Threading, Controleur et 3D

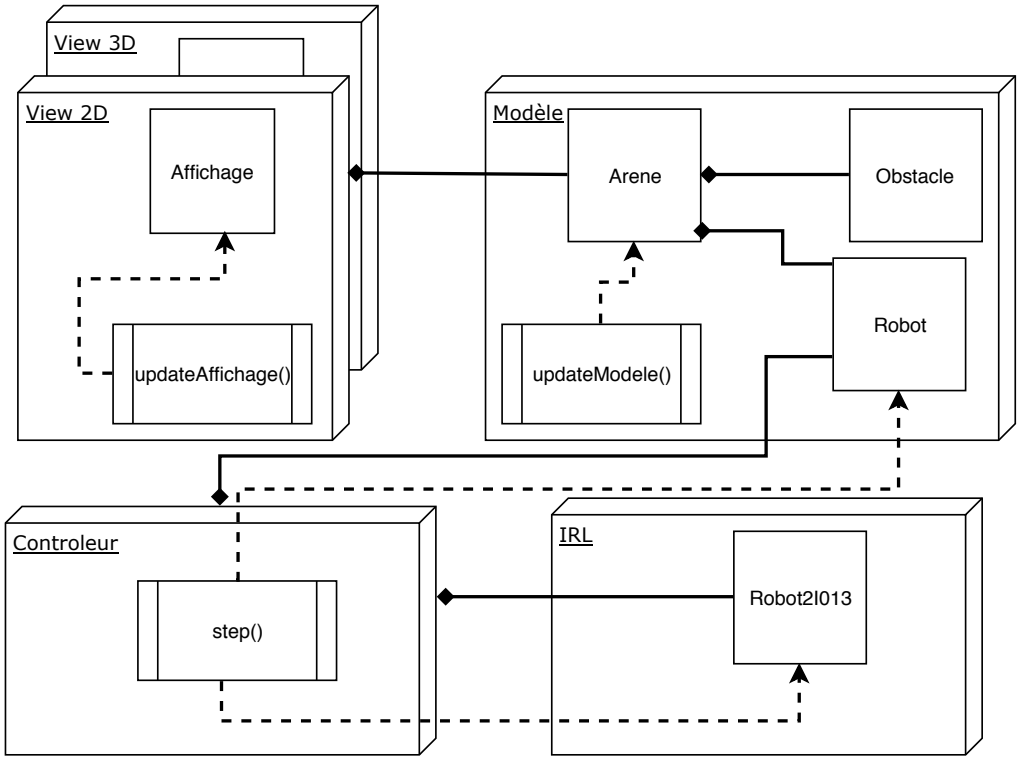
Nicolas Baskiotis
nicolas.baskiotis@lip6.fr

équipe MLIA, Laboratoire d'Informatique de Paris 6 (LIP6)
Sorbonne Université

S2 (2018-2019)

Plan

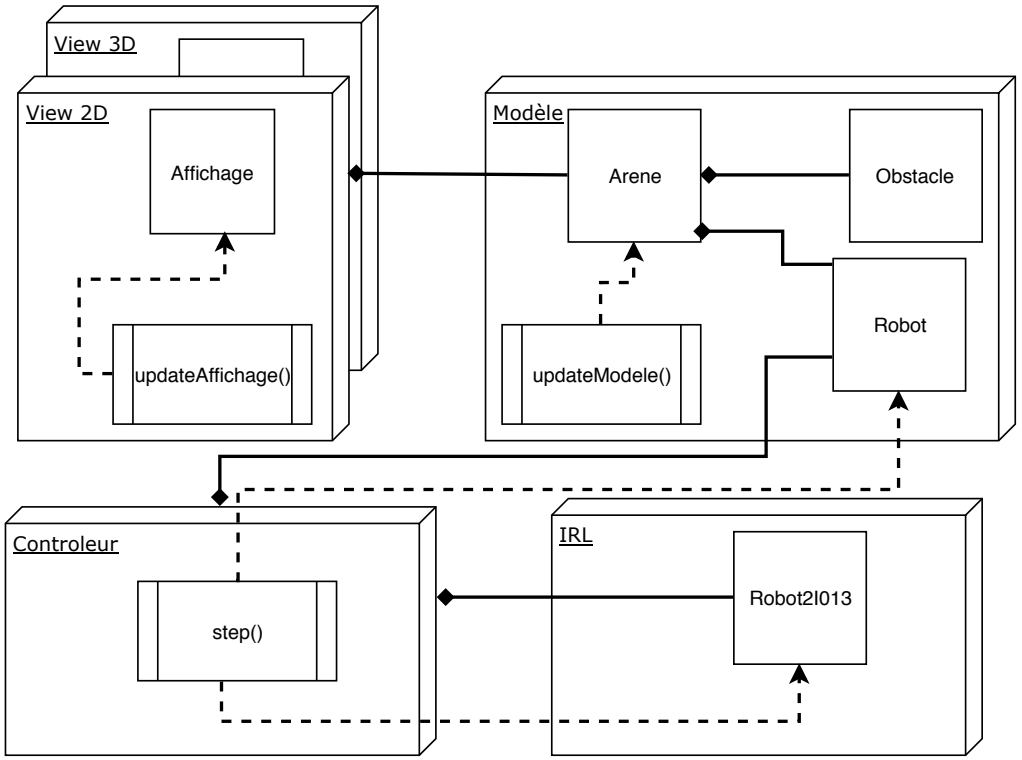
Etat des lieux



Modules indépendants

- Vue : un `update` pour maj de l'affichage
- Modèle : un `update` pour la maj du monde virtuel
- Contrôleur : un `step` pour les ordres au robot
- IRL : rien . . .

Etat des lieux



Contraintes

- `updateModele` (au moins) plus souvent que `step`
- `updateAffichage` (au moins) plus souvent que `step`
- un script (presque) commun à l'IRL et à la simulation

Organisation code (exemple)

```
arene = None
affichage = None
robot = None
from monprojet import Controleur
try:
  from robot2I013 import Robot2I013 as Robot
  robot = Robot()
except ImportError:
  from monprojet import MonRobot as Robot
  from monprojet import Arene,affichage
  robot = Robot()
  arene = Arene(robot)
  affichage = Affichage(arene)

....
ctrl = Controleur(...)
```

Solution naïve

```
def runCtrl(ctrl,fps=100):
    while True:
      ctrl.step()
      if arene is not None:
        arene.update()
      if affichage is not None:
        affichage.update()
      time.sleep(1./fps)
```

Mais . . .

