OpenGL, OpenGL/ES, Introduction

Renaud BLANCH
Université Joseph Fourier
Laboratoire d'Informatique de Grenoble

http://iihm.imag.fr/blanch/
mailto:blanch@imag.fr

Plan

0. Introduction
1. Pipeline graphique fixe
2. Pipeline graphique programmable

Plan

0. Introduction
1. Pipeline graphique fixe
2. Pipeline graphique programmable

1. Pipeline graphique fixe

1.0 OpenGL : un état graphique partagé
1.1 Projection : du monde 3D à la fenêtre 2D
1.2 Modèle géométrique
1.3 Éclairement
1.4 Images et textures
1.5 Au-delà du mode direct

1. Pipeline graphique fixe

1.0 OpenGL : un état graphique partagé
1.1 Projection : du monde 3D à la fenêtre 2D
1.2 Modèle géométrique
1.3 Éclairement
1.4 Images et textures
1.5 Au-delà du mode direct

1.0 OpenGL : état graphique

L'**appel de** toute **fonction** d'OpenGL
n'est **valide** qu'une fois un **contexte de rendu**
mis en place.

Ce contexte contient l'**état courant**
du moteur graphique
(e.g., couleur du tracé, épaisseur des traits, …)

1.0 OpenGL : état graphique

Certains états sont modifiables
de manière générique :

`gl`*`Something`*`Mode(GL_`*`MODE`*`)`
`gl`**`Enable`**`/gl`**`Disable`**`(GL_`*`CAPABILITY`*`)`
`gl`**`Push`**`Attrib/gl`**`Pop`**`Attrib(GL_`*`CAPABILITY`*`_BIT)`

D'autres de manière spécifique :

`gl`**`Color`**, `gl`**`ClearColor`**, …

L'**état initial** est **spécifié** par le standard.

1.0 OpenGL : état graphique

On peut **accéder** à l'état courant :
(peu recommandé car coûteux)

`gl`**`Get`**`Type[v](GL_`*`NAME`*`, *param)`
`gl`**`IsEnabled`**`(GL_`*`CAPABILITY`*`)`

1.0 OpenGL : état graphique

Les modifications de l'état courant ont un coût,
il convient de les **grouper** par niveau de granularité :

- à l'**initialisation** (lumières, couleur de fond, ...) ;
- à chaque **redimensionnement** de vue (perspective, …) ;
- à chaque **réaffichage** (animation, …) ;
- pour chaque **objet**, **face**, **sommet**
  (couleur, texture, …)

Il peut parfois être utile de **sauver** avant modification,
puis de **restaurer** ensuite, **l'état antérieur**.

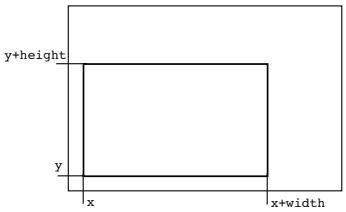
1. Pipeline graphique fixe

1.0 OpenGL : un état graphique partagé
1.1 Projection : du monde 3D à la fenêtre 2D
1.2 Modèle géométrique
1.3 Éclairement
1.4 Images et textures
1.5 Au-delà du mode direct

1.1 Projection : de la 3D à la 2D

Le ***viewport*** définit la zone rectangulaire 2D de l'écran dans laquelle le rendu va être fait.

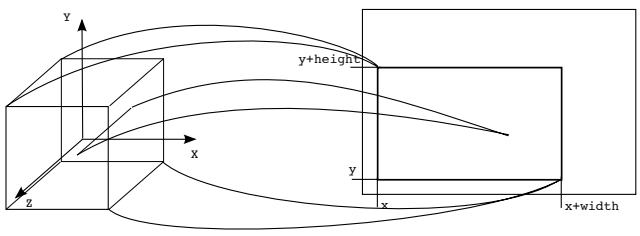
`gl`**`Viewport`**`(x, y, width, height)`
x, y, width, height sont des dimensions en **pixels**.



1.1 Projection : de la 3D à la 2D

La **projection** définit comment le monde 3D va être transformé pour être en partie affiché dans le *viewport*.

