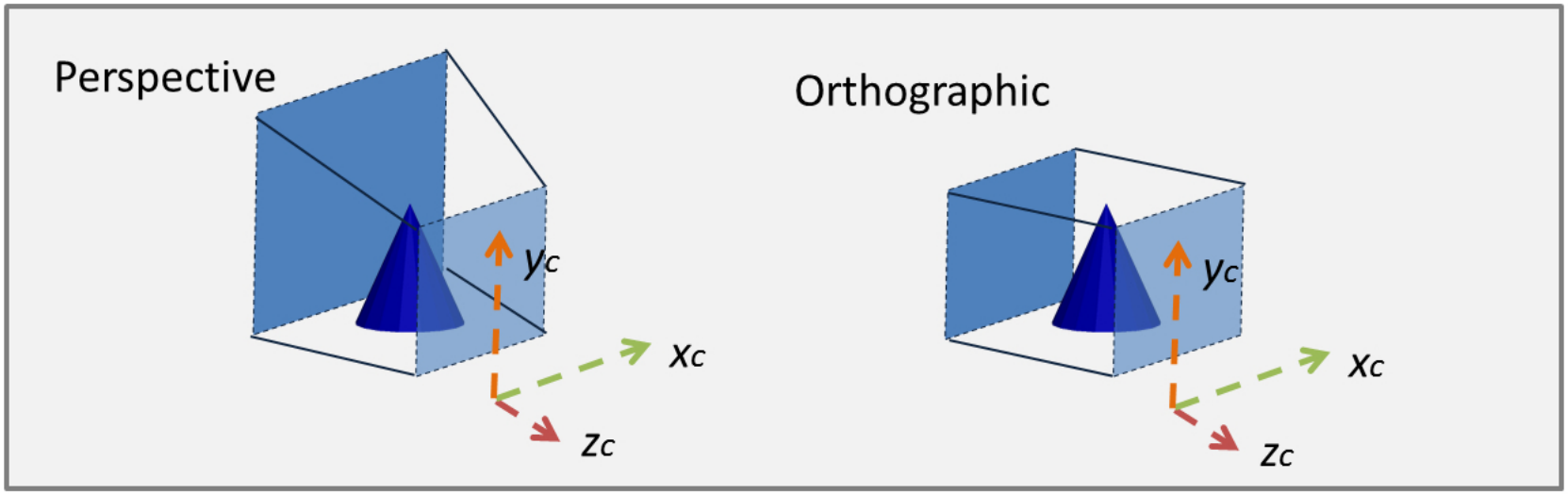
Clip Coordinates
Perspective Division
Normalized Device Coordinates


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 :

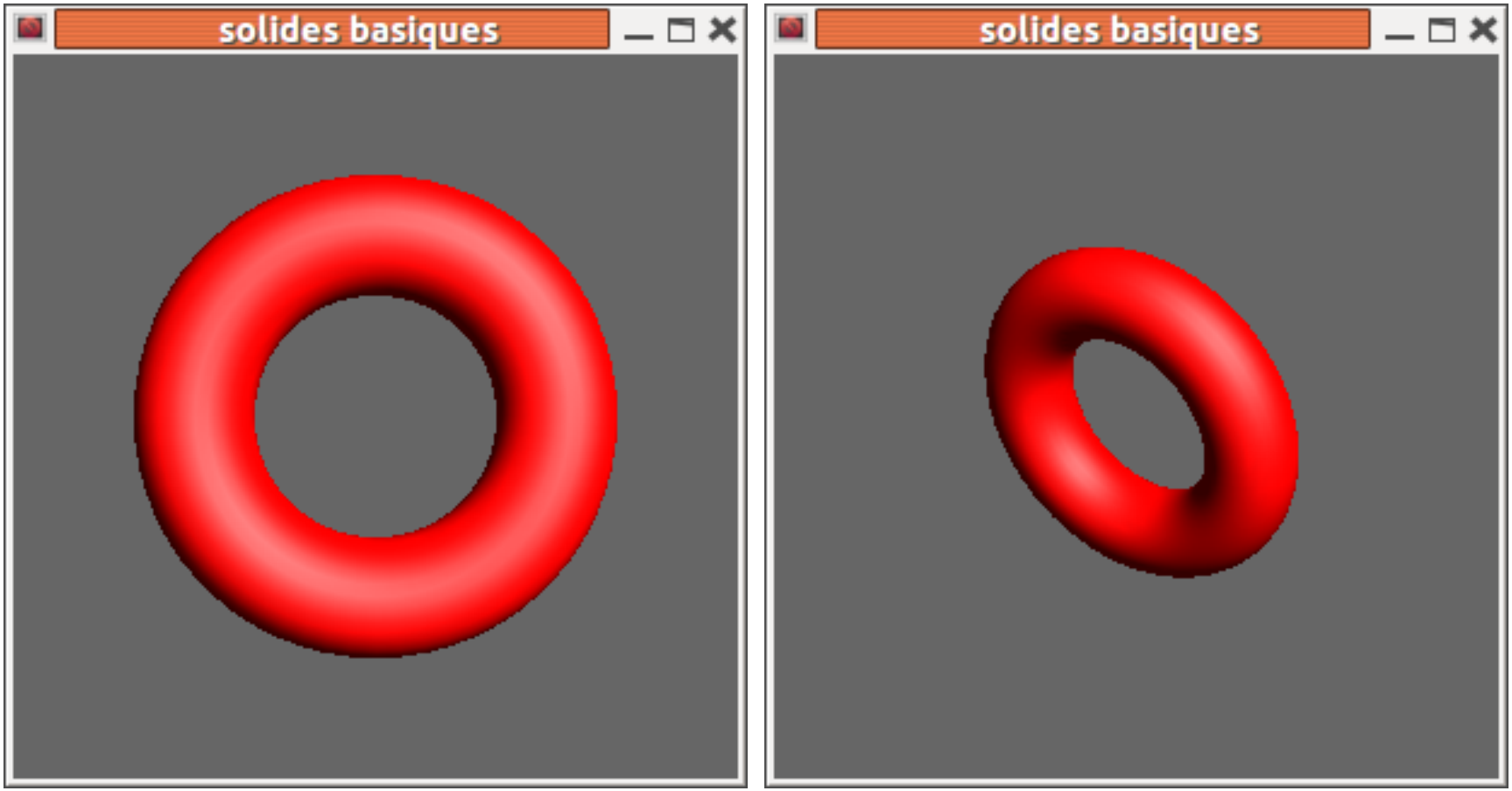
- *gluLookAt(point, refpoint, upvector)* : positionne intuitivement la caméra,
- *gluPerspective(T, aspect, near, far)* avec T : profondeur de champ.