- les `update` ne sont pas indépendants
- mélange dans la boucle de IRL et simulé, pas idéal

Solution : Thread

Thread

- un thread = bout de code qui s'exécute en parallèle
- Attention : délicat dans le cas général (programmation concurrente) :
  - Cas de variables partagées entre thread
  - Cas des ressources partagées (fichiers, connections)
- pour faire un thread : soit appel de la classe `Thread` avec en paramètre `target` la fonction à exécuter; soit héritage de la classe `Thread` et redéfinition de `run`.

```
from threading import Thread

class Affichage:
  ....
  def boucle(self,fps):
    while True:
      self.update()
      time.sleep(1./fps)
affichage = Affichage(...)
threadAff = Thread(target=affichage.boucle,args=(fps,))
thread.start()
```

Solution : Thread

Thread

- un thread = bout de code qui s'exécute en parallèle
- Attention : délicat dans le cas général (programmation concurrente) :
  - Cas de variables partagées entre thread
  - Cas des ressources partagées (fichiers, connections)
- pour faire un thread : soit appel de la classe `Thread` avec en paramètre `target` la fonction à exécuter; soit héritage de la classe `Thread` et redéfinition de `run`.

```
from threading import Thread

class Affichage(Thread):
  def __init__(self,...):
    super(Affichage,self).__init__()
  def run(self,fps):
    while True:
      self.update()
      time.sleep(1./fps)
affichage = Affichage(...)
affichage.start()
```

Quelques autres objets pour les threads

- la méthode `join` d'un thread permet d'attendre la fin du thread
- la méthode `setDaemon(true)` permet de rendre le thread indépendant de la fin du programme principal
- l'objet `RLock` permet de synchroniser les threads :

```
lock = RLock()
    ...
#dans affichage
def updateAffichage():
  with lock: #bloque execution des autres threads qui testent lock
    dessine(robot)

#dans modele
def updateModele():
  with lock:  #est bloque qd updateAffichage s'execute
      updatePosition(robot)
```

Plan

Retour sur les Design patterns

Problèmes :

- vous voulez parler ”simplement” à votre robot (*avance*, *tourne*)
- votre robot ne comprend que `set_motor_dps`
- Capteurs bruités : `get_distance` pas précis

Solutions

- Adapter : quand vous avez deux APIs qui n'ont pas les mêmes noms de méthode mais font plus ou moins la même chose
- Decorator : ajouter des fonctionalités à un objet
- Facade : rendre plus simple les appels à un/plusieurs sous-sytèmes
- Bridge : proche de l'adapter, mais rend abstrait l'implémentation de la ”traduction”; permet de jongler entre plusieurs implémentations.

Solution Adapter

```
class Adapter:
  def __init__(self,robot):
    self.robot = robot
  def forward(self,speed):
    self.robot.set_motor_dps(self.robot.MOTOR_LEFT+\
            self.robot.MOTOR_RIGHT,speed)
  def turnRight(self,speed):
    self.set_motor_dps(self.robot.MOTOR_LEFT,speed)
  def get_distance(self):
    return self.robot.get_distance()
....
```

Problèmes :

Solution Adapter

```
class Adapter:
  def __init__(self,robot):
    self.robot = robot
  def forward(self,speed):
    self.robot.set_motor_dps(self.robot.MOTOR_LEFT+\
            self.robot.MOTOR_RIGHT,speed)
  def turnRight(self,speed):
    self.set_motor_dps(self.robot.MOTOR_LEFT,speed)
  def get_distance(self):
    return self.robot.get_distance()
....
```

Problèmes :

- Pas flexible
- Pas possible de tester diverses implémentations
- Toujours même façon d'avancer quelque soit l'action

Décorateur

Vous avez envie de rajouter des fonctionalités :

- Pouvoir logger les actions du robot
- Lisser les résultats de `get_distance()`
- Avoir une mémoire

Solution possible : décorateur

```
class Lisser:
  def __init__(self,obj,size=5):
    self._obj = obj
    self.hist =[0]*size
    self.cpt = 0
  def __getattr__(self,name):
    self.hist[self.cpt]=self._robot.get_distance()]
    self.cpt= (self.cpt+1) %len(self.hist)
    return getattr(self._robot,name)
  def get_distance(self):
      return sum(self.hist)/len(self.hist)
```

Décorateur

Vous avez envie de rajouter des fonctionalités :

- Pouvoir logger les actions du robot
- Lisser les résultats de `get_distance()`
- Avoir une mémoire

Solution possible : décorateur

```
class LogAction:
  def __init__(self,obj):
    self._obj = obj
    self.log =[]
  def __getattr__(self,name):
    if name in ["get_distance","set_motor_dps"]:
      self.log.append(name)
    return getattr(self._robot,name)
```

Avantage : on peut mixer les décorateurs :

```
monobjet = LogAction(Lisser(objet))
```

Stratégie

Principe: découper les actions de bases en petits blocs

- Aller tout droit sur une certaine distance
- Tourner d'un certain angle
- Approcher un point

Puis imbriquer les stratégies entre elles : une meta-stratégie

Nécessite :

- initialiser une stratégie
- savoir quand elle est finie
- savoir où en est (asynchrone !!!)

Stratégie

Mauvaise solution

```
class StrategieDroit:
  def __init__(self,distance,...):
    self.distance = distance
  def step(self):
    parcouru = 0
    while parcouru<self.distance:
      self.avancer()
```

PAS ASYNCHRONE !!!

Bonne solution

```
class StrategieDroit:
  def __init__(self,distance,...):
    self.distance = distance
  def start(self):
    self.parcouru = 0
  def step(self):
    self.parcouru += ...
    if self.stop(): return
    self.avancer()
  def stop(self):
    return self.parcouru>self.distance
```

De même pour l'angle . . .

Mixer les stratégies

Stratégie conditonnelle

```
def step(self):
    if loin: self.avanceVite.step()
    else: self.avanceLentement.step()
```

Stratégie séquentielle

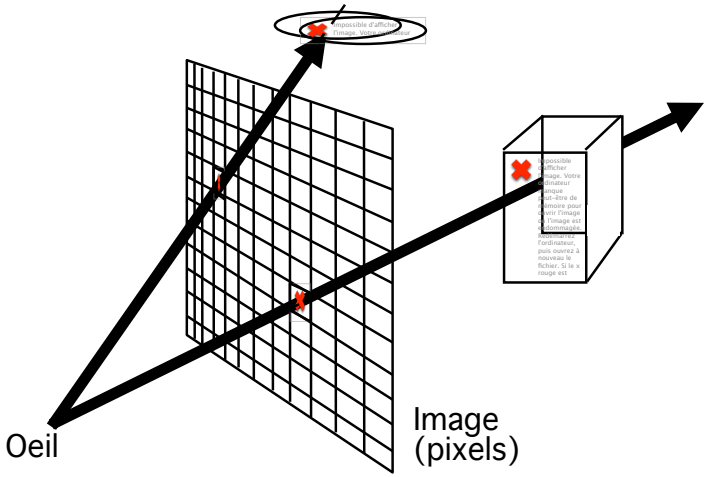
```
def __init__(self):
  self.strats = [stratDroit, stratTourne,stratDroit,stratTourne]
  self.cur= -1
def start():
  self.cur = -1
def step(self):
  if self.stop():return
  if self.cur <0 or self.strats[self.cur].stop():
    self.cur+=1
    self.strats[self.cur].start()
  self.strats[self.cur].step()
def stop(self):
  return self.cur== len(self.strats)-1\
      and self.strats[self.cur].stop()
```

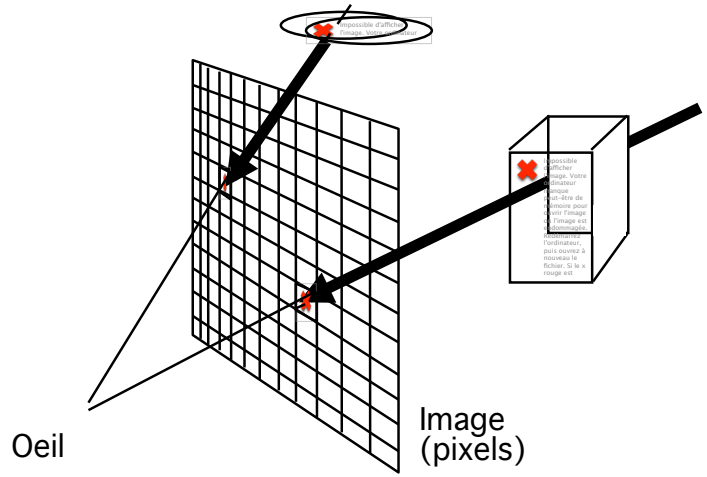
Plan

Quelques explications sur le fonctionnement

- Slides tirés de V. Guigue (inspirés également du cours de A. Meyer, Lyon 1)
- Beaucoup de calcul matriciel (d'où les nouveaux usages des cartes graphiques)