Par défaut, c'est le cube unité $[-1,1]^3$ qui est projeté sur le *viewport*.



1.1 Projection : de la 3D à la 2D

La **matrice de projection** modifie le repère du monde 3D.

Ici, une mise à l'échelle d'un facteur 1/2 va envoyer le cube $[-2,2]^3$ sur le *viewport* :

```
glMatrixMode(GL_PROJECTION)
glLoadIdentity()
glScalef(.5, .5, .5)
```

13

1.1 Projection : de la 3D à la 2D

Deux transformations très pratiques sont fournies :
- la projection **orthographique** ; et
- la projection en **perspective**.

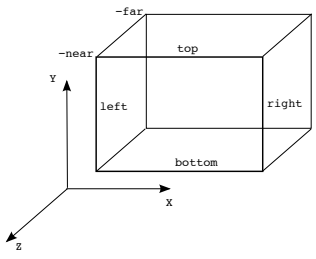
14

1.1 Projection : de la 3D à la 2D

Deux transformations très pratiques sont fournies :
- la projection **orthographique** ; et
- la projection en perspective.

```
glOrtho(left, right, bottom, top,
        near, far)
```

découpe un parallélépipède rectangle dans le monde 3D.



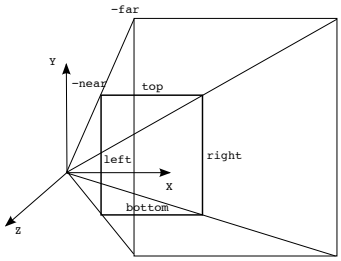
15

1.1 Projection : de la 3D à la 2D

Deux transformations très pratiques sont fournies :
- la projection orthographique ; et
- la projection en **perspective**.

```
glFrustum(left, right, bottom, top,
          near, far)
```

découpe une pyramide tronquée dans le monde 3D.



1.1 Projection : de la 3D à la 2D

exemple 1 :
travailler en **coordonnées "pixel"**
(origine en haut à gauche)

```
def reshape(widht, height):
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    glOrtho(0, width,
            height, 0,
            -1, 1)
```


1.1 Projection : de la 3D à la 2D

exemple 2 :
travailler dans le **cube unité**
en **conservant les proportions** quelque soit
le **rapport hauteur/largeur** de la fenêtre

```
def reshape(widht, height):
    glViewport(0, 0, width, height)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    radius = .5 * min(width, height)
    w, h = width/radius, height/radius
    glOrtho(-w, w, -h, h, -2, 2)
```

1.1 Projection : de la 3D à la 2D

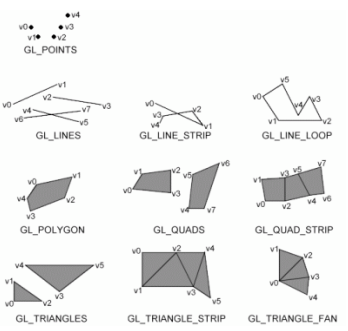
exemple 2 bis :
travailler dans le **cube unité**
en **conservant les proportions** quelque soit
le **rapport hauteur/largeur** de la fenêtre
en **perspective**

```
...
radius = .5 * min(width, height)
w, h = width/radius, height/radius
glFrustum(-w, w, -h, h, 8, 16)
glTranslate(0, 0, -12)
glScale(1.5, 1.5, 1.5)
```


1. Pipeline graphique fixe

1.2 Modèle géométrique

OpenGL permet de dessiner des **primitives** définies par un ensemble de ***vertex*** (points de l'espace).



1.2 Modèle géométrique

OpenGL permet de dessiner des **primitives** définies par un ensemble de ***vertex*** (points de l'espace).

```
glBegin(GL_LINE_LOOP)
for (x, y) in [(-5,  0),
               ( 0, 10),
               ( 5,  0)]:
   glVertex(x, y)
glEnd()
```


1.2 Modèle géométrique

Les coordonnées des points sont données dans le repère "**vue/modèle**" (*modelview*), initialement confondu avec celui du monde.