Exemple de scène (2)

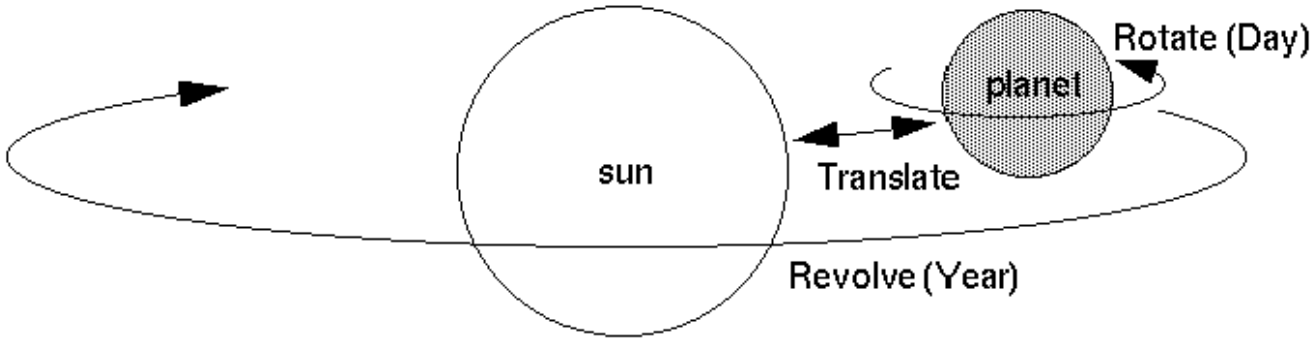
reshape()

```
def reshape(width, height):
    ar = GLfloat(width) / GLfloat(height)

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

    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()

    glFrustum(-ar, ar, -1, 1, 5.0, 60.0)

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

Copie d'écran (2)



Composition de transformations

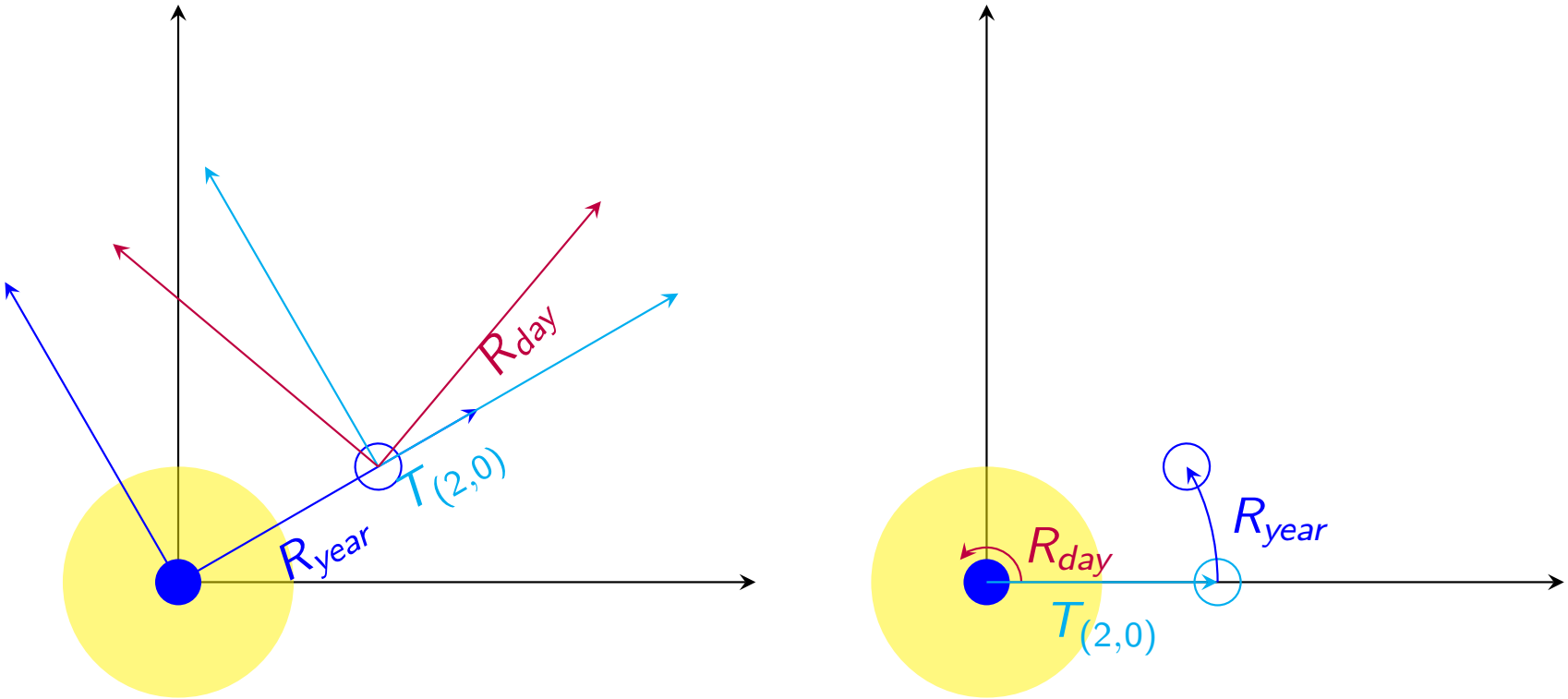


Exemple de mouvement Terre / Soleil