2 approches duales en SI





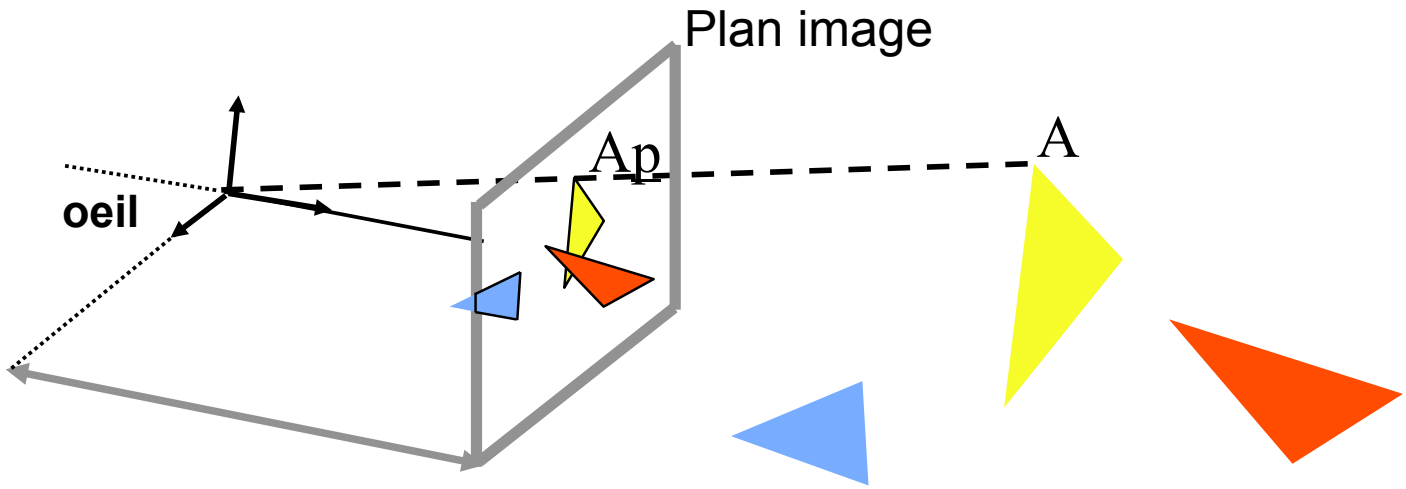
Des rayons sont lancés depuis l'œil vers la scène en passant par un pixel

Ray-tracing
- Image réaliste
- Lent

Les objets sont projetés sur l'écran dans la direction de l'œil.

Rendu projectif (cablé sur les cartes graphiques modernes -> temps réel)

Rendu projectif : PIPELINE

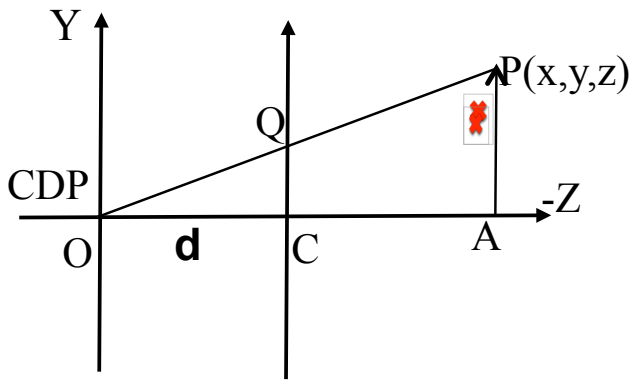


Pipeline

1. Clipping des polygones en 3D suivant la pyramide de vue
2. Projection des points sur le plan image
3. Remplissage des triangles (Rasterizing) dans l'image
   a. Suppression des parties cachées : Z-Buffer
   b. Calcul de la couleur : illumination

Projection perspective

- Besoin de perspective
- Configuration simple :



Plan image

$$\frac{CQ}{AP} = \frac{CO}{AO} \Leftrightarrow CQ = y' = \frac{y.d}{z}$$

$$x' = \frac{x.d}{z}$$

Si $d = 1 \Rightarrow y' = \frac{y}{z}$ et $x' = \frac{x}{z}$

d=distance focale

Matrice de projection : $M_{I \leftarrow C}$

$$M_{I \leftarrow C} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix}$$

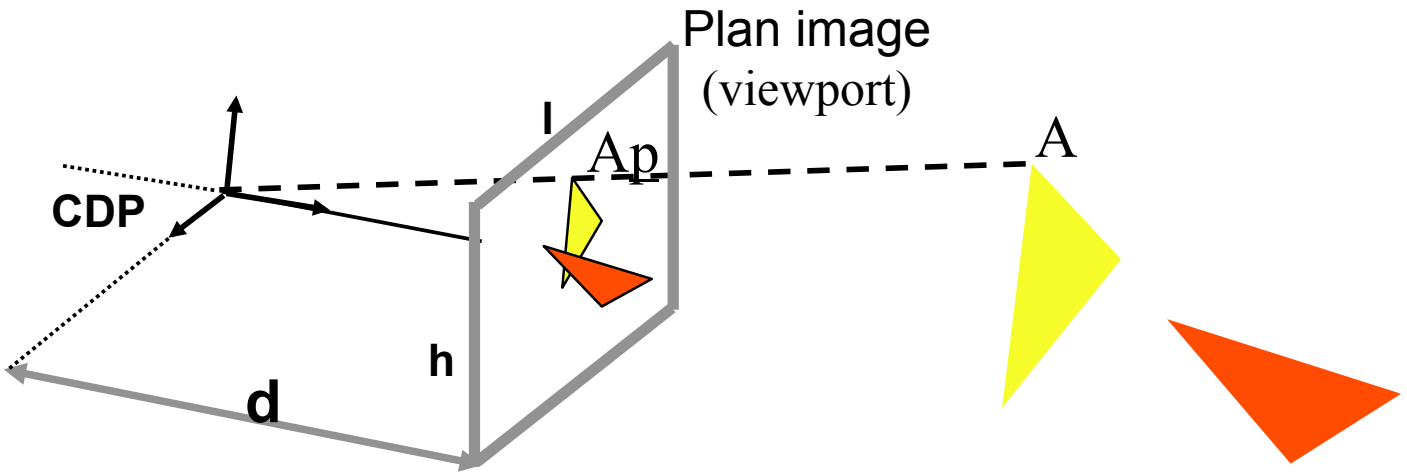
- Soit un point dans l'espace de la camera