1.2 Modèle géométrique

Ce repère peut être modifié grâce à des opérations sur la matrice ***modelview*** :

```
glMatrixMode(GL_MODELVIEW)
glPushMatrix()
glPopMatrix()
glLoadIdentity()
glScalef(sx, sy, sz)
glRotatef(a, rx, ry, rz)
glTranslatef(tx, ty, tz)
glMultMatrixf(m)
glLoadMatrixf(m)
```

1.2 Modèle géométrique

Ce repère peut être modifié grâce à
des opérations sur la matrice ***modelview***.

Exemple de composition
des transformations :

```
glMatrixMode(GL_MODELVIEW)
glLoadIdentity()
draw_sun()
glPushMatrix()
glRotate(year_angle, 0, 0, 1)
glTranslate(sun_earth, 0, 0)
glRotate(day_angle, 0, 0, 1)
draw_earth()
glPopMatrix()
```

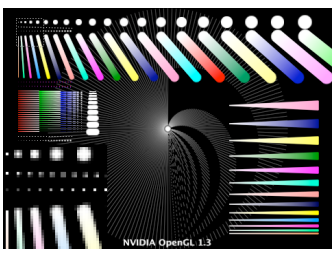

1.2 Modèle géométrique

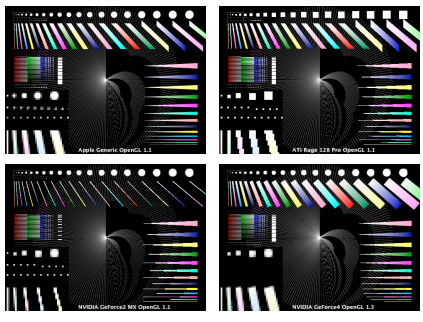
La taille de points, l'épaisseur des lignes,
le **type de remplissage des polygones**
peuvent être spécifiés.

```
glPointSize(size), glLineWidth(width),
glPolygonMode(GL_FACE, GL_MODE)
```


1.2 Modèle géométrique

La **taille de points**, l'**épaisseur des lignes**,
le type de remplissage des polygones
peuvent être spécifiés,
mais **peu de garanties** offertes :





http://homepage.mac.com/arekkusu/bugs/invariance/

1.2 Modèle géométrique

La **couleur courante** est utilisée
pour chaque **vertex**,
et **interpolée linéairement**
pour remplir les primitives.

```
for (color, verticies) in faces:
   glColor(color)
   glBegin(GL_POLYGON)
   for vertex in verticies:
      glVertex(vertex)
   glEnd()
```


1.2 Modèle géométrique

La **couleur courante** est utilisée
pour chaque **vertex**,
et **interpolée linéairement**
pour remplir les primitives.

```
for face in faces:
   glBegin(GL_POLYGON)
   for (color, vertex) in face:
      glColor(color)
      glVertex(vertex)
   glEnd()
```


1.2 Modèle géométrique

L'interpolation se fait séparément pour chacune
des **composantes rouge, verte, bleue**
(et éventuellement **alpha**).

1.2 Modèle géométrique

Les **primitives** sont **dessinées dans l'ordre** demandé,
si la **semi-transparence** est activée,
la **nouvelle couleur** (*source*) est **combinée**
avec celle déjà présente (*destination*).

```
glEnable(GL_BLEND)
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)
```


1.2 Modèle géométrique

Les **primitives** sont **dessinées dans l'ordre** demandé,
si la **semi-transparence** est activée,
la nouvelle couleur (*source*) est combinée
avec celle déjà présente (*destination*),
sinon, la **nouvelle couleur remplace**
celle déjà présente.

1.2 Modèle géométrique

En conséquence, pour obtenir un **rendu correct**
du point de vue de l'**occlusion visuelle**,
il faudrait dessiner les primitives
de la plus éloignée à la plus proche
(et aussi subdiviser les polygones qui s'intersectent …)

1.2 Modèle géométrique

En l'absence de semi-transparence,
pour éviter de trier les faces,
on peut utiliser le **test de profondeur** pour obtenir les occlusions correctes :

- **pour chaque pixel, on stocke la profondeur** du point de la primitive qui lui a donné sa couleur (dans un ***depth-*** ou ***z-buffer***) ;
- on ne **met à jour** un pixel, **que si la profondeur** de la source est bien **inférieure** à celle de la destination.