```
def display():
    ...
    gluSphere(sph, 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)
    gluSphere(sph, 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

```
def Roue_Boulons():
  Roue()
  for i in range(5):
    glPushMatrix()
      glRotatef(degre*i, 0, 0, 1)  # autour de Z
      glTranslatef(Xb, 0, 0)
      Boulon()  # ATTENTION : R*T*Boulon()
    glPopMatrix();

def 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 :

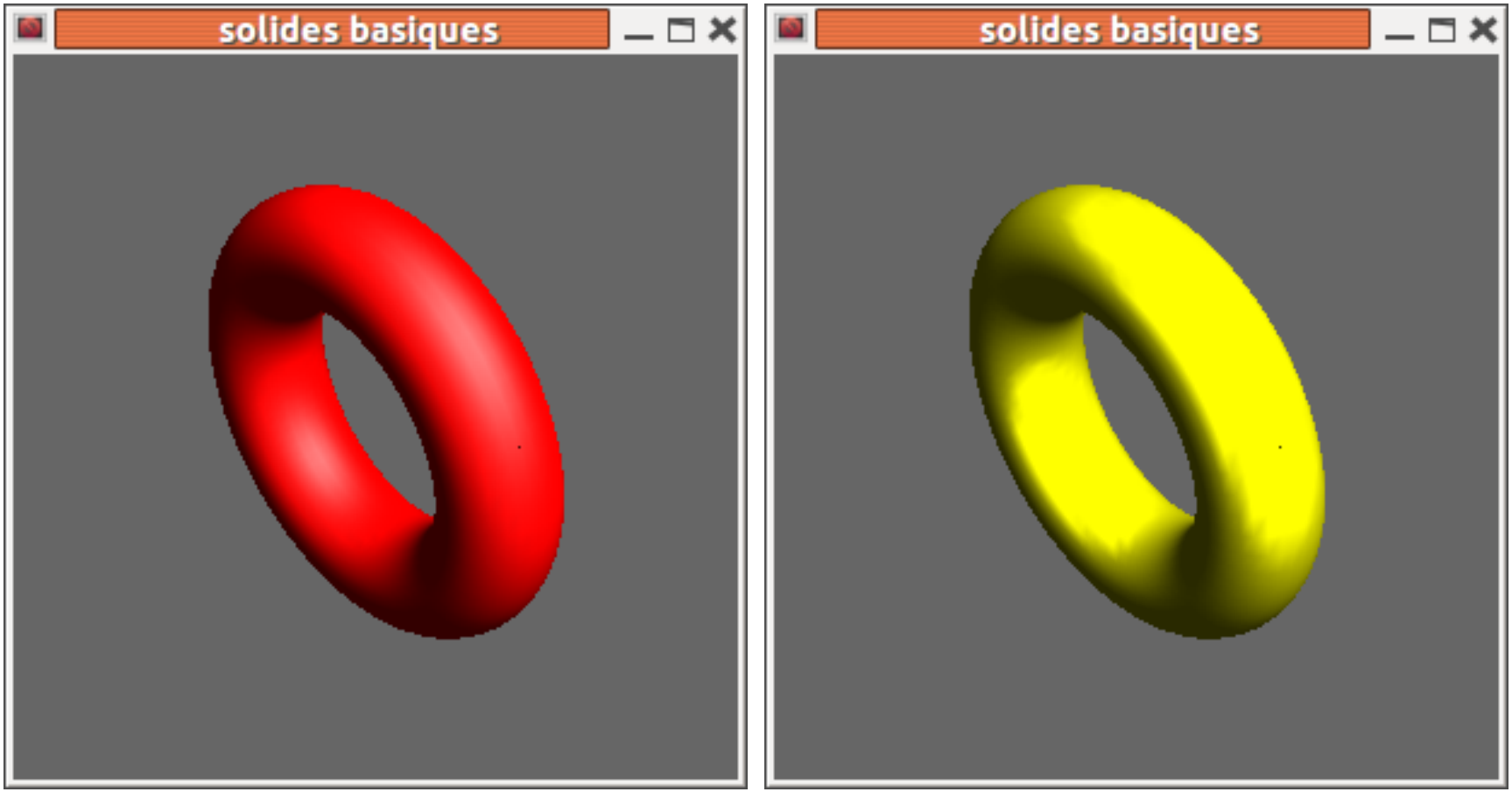
1. définition des vecteurs normaux de chaque sommet des objets,
2. création, sélection et position d'une ou de plusieurs sources lumineuses,
3. création et sélection d'un modèle de lumière, qui définit le niveau de la contribution ambiante et la position effective du point de vue (local ou à l'infini),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

Exemple de scène (3)

light()

```
def light():
    red = (1, 0, 0, 0)
    white = (1.0, 1.0, 1.0, 1.0)
    diffuse = (0.7, 0.7, 0.7, 1.0)
    specular = (0.001, 0.001, 0.001, 1.0)
    pos = (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

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

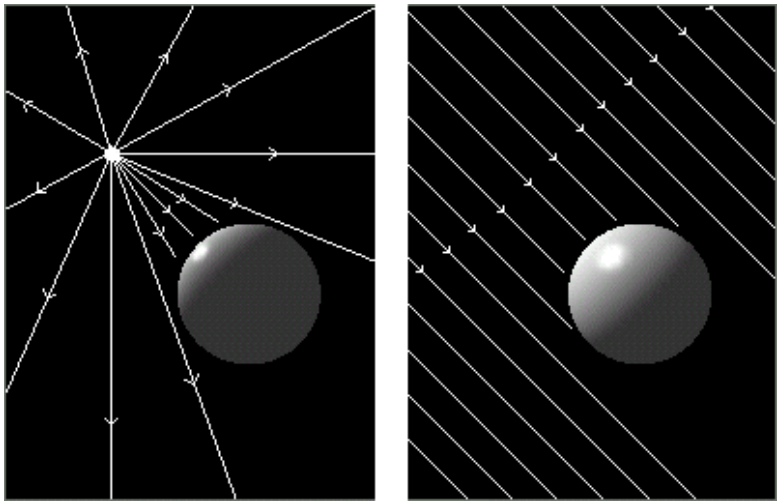
Activation d'*une* source : `glEnable(GL_LIGHT`$_i$`)` avec $i = 0 \ldots 7$.