$$(x, y, z, 1)$$

- Résultat : un point dans l'espace Image

$$(x, y, z, z/d) = \left(\frac{xd}{z}, \frac{yd}{z}, d, 1 \right)$$

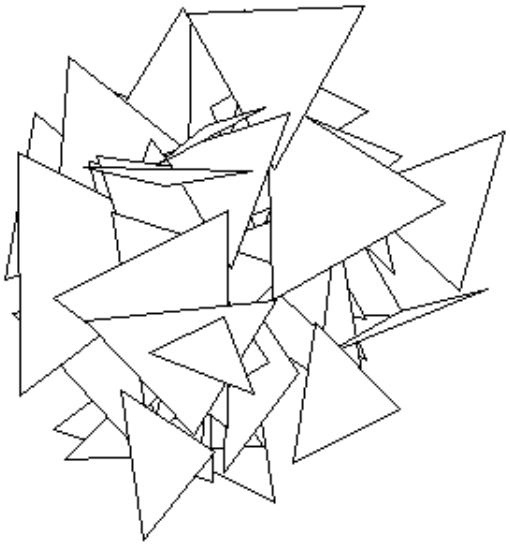
Rendu projectif

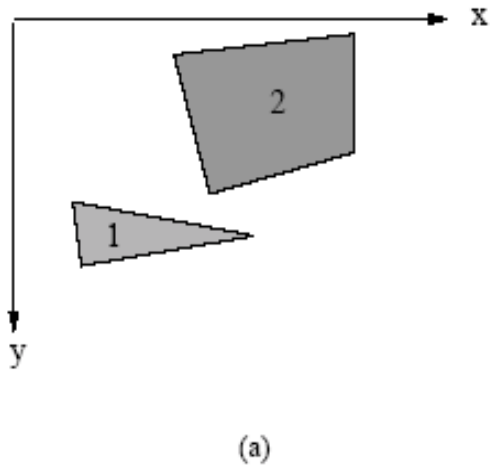


1. Projection des points sur le plan image
2. Clipping
3. Remplissage des triangles (rasterisation) dans l'image
5. Suppression des parties cachées
6. Calcul de la couleur : illumination

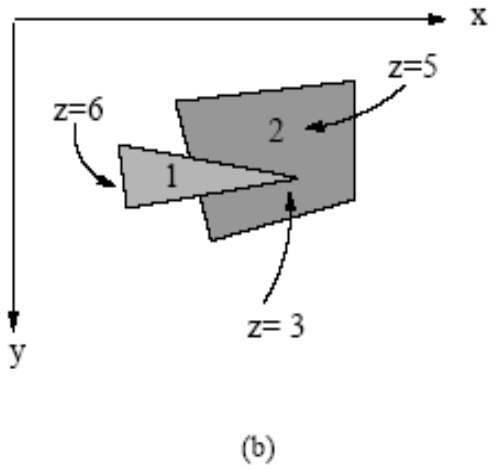
1ere idée : algo du peintre

Ambiguïtés

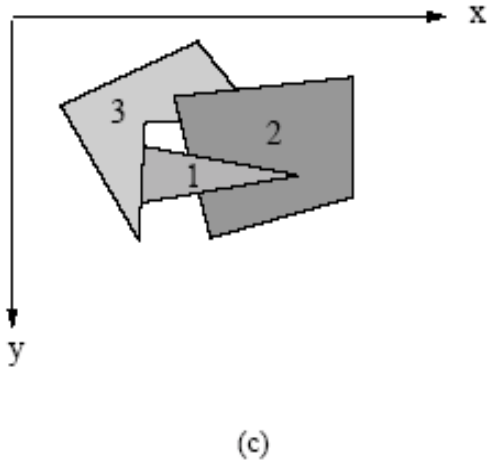




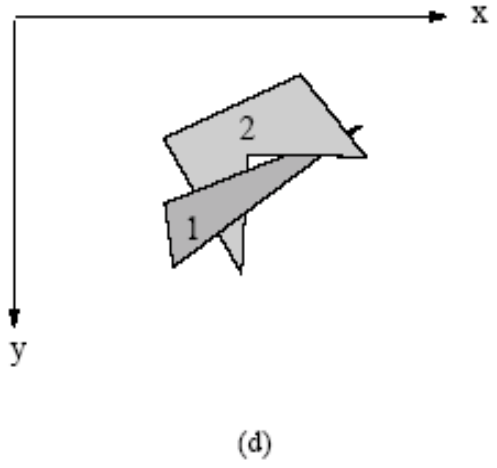
(a)



(b)

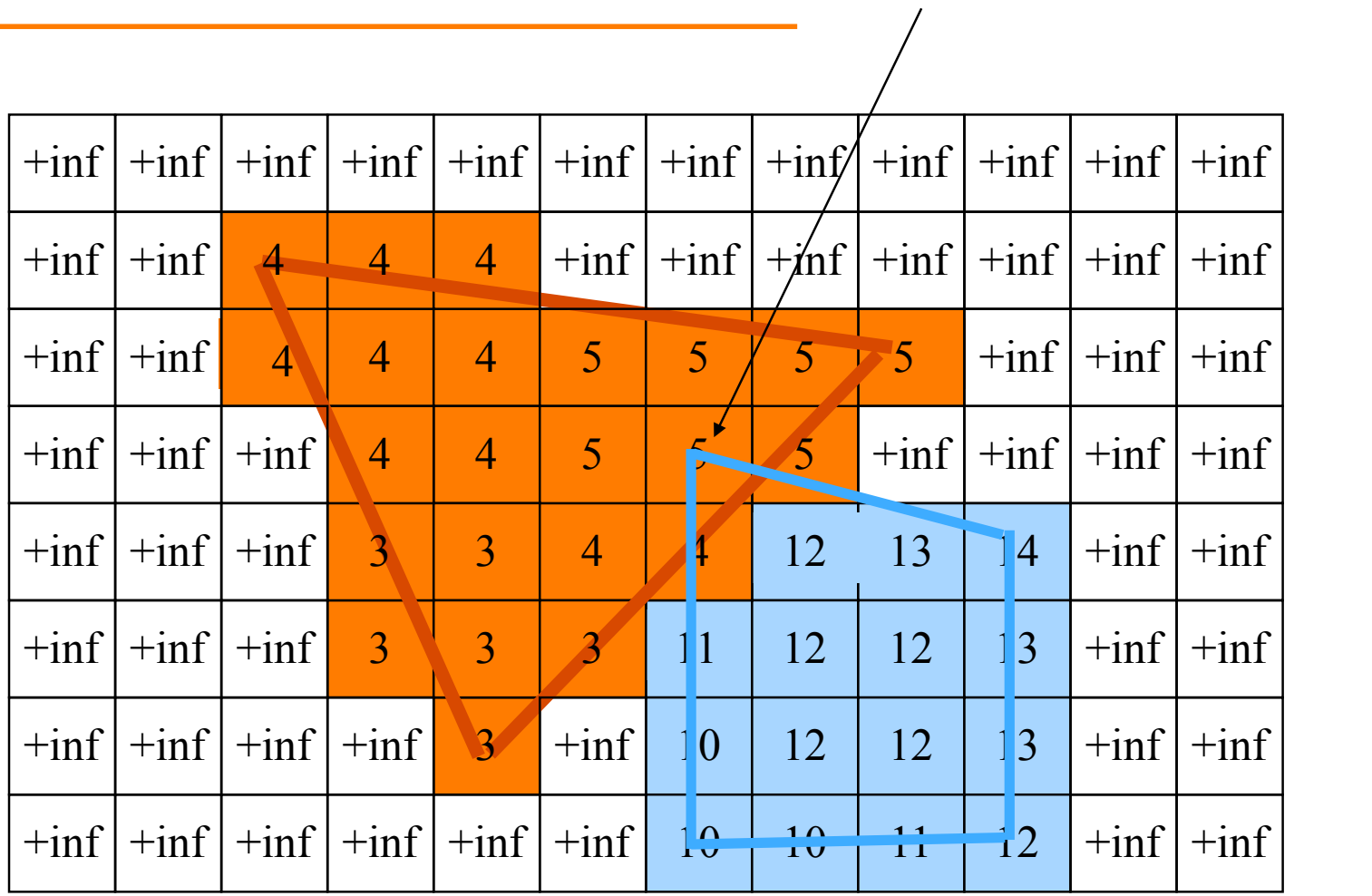


(c)