1.2 Modèle géométrique

```
def init():
   glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH)
   ...
   glEnable(GL_DEPTH_TEST)
   glDepthFunc(GL_LEQUAL)
   glClearDepth(1) # default state
   ...

def display():
   glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
   draw_model()
   glutSwapBuffers()
```


1.2 Modèle géométrique

D'autres tests ont lieu avant le test de profondeur :
- ***scissor*** *test* (pochoir rectangulaire) ;
- ***alpha*** *test* (critère sur la transparence) ; puis
- ***stencil*** *test* (pochoir définit par le *stencil-buffer*).

On peut aussi interdire (masquer) ou non l'écriture dans les différents *buffers* :
`glColorMask, glDepthMask, glStencilMask`

1. Pipeline graphique fixe

1.0 OpenGL : un état graphique partagé
1.1 Projection : du monde 3D à la fenêtre 2D
1.2 Modèle géométrique
1.3 Éclairement
1.4 Images et textures
1.5 Au-delà du mode direct

1.3 Éclairement

OpenGL propose un **modèle d'éclairement** inspiré du monde réel.

Quand il est **activé**, la **couleur** d'un vertex **n'est plus** spécifiée par **la couleur courante**.

```
glEnable(GL_LIGHTING)
glEnable(GL_LIGHT0)
glLight(GL_LIGHT0, GL_POSITION,
        [1, 1, 4, 0])
```


1.3 Éclairement

OpenGL propose un **modèle d'éclairement** inspiré du monde réel.

Quand il est **activé**, la **couleur** d'un vertex n'est plus **spécifiée** par la couleur courante mais par **le matériau** (*material*) et son interaction avec les **sources de lumières actives**.

1.3 Éclairement

Un **matériau** est caractérisé par des couleurs :
- ***emision*** (objet source de lumière mais n'éclairant pas les autres objets) ;
- ***ambiant*** (couleur dans le noir) ;
- ***diffuse*** (diffusion de la lumière incidente) ;
- ***specular*** (réflexion de la lumière incidente),

et par sa "brillance" (***shininess***)

```
glMaterial(GL_FRONT, GL_AMBIANT, [1., 0., 0., 1.])
glMaterial(GL_FRONT, GL_SHININESS, 0)
```


1.3 Éclairement

Une **lumière** est caractérisé par des couleurs :
- ***ambiant*** ;
- ***diffuse*** ;
- ***specular***,

par des contantes d'atténuation,
par son type (directionnelle, spot, etc.)

1.3 Éclairement

La couleur finale du vertex sera
la **somme des contributions**
paramétrées par les couleurs de son matériau
et celles des lumières actives.

1.3 Éclairement

terme global d'**émission** et d'**ambiance** :
• *emission* + *ambiant* × *light model ambiant*

1.3 Éclairement

contribution par lumière active sommant :
• *ambiant* × *light ambiant*
• *diffuse* × *light diffuse* × (**L**•**n**)
• *specular* × *light specular* × (**s**•**n**)^{shininess}
avec :
• **n** la normale au vertex ;
• **L** la direction vertex-source ;
• **s** l'axe de symétrie entre **L**
et la direction de projection.

1.3 Éclairement

Il faut **spécifier les normales** pour chaque vertex :

```
for face in faces:
   glBegin(GL_POLYGON)
   for (vertex, normal) in face:
      glNormal(normal)
      glVertex(vertex)
   glEnd()
```

1.3 Éclairement

contribution par lumière active pondérée par un coefficient d'atténuation :

$1 / (k_c + k_l \times d + k_q \times d^2)$

avec d distance lumière/vertex et
k constances d'atténuation de la lumière.

1.3 Éclairement

contribution par lumière active pondérée par l'écart à l'axe de la lumière pour les sources de type spot.