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

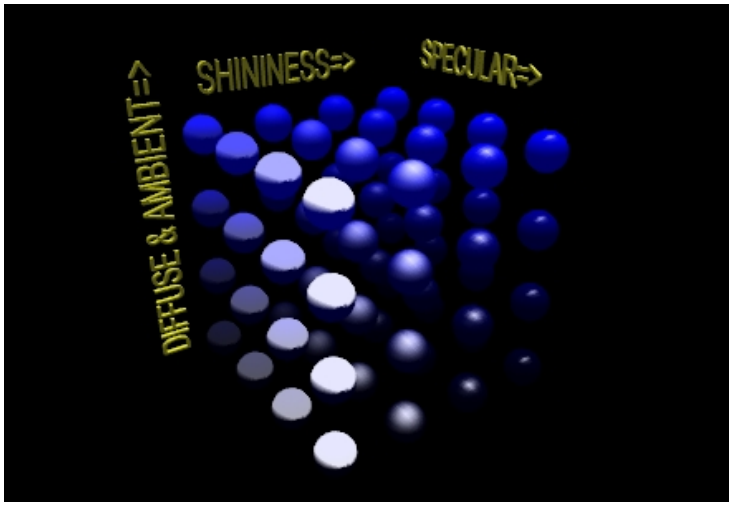
- GL_AMBIENT(0,0,0,1),
- GL_DIFFUSE(1,1,1,1),
- GL_SPECULAR(1,1,1,1),
- GL_POSITION(0,0,1,0).