(d)

Z-Buffer


z=11>5 donc caché
+inf +inf +inf +inf +inf +inf +inf +inf +inf +inf +inf +inf
+inf +inf 4 4 4 +inf +inf +inf +inf +inf +inf +inf
+inf +inf 4 4 4 5 5 5 5 +inf +inf +inf
+inf +inf +inf 4 4 5 5 5 +inf +inf +inf +inf
+inf +inf +inf 3 3 4 4 12 13 14 +inf +inf
+inf +inf +inf 3 3 3 11 12 12 13 +inf +inf
+inf +inf +inf +inf 3 +inf 10 12 12 13 +inf +inf
+inf +inf +inf +inf +inf +inf 10 10 11 12 +inf +inf

OpenGL : par exemple

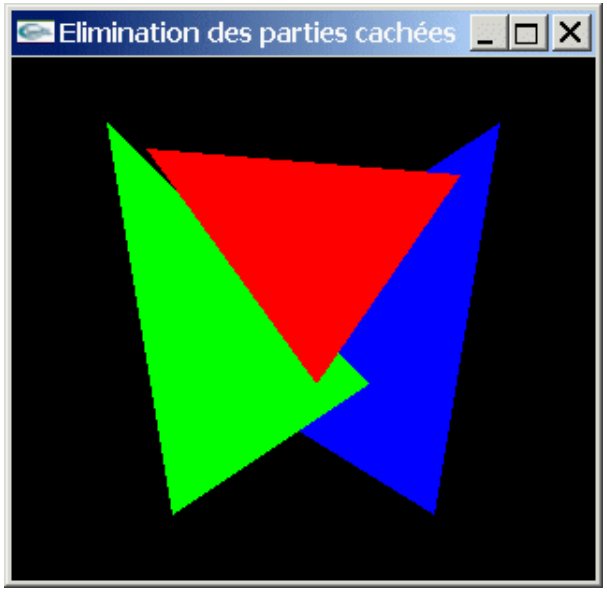
- Effacer le buffer et le zbuffer entre chaque image

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
```

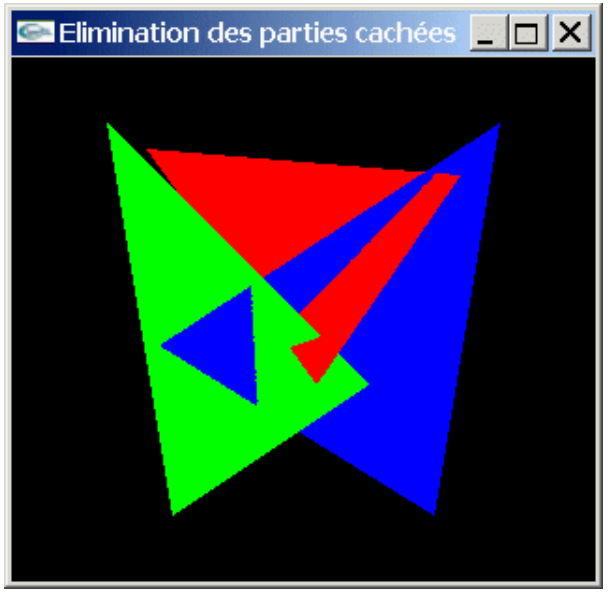
- Active le test des Z avec le Z-buffer

```
glEnable(GL_DEPTH_TEST) ;
```

non activé



activé



Python et 3D

- On propose d'utiliser pyglet
  - `pip install paquet --user --proxy=proxy.ufr-info-p6.jussieu.fr:3128`
- La puissance de l'architecture vient du fait qu'il n'y a pas grand chose à modifier pour faire de la 3D...

Gestion d'une fenêtre

```
import pyglet
from OpenGL.GL import glLight
from pyglet.gl import *
from pyglet.window import key
from OpenGL.GLUT import *
from pyglet.image.codecs.png import PNGImageDecoder
```

Par extension d'un objet existant:

```
class Window(pyglet.window.Window): # syntaxe de l'heritage
    xRotation = yRotation = 0
    increment = 5
    toDraw = []

    def __init__(self, width, height, title=''):
        super(Window, self).__init__(width, height, title)
        self.setup()
```

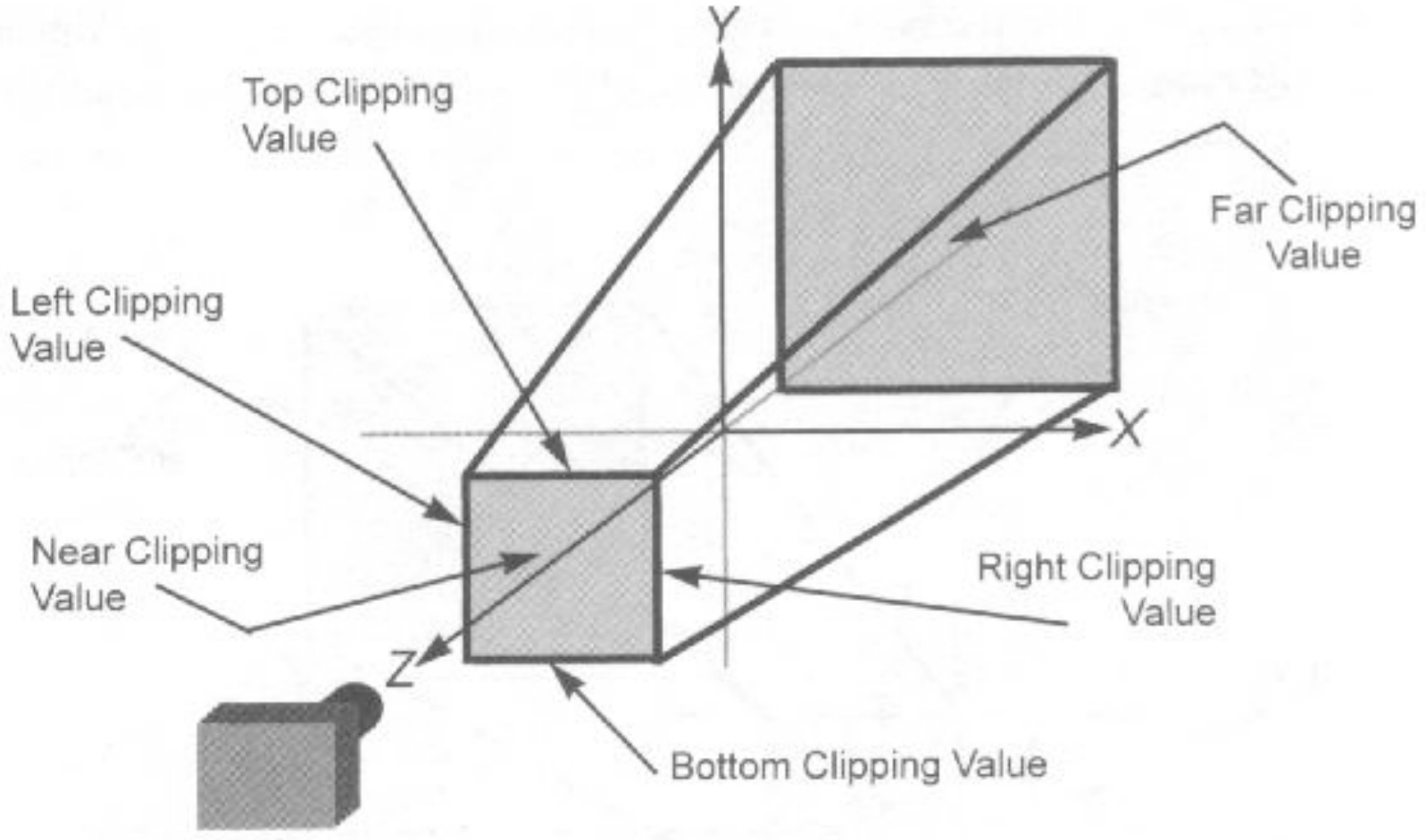
Setup

Réglages à vérifier:

- Tests de profondeur
- Définition de la scène