1.3 Éclairement

Les caractéristiques du matériau peuvent être spécifiées par la couleur.

```
glEnable(GL_COLOR_MATERIAL)
glColorMaterial(GL_FACE, GL_AMBIANT_AND_DIFFUSE)
```

1.3 Éclairement

La couleur de chaque polygone peut être :
- **uniforme** (*flat*), calculée en 1 sommet ;
- **interpolé** (*smooth*), calculée en chaque sommet (c'est l'ombrage de **Gouraud**).

```
glShadeModel(GL_SMOOTH)
```

L'ombrage de **Phong** nécessite d'interpoler les normales et de calculer les contributions de la lumière à chaque pixel (*per pixel lighting*), ce qui n'est possible qu'avec les *fragment shaders*.

1. Pipeline graphique fixe

1.4 Images et textures

OpenGL permet de **manipuler des images** (*raster*) comme des tableaux de pixels :
- **bitmaps** (monochromes) ;
- **images** (pixels de différents formats).

```
glRasterPos(10, 20)
glPixelStore(GL_UNPACK_PARAMETER, value)
glBitmap(width, height, xo, yo, dx, dy, data)
glDrawPixels(width, height,
             GL_RGBA, GL_UNSIGNED_BYTE,
             data)
glCopyPixels(x, y, width, height, GL_COLOR)
```

1.4 Images et textures

OpenGL permet de “**remonter**” le pipeline graphique :

```
glPixelStore(GL_PACK_PARAMETER, value)
char data[width*height*4];
glReadPixels(x, y, width, height,
             GL_RGBA, GL_UNSIGNED_BYTE,
             data);
```


1.4 Images et textures

OpenGL permet d’utiliser des images
pour “**texturer**” les primitives et permettre
de synthétiser des images plus intéressantes.

1.4 Images et textures

Chargement d’une image en mémoire de **texture**

```
tex_id = glGenTextures(1)
glEnable(GL_TEXTURE_2D)
glBindTexture(GL_TEXTURE_2D, tex_id)
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA,
             width, height, 0,
             GL_RGBA, GL_UNSIGNED_BYTE,
             data)
glBindTexture(GL_TEXTURE_2D, 0)
glDisable(GL_TEXTURE_2D)
```

width et *height* doivent (devaient) être
des puissances de 2.

1.4 Images et textures

La texture est munie d'un **repère** tel que le coin en bas à droite ait pour coordonnées (s, t) = (0, 0) et le coin en haut à gauche (s, t) = (1, 1).

Ce repère peut être modifié à l'aide des opérations matricielles habituelles avec :

```
glMatrixMode(GL_TEXTURE)
```


1.4 Images et textures

Placage de **texture** sur une primitive

```
glEnable(GL_TEXTURE_2D)
glBindTexture(GL_TEXTURE_2D, tex_id)

for face in faces:
   glBegin(GL_POLYGON)
   for (vertex, tex_coord) in face:
      s, t = tex_coord
      glTexCoord(s, t)
      glVertex(vertex)
   glEnd()
```


1.4 Images et textures

Les **coordonnées** de texture sont **interpolées** lors du **remplissage** des primitives et sont utilisées pour aller chercher le ***texel*** à utiliser pour remplir chaque pixel.

1.4 Images et textures

Les **coordonnées** de texture sont **interpolées** lors du **remplissage** des primitives et sont utilisées pour aller chercher le ***texel*** à utiliser pour remplir chaque pixel.

La recherche peut ramener :
• le *texel* le plus proche (*nearest*) ;
• une interpolation entre les plus proches (*linear*).

```
glTexParameter(GL_TEXTURE_2D,
               GL_TEXTURE_MIN_FILTER, GL_NEAREST)
```


1.4 Images et textures

Les **coordonnées** de texture sont **interpolées** lors du **remplissage** des primitives et sont utilisées pour aller chercher le ***texel*** à utiliser pour remplir le pixel.

Si **plusieurs niveaux de détails** sont fournis (*mipmaping*), la recherche peut utiliser celui dont la **taille des *texels*** est la plus **proche** de celle **des pixels**, ou encore une **interpolation** entre les *texels* trouvés dans les *mipmaps* le plus proches.