Création des sources lumineuses

Lumière directionnelle ou positionnelle



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



Mouvement d'une lumière

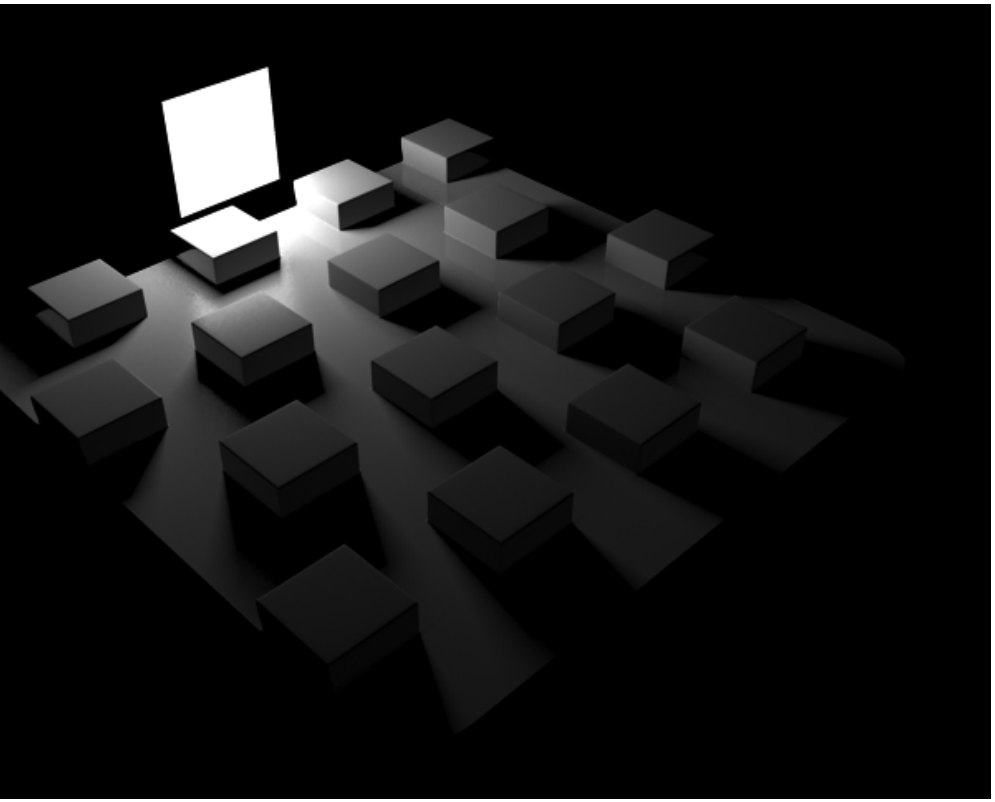
Exemple de lumière comme objet

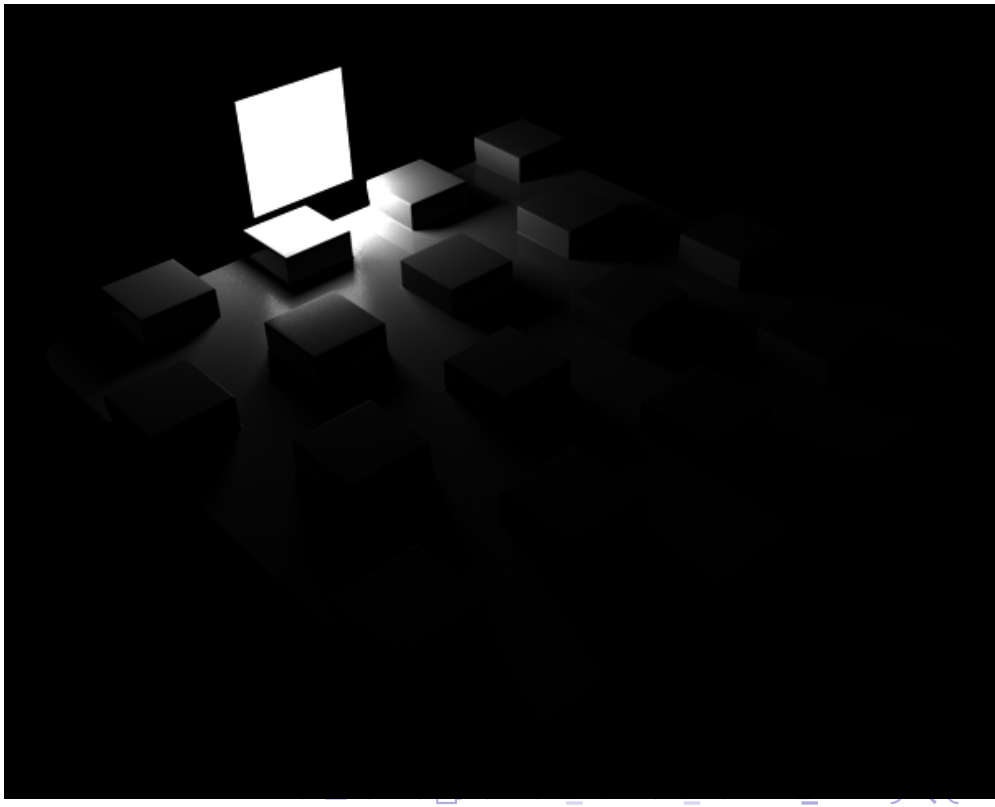
```
def display():
  light_position = (0.0, 0.0, 1.5, 1.0)

  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 :

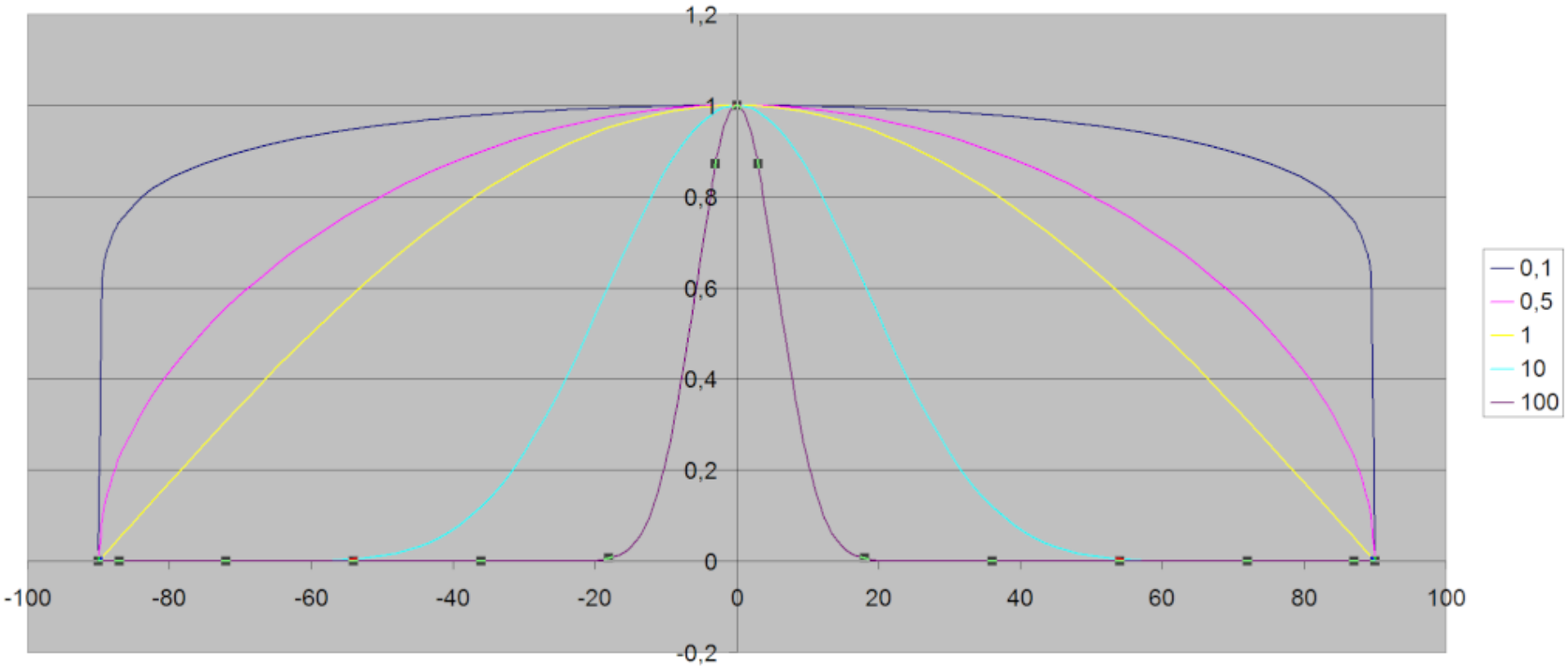
- de couleurs : `GL_AMBIENT`, `GL_DIFFUSE`, `GL_SPECULAR`,
- atténuation (avec position) : $1/(k_c + k_l.d + k_q.d^2)$
  `GL_CONSTANT_ATTENUATION` (k_c),
  `GL_LINEAR_ATTENUATION` (k_l),