```
class Window(pyglet.window.Window): # syntaxe de l'heritage
    ...
    def setup(self):
        # One-time GL setup
        glClearColor(1, 1, 1, 1)
        glColor3f(1, 0, 0)
        glEnable(GL_DEPTH_TEST)
        # using Projection mode
        glViewport(0, 0, super(Window, self).width*2, \
              super(Window, self).height*2)  # taille de la scene
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        # perspective
        aspectRatio = super(Window, self).width / \
              super(Window, self).height
        gluPerspective(35*self.zoom, aspectRatio, 1, 1000)
        #
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
```

Définition de la fenêtre

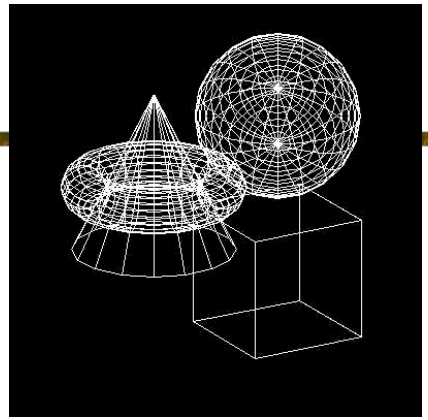


Option avancée

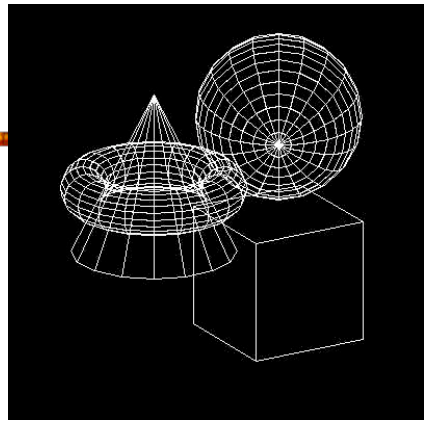
- Lumière, textures...

```
def setup_light(self):
    # Simple light setup.  On Windows GL_LIGHT0 is enabled by defa
    # but this is not the case on Linux or Mac, so remember to alw
    # include it.
    glEnable(GL_LIGHTING)
    glEnable(GL_LIGHT0)
    glEnable(GL_LIGHT1)

    # Define a simple function to create ctypes arrays of floats:
    def vec(*args):
        return (GLfloat * len(args))(*args)

    glLightfv(GL_LIGHT0, GL_POSITION, vec(.5, .5, 1, 0))
    glLightfv(GL_LIGHT0, GL_SPECULAR, vec(.5, .5, 1, 1))
    glLightfv(GL_LIGHT0, GL_DIFFUSE, vec(1, 1, 1, 1))
    glLightfv(GL_LIGHT1, GL_POSITION, vec(1, 0, .5, 0))
    glLightfv(GL_LIGHT1, GL_DIFFUSE, vec(.5, .5, .5, 1))
    glLightfv(GL_LIGHT1, GL_SPECULAR, vec(1, 1, 1, 1))

    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, \
        vec(0.5, 0.5, 0.5, 1))
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, vec(1, 1, 1, 1))
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 50)
```

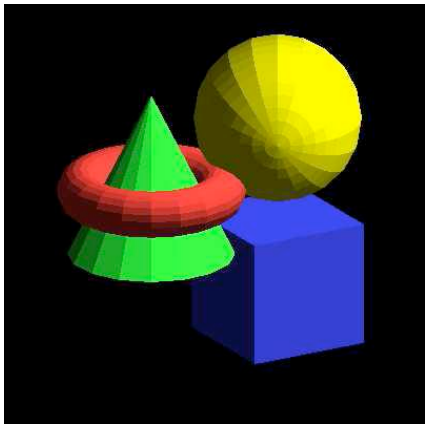
Modes de rendu



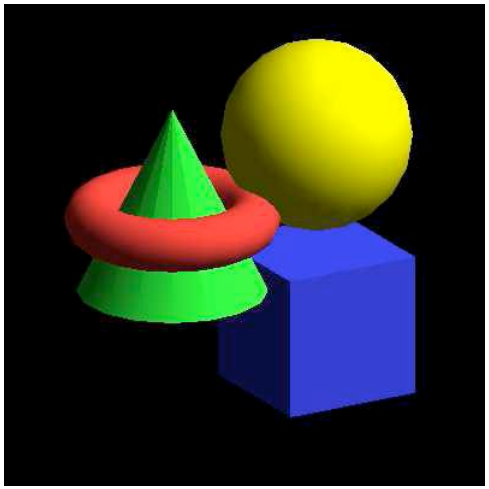
Fil de fer



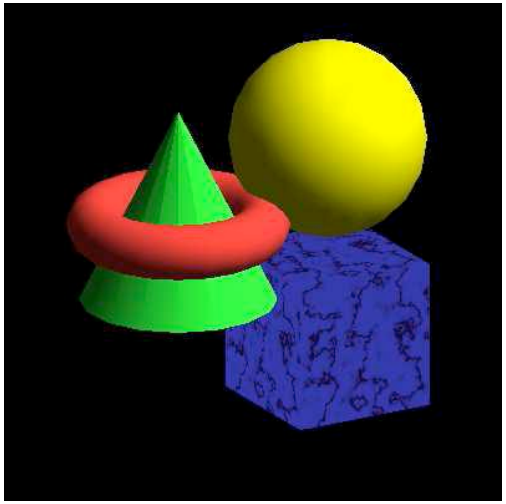
Faces cachées (objet)



Rendu Gouraud



Rendu Phong



Texture

Historique

Méthode d'affichage

- Conserver la structure générale
- Déporter l'affichage dans des éléments de base codés par ailleurs

```
def on_draw(self):
    # Clear the current GL Window
    self.clear()
    self.set_camera() # cf plus tard
    # Push Matrix onto stack
    glPushMatrix()

    glRotatef(self.xRotation, 1, 0, 0)
    glRotatef(self.yRotation, 0, 1, 0)

    for c in self.toDraw:
        # Draw the six sides of the cube
        c.draw()