1.4 Images et textures

Les **coordonnées** de texture sont **interpolées** lors du **remplissage** des primitives et sont utilisées pour aller chercher le ***texel*** à utiliser pour remplir chaque pixel.

Si les coordonnées sortent de la texture, on peut les ramener au bord (*clamp*) ou faire comme si la texture était périodique (*repeat*).

```
glTexParameter(GL_TEXTURE_2D,
               GL_TEXTURE_WRAP_S, GL_REPEAT)
```

1.4 Images et textures

Il existe des textures :
• **1D**, utiles comme tables de *lookup* ;
• **2D**, pour plaquer des images sur les primitives ;
• **3D** (peut aussi être utilisée comme 2D + le temps).

1. Pipeline graphique fixe

1.5 Au-delà du mode direct

Le **mode direct** (`glBegin`/`glEnd`) est **inefficace** (plus d'un appel de fonction par vertex).
Il est par ailleurs **absent d'OpenGL ES** !

Plusieurs mécanismes permettent d'éviter cela :
• les ***display lists*** ; et
• les **tableaux de vertex**/couleur/normales ...

1.5 Au-delà du mode direct

Les ***display lists*** permettent de **stocker des groupes** d'appels OpenGL pour les **réexécuter** en un seul appel ensuite.

avant :

```
def display(self):
   for f in self.faces:
      glColor(*f.color)
      glNormal(*f.normal)
      glBegin(GL_POLYGON)
      for i in f.vertex_indicies:
         glVertex(self.verticies[i])
      glEnd()
```


1.5 Au-delà du mode direct

Les ***display lists*** permettent de **stocker des groupes** d'appels OpenGL pour les **réexécuter** en un seul appel ensuite.

après :

```
def display_raw(self):
   for f in self.faces:
      glColor(*f.color)
      glNormal(*f.normal)
      glBegin(GL_POLYGON)
      for i in f.vertex_indicies:
         glVertex(self.verticies[i])
      glEnd()
```

```
def display(self):
   if self.list_id == None:
      self.list_id = glGenLists(1)
      glNewList(self.list_id,
                GL_COMPILE_AND_EXECUTE)
      self.display_raw()
      glEndlist()
   else:
      glCallList(self.list_id)
```


1.5 Au-delà du mode direct

Les ***display lists*** permettent de **stocker des groupes** d'appels OpenGL pour les **réexécuter** en un seul appel ensuite.

Les ***display lists*** ont des performances spectaculaires, mais ne s'appliquent qu'à de la géométrie statique.

1.5 Au-delà du mode direct

Le **mode direct** (`glBegin`/`glEnd`) est **inefficace** (plus d'un appel de fonction par vertex).

Il est par ailleurs **absent d'OpenGL ES** (!) car depuis OpenGL 1.1 on dispose d'une alternative : les **tableaux de vertex**/couleur/...

1.5 Au-delà du mode direct

Les ***vertex (color, normal, texcoord, index) arrays*** permettent de passer des morceaux entiers de géométrie avec peu d'appels.

```
glEnableClientState(GL_VERTEX_ARRAY)
glVertexPointer(3, GL_FLOAT, 0, data)
glDrawElements(GL_TRIANGLES, len(indicies),
               GL_UNSIGNED_INT, indicies)
```

`glInterleavedArrays`
permet d'entrelacer les données au sein d'un seul tableau.

1.5 Au-delà du mode direct

Les ***buffer objects*** (OpenGL 1.5) permettent de mémoriser les tableaux dans la mémoire de la carte graphique.

```
glEnableClientState(GL_VERTEX_ARRAY)

glBindBuffer(GL_ARRAY_BUFFER, glGenBuffers(1))
glBufferData(GL_ARRAY_BUFFER, data,
             GL_STATIC_DRAW)

glVertexPointer(3, GL_FLOAT, 0, 0)
glDrawElements(GL_TRIANGLES, len(indicies),
               GL_UNSIGNED_INT, indicies)
```

1.5 Au-delà du mode direct

Les **tableaux d'attributs de vertex** (OpenGL 2.0) généralisent cette utilisation des *buffer objects* pour passer des attributs génériques de vertex au pipeline programmable.