  `GL_QUADRATIC_ATTENUATION` (k_q).





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*)

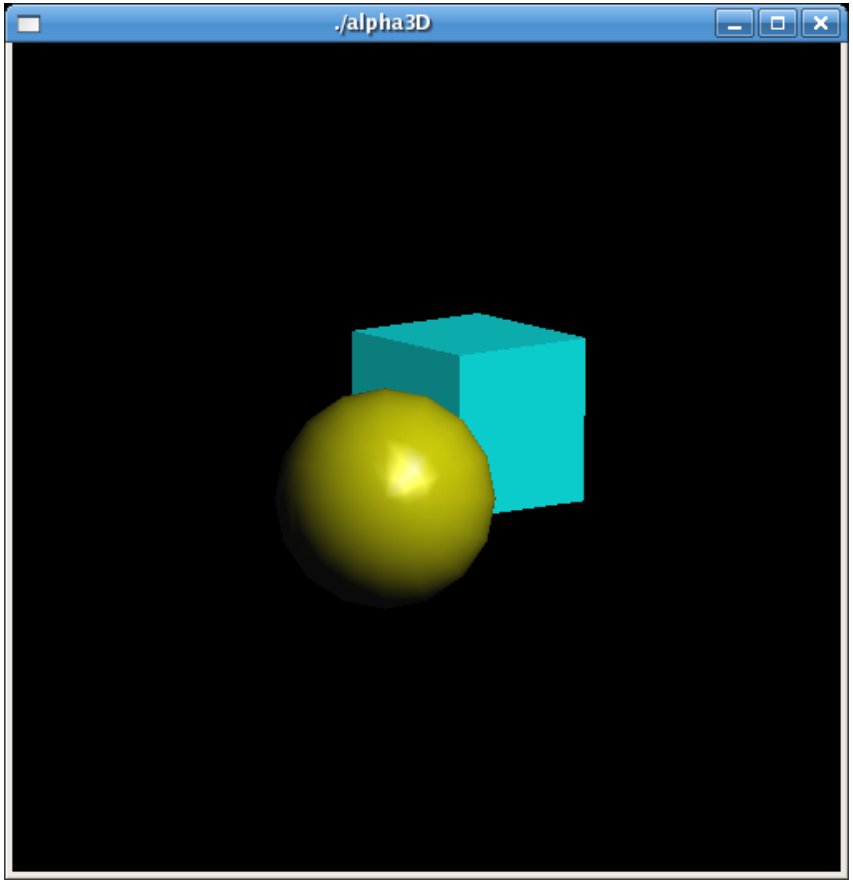
- `glEnable(GL_BLEND)`
- calcul de la couleur finale :
  $(s_r.R_s + d_r.R_d,\ s_d.V_s + d_v.V_d,\ s_b.B_s + d_b.B_d,\ s_a.A_s + d_a.A_d)$,
- le calcul des *facteurs* est défini par la fonction `glBlendFunc(`F_s`,` F_d`)`,

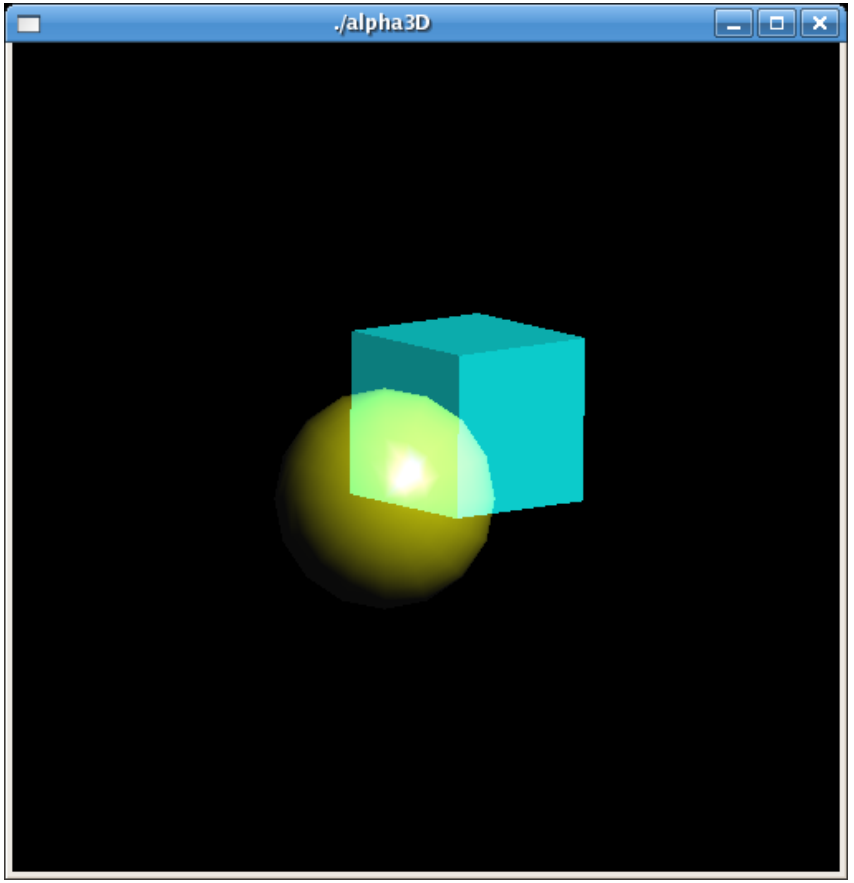
Exemple en 2D

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

- 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





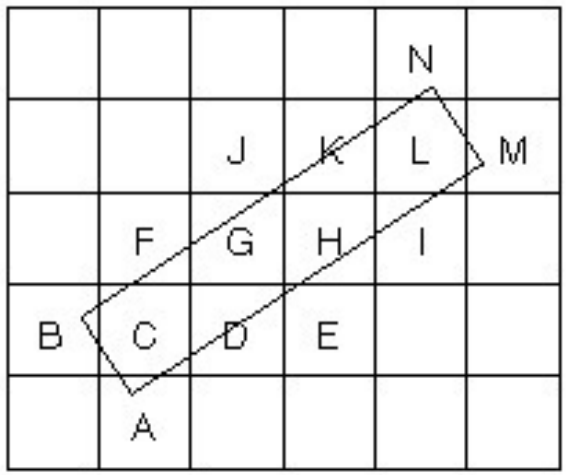
Blending : application à la transparence

Exemple d'une sphère transparente