    # Pop Matrix off stack
    glPopMatrix()
```

Dessinons !

```
class Coord():
    def __init__(self,x,y,z,r,g,b):
        self.x=x
        self.y=y
        self.z=z
        self.r = r
        self.g = g
        self.b = b
        self.cote = 20

    def draw(self):
        glBegin(GL_QUADS)
        glColor3ub(self.r, self.g, self.b) # couleur
        glVertex3f(self.x, self.y, self.z) # point 1
        glVertex3f(self.x+self.cote, self.y, self.z) # point 2
        glVertex3f(self.x+self.cote, self.y+self.cote, self.z) # ...
        glVertex3f(self.x, self.y+self.cote, self.z)
        glEnd()

if __name__ == '__main__':
    WINDOW = 400
    w= Window(WINDOW, WINDOW, 'Pyglet Colored Cube')
    w.toDraw += [ Coord(0,0,0, 100, 0, 0)]
    w.toDraw += [Coord(0, 40, 0, 0, 255, 0)]
```

Interaction clavier

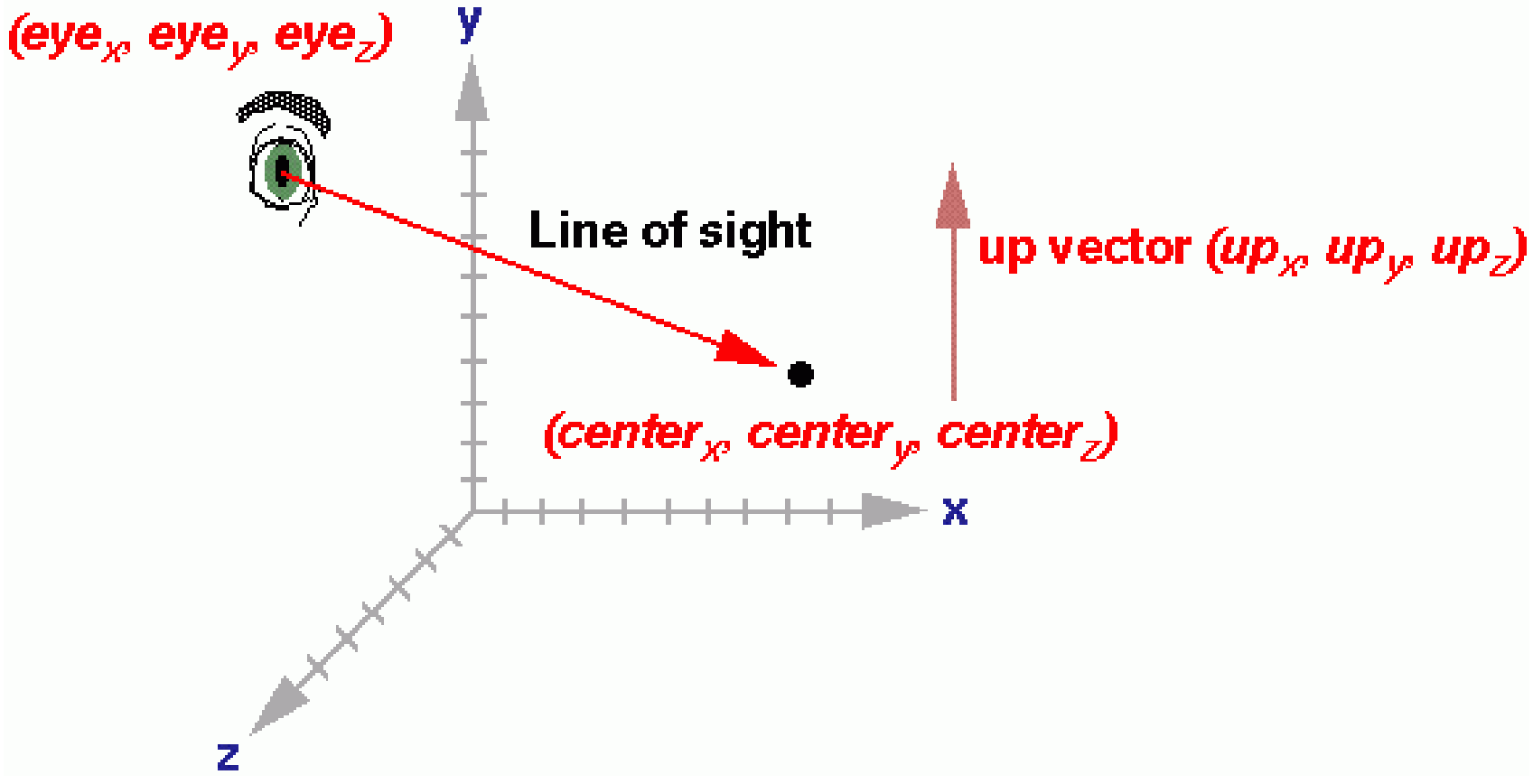
Gestion déjà prévue dans la fenêtre

```
def on_text_motion(self, motion):
    if motion == key.UP:
        self.xRotation -= self.increment
    elif motion == key.DOWN:
        self.xRotation += self.increment
    elif motion == key.LEFT:
        self.yRotation -= self.increment
    elif motion == key.RIGHT:
        self.yRotation += self.increment

def on_text(self, text):
    print(self.zoom)
    if text.find('z')>-1:
        self.zoom *= 0.75
    elif text.find('Z')>-1:
        self.zoom *= 1.15
    elif text.find('i')>-1:
        pyglet.image.get_buffer_manager().get_color_buffer().\
           save('screenshot.png')
```

- Dernière ligne plus importante !!!

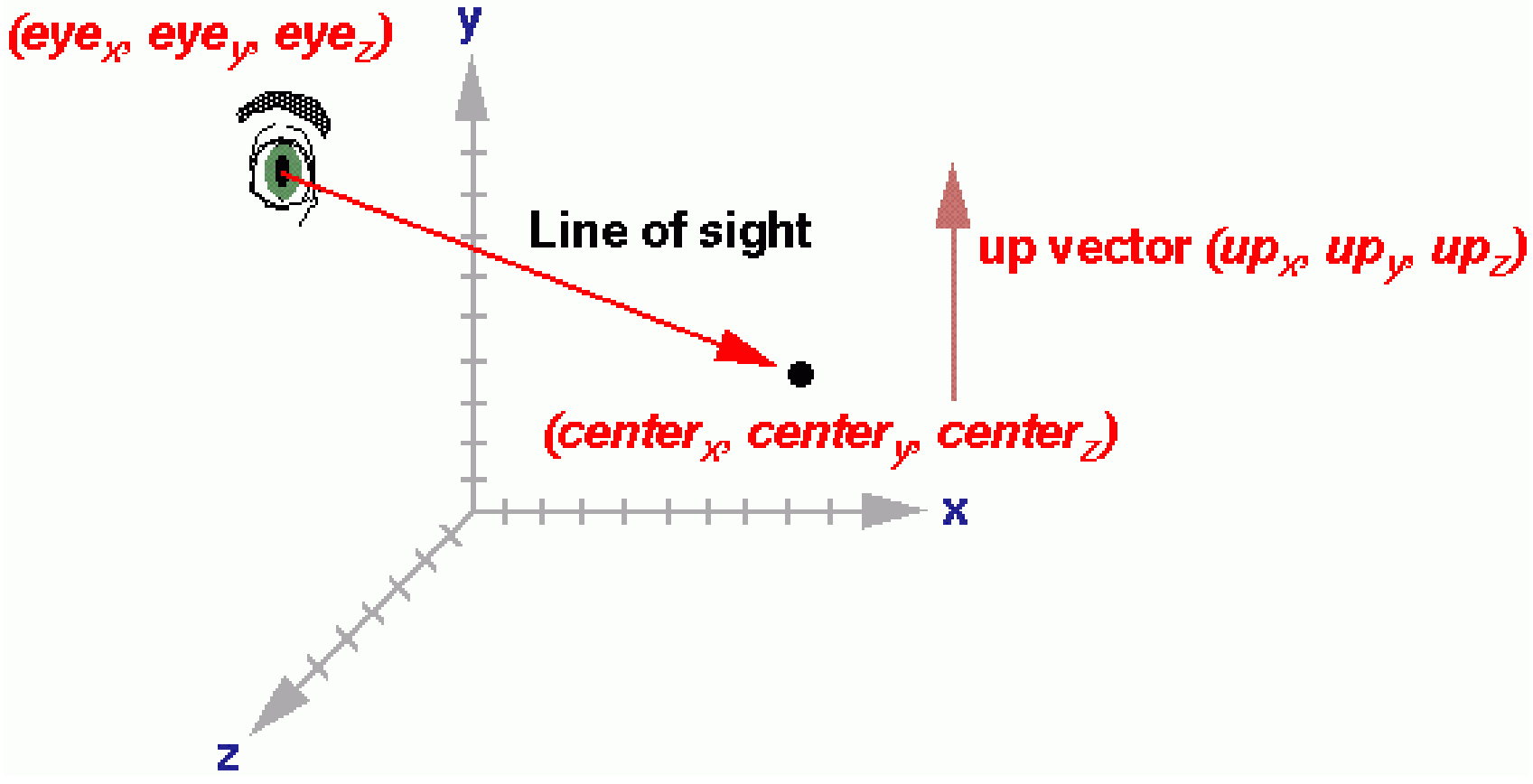
Fonctionnement de la caméra:

```
GLvoid gluLookAt( GLdouble eyex, GLdouble eyey,
                  GLdouble eyez, GLdouble centerx, GLdouble
                  centery, GLdouble centerz, GLdouble upx,
                  GLdouble upy, GLdouble upz )
```



Fonctionnement de la caméra:

```
GLvoid gluLookAt( GLdouble eyex, GLdouble eyey,
                  GLdouble eyez, GLdouble centerx, GLdouble
                  centery, GLdouble centerz, GLdouble upx,
                  GLdouble upy, GLdouble upz )
```



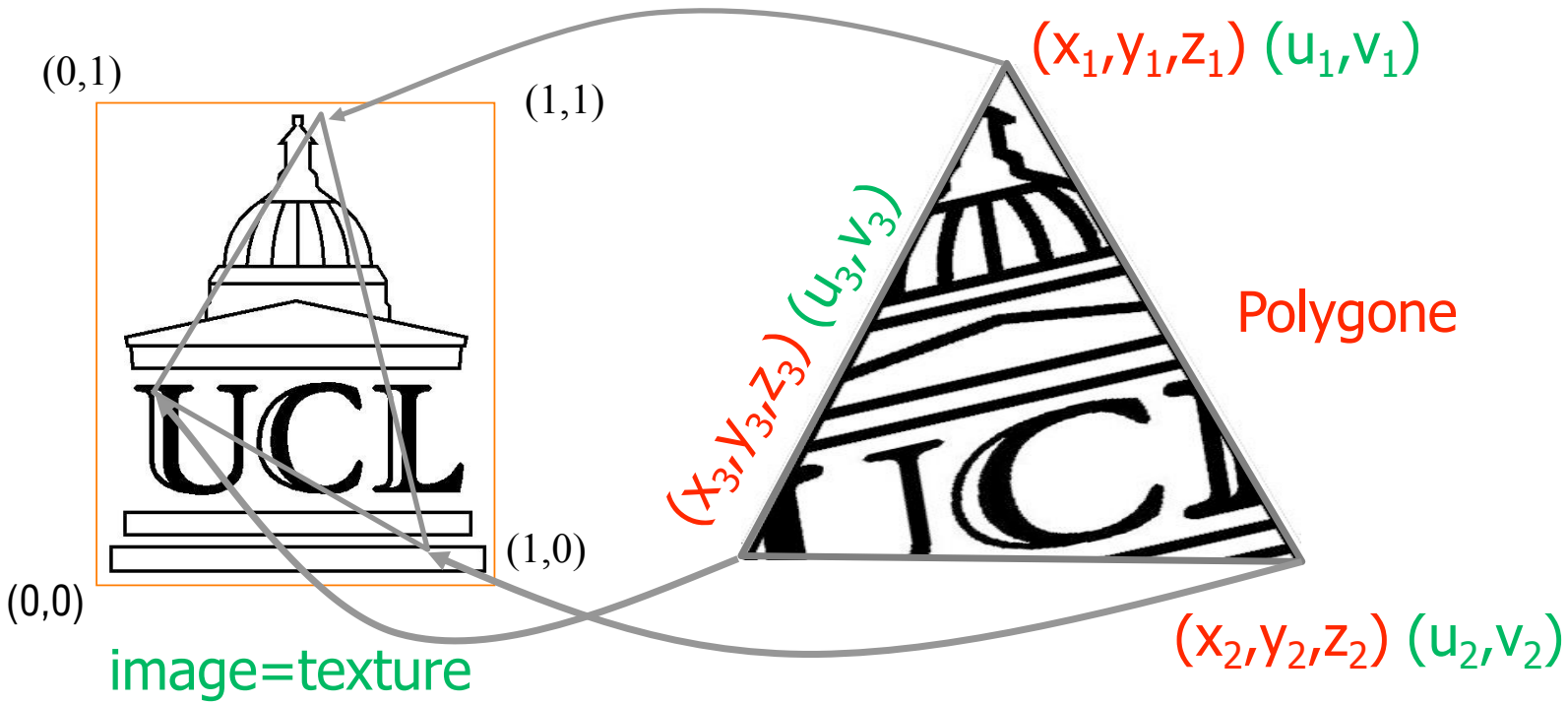
Mise en oeuvre de la caméra

```
    def set_camera(self):
        # using Projection mode
        glViewport(0, 0, super(Window, self).width*2, super(Window, se
# taille de la scene
        glMatrixMode(GL_PROJECTION)
        glLoadIdentity()
        # perspective
        aspectRatio = super(Window, self).width / super(Window, self).
        gluPerspective(35*self.zoom, aspectRatio, 1, 1000)
        #
        glMatrixMode(GL_MODELVIEW)
        glLoadIdentity()
        gluLookAt(-200, -200, 0, 200, 200, -100, 0, 0, 1)
```

Textures

Pour des rendus plus agréables et pour le terrain, il faut gérer les images en plus de la 3D:

Plaquage de la texture



- A chaque sommet de la face
  - Coordonnées textures (u,v)

Texture: application sur le cube

- texture = mémoire graphique + possède un identifiant
- Texture = image carrée, dim. en puissance de 2

```
class CoordTex(Coord):
    def __init__(self,x,y,z,r,g,b, fname):
        Coord.__init__(self, x, y, z, r, g, b)
        im = pyglet.image.load(fname, decoder=PNGImageDecoder())
        self.texture = im.get_texture()

    def draw(self):
        glBindTexture(self.texture.target, self.texture.id)
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1)

        glBegin(GL_QUADS)
        glTexCoord2f(0,0)
        glVertex3f(self.x, self.y, self.z)
        glTexCoord2f(1, 0)
        glVertex3f(self.x+self.cote, self.y, self.z)
        glTexCoord2f(1, 1)
        glVertex3f(self.x+self.cote, self.y+self.cote, self.z)
        glTexCoord2f(0, 1)
        glVertex3f(self.x, self.y+self.cote, self.z)
        glEnd()
        glBindTexture(GL_TEXTURE_2D, 0)
```

Plan

Jouer avec les images

```
from PIL import Image
import random
import numpy as np

if __name__ == '__main__':
    # Image.load(filename)
    d = 512
    img = Image.new("RGB", (d, d), "white")
    for i in range(d):
        for j in range(d):
            img.putpixel((i,j), (0,255,0))

    for i in range(15000):
        img.putpixel((random.randint(0,d-1), random.randint(0,d-1)), \
          (random.randint(0,255), random.randint(0,255), random.randin

    for i in range(50):
        for j in range(50):
            img.putpixel((i+50,j+50), (255,0,0))

    img.save("matexture.png", "png")
    print(img.getpixel((10,10)))
    np.array(img) #transforme en tableau numpy
```

Résultat