```
def display():
   mat_solid = (0.75, 0.75, 0.0, 1.0)
   mat_zero = (0.0, 0.0, 0.0, 1.0)
   mat_transparent = (0.0, 0.8, 0.8, 0.6)
   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()
   ...
```

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 .759952
H .759952
I .141435
J .007639
K .434258
L .878469
M .040510
N .040510

Antialiasing

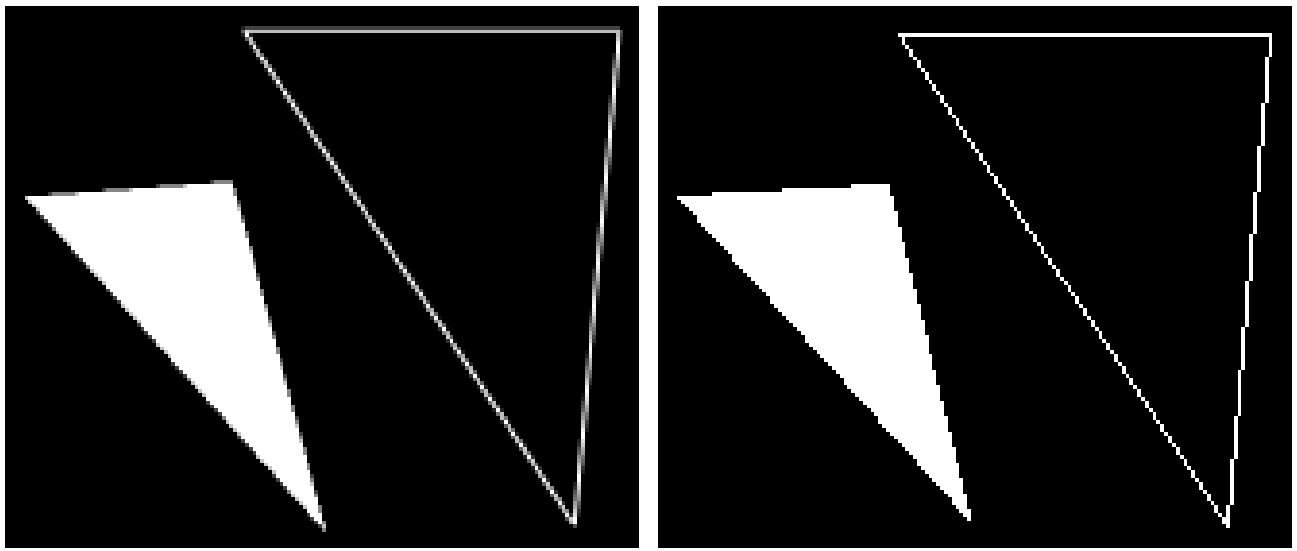
Détermination de la qualité du rendu : `glHint(cible, hint)` avec :

- 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.

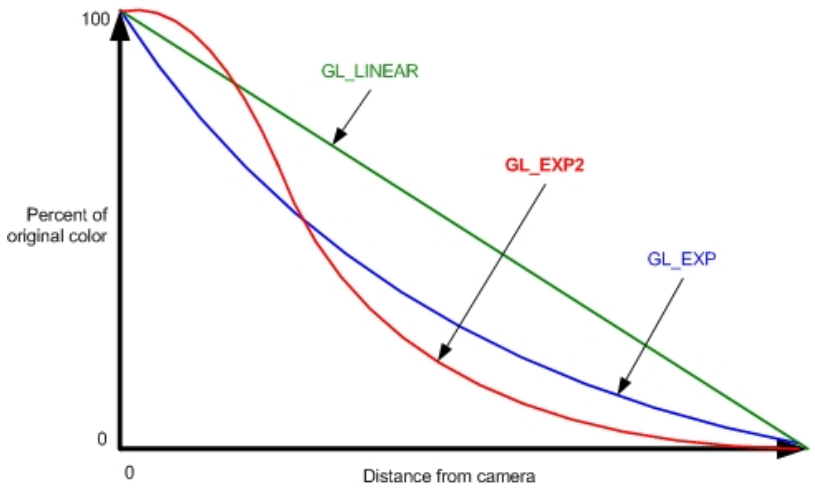
Antialiasing des points et des droites

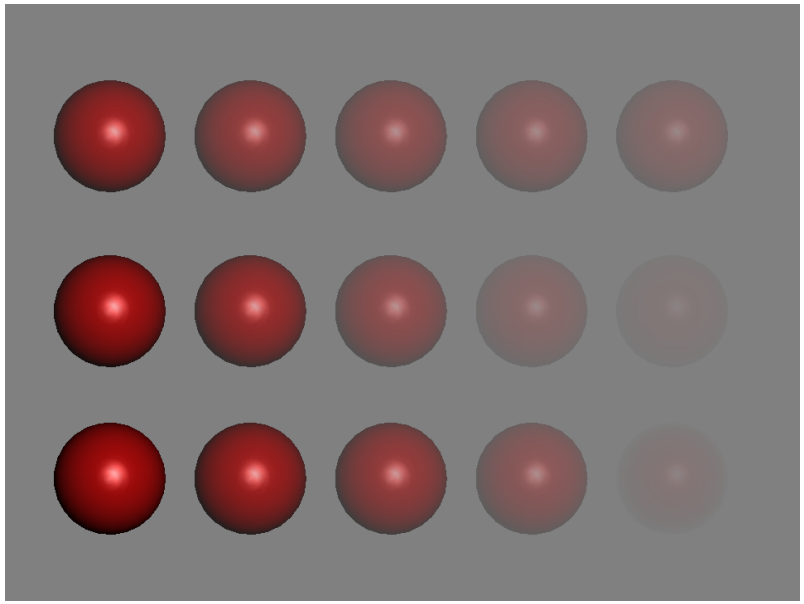
1. `glEnable(GL_POINT_SMOOTH)` ou `glEnable(GL_LINE_SMOOTH)`,
2. `glEnable(GL_BLEND)`,
3. `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.

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 associer chaque texture à un ID :

- `glGenTextures(n)`, génère et retourne le nombre n d'ID demandé,
- `glBindTexture(GL_TEXTURE_2D, id)`, associe l'id aux spécifications de la texture courante.

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

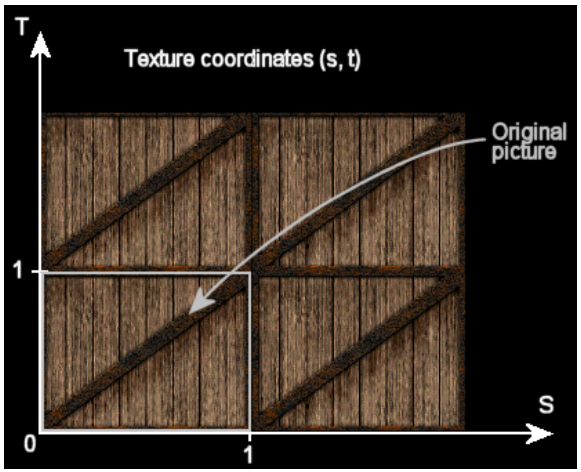
- `glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, la, ha, bordure, format, type, data)` avec
  - 0 : niveau de mipmap (manuel),
  - la (2^m), ha (2^n), bordure (0),
  - format : `GL_ALPHA`, `GL_RGB`, `GL_RGBA`, `GL_LUMINANCE` ou `GL_LUMINANCE_ALPHA`,
  - type : `GL_UNSIGNED_BYTE` (ou `SHORT`).

Définition d'une texture

Etape 3 répétition et filtrage appliqués lors du placage sur la primitive :

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

1. `(...WRAP_S, GL_REPEAT)` et `(...WRAP_T, GL_REPEAT)`.
2. `(...WRAP_S, GL_REPEAT)` et `(...WRAP_T, GL_CLAMP)`.



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

Définition d'une texture (suite)

Etape 4 définition de son influence sur chaque pixel :

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

- mode DECAL (le texel remplace le pixel), attention : aucun calcul de luminosité,
- MODULATE (la couleur du fragment est transformée par $\times$),
- BLEND (mélange simple)

Etape 5 autoriser le placage de texture :
`glEnable(GL_TEXTURE_2D)`

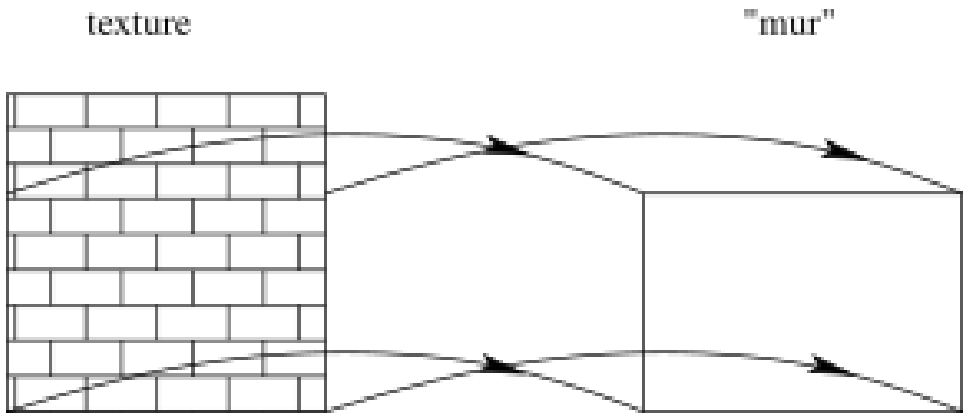
Etape 6 construire une scène, en spécifiant les coordonnées géométriques de texture.

Exemple

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

Exemple d'application d'une texture

```
tarray = (0, 0, 1, 0, 1, 2/3, 0, 2/3)
varray = (-1, 0, 1, 0, -1, 3, 1, 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).

Le Stencil Buffer (SB)

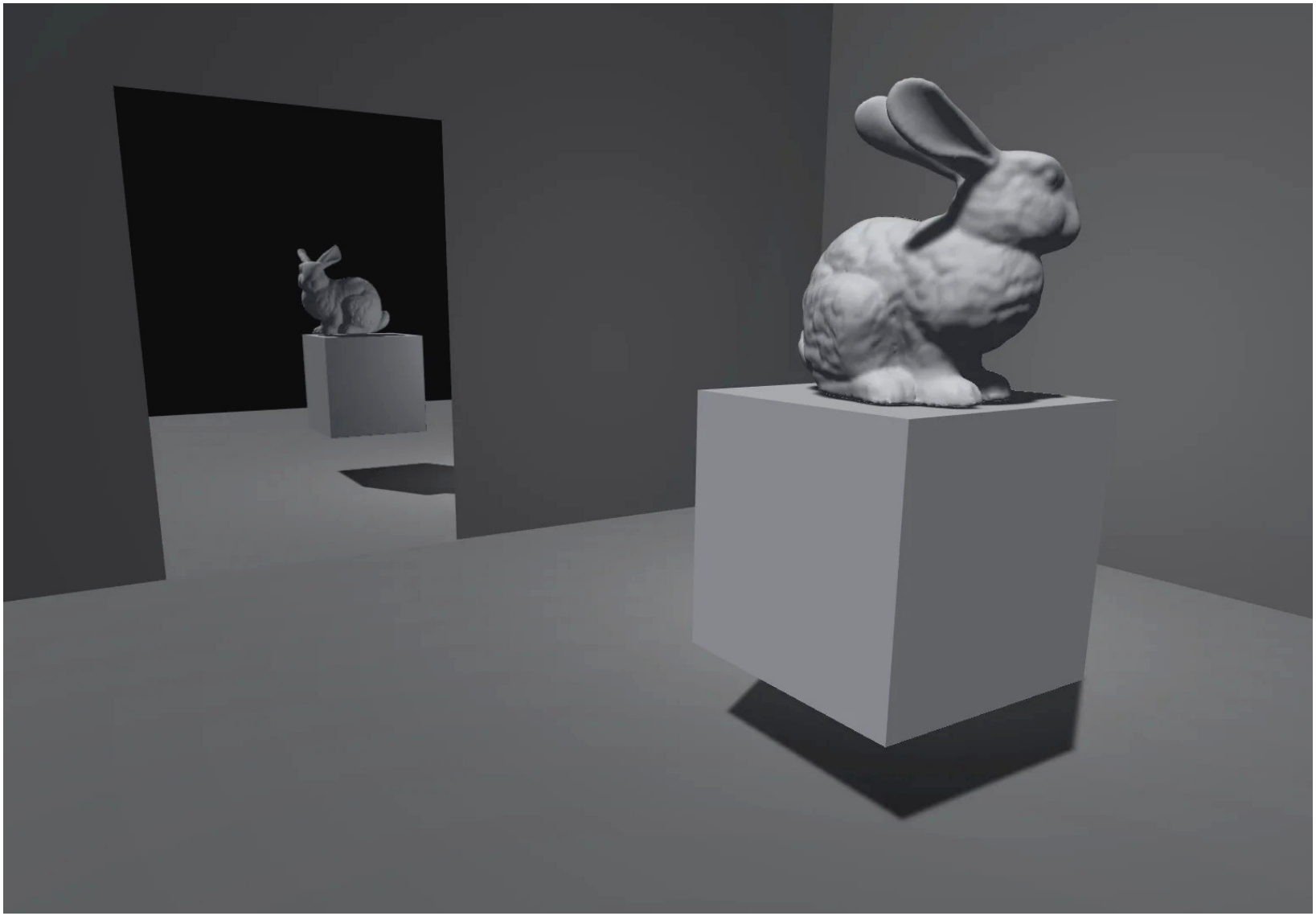
Ombre portée

1. Détermination des zones d'ombres portées dans le SB.
2. Premier rendu sans SB en lumière atténuée.
3. Deuxième rendu avec SB en lumière normale, par dessus le premier rendu sauf dans les zones identifiées par le SB.

Réflexion

1. Rendu de l'ensemble de la scène.
2. Pour chaque zone réflexive :
    1. Initialisation du SB pour cette zone.
    2. Symétrie de la scène par rapport à cette zone réflexive.
    3. Nouveau rendu de la scène avec le SB activé.

Le Stencil Buffer (SB)



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*.