Table des matières

1.Introduction

Les techniques graphiques qui permettent de composer et d'afficher des images à l'aide de l'ordinateur forment un sujet vaste et en constante évolution. Le but du cours d'infographie est de proposer une introduction à certaines de ces techniques, qui constitue une ouverture vers le domaine. Les principes de base de création et de rendu d'images 2D et 3D seront abordés. Afin d'assurer une dimension pratique, les notions vues seront illustrées via la librairie de fonctions graphiques OpenGL. Cette librairie permet de définir et d'afficher un modèle graphique en C, C++, JAVA. Les applications seront écrites en C et la syntaxe correspondante sera spécifiquement reprise dans ce cours.

Commençons par un tour rapide de la question. Un modèle graphique est composé d'objets élémentaires : points, lignes, polygones, assemblés pour former des objets plus complexes. Ces objets élémentaires, appelés primitives graphiques, sont ensuite *habillés* à l'aide de couleurs ou de textures. Des effets plus sophistiqués peuvent entrer en jeu, comme la superposition d'objets avec effet de transparence ou la spécification d'un effet de brouillard baignant la scène pour n'en citer que deux. Cet habillage peut se compléter par la définition d'un modèle d'éclairage prenant en compte la spécification de sources d'éclairage et de leur intéraction avec les propriétés des *matériaux* composant les primitives graphiques.

Une fois qu'un object est défini, il est possible de le déplacer par rapport au reste de la scène en lui faisant subir des translations et des rotations. Ces transformations peuvent être composées entre elles, ainsi qu'avec des changements d'échelle, suivant une approche systématique. La représentation de scènes animées se fera selon le principe de défilement d'une succession d'*images* représentant la scène dans laquelle les objects animés sont déplacés suivant les trajectoires requises. Une animation est donc caractérisée par un grand nombre de redéfinitions de la scène avec de légères modifications, suivies chaque fois de son affichage à l'écran. On comprend dès lors qu'OpenGL soit conçu avec comme but premier de permettre d'actualiser la scène et de la redessiner dans un laps de temps le plus court possible.

Le processus de transcription du modèle graphique en une image rendue à l'écran est relativement complexe. En résumé, le modèle graphique *habillé* est analysé et traité de manière à définir in fine la couleur de chaque pixel d'un *buffer* de couleurs correspondant à la portion de la scène qui doit être rendue à l'écran. En 3 dimensions, le volume correspondant à cette portion de scène est d'abord défini, puis orienté de telle sorte que l'on obtienne la vision désirée pour l'utilisateur, et finalement projeté en 2D et rendu à l'écran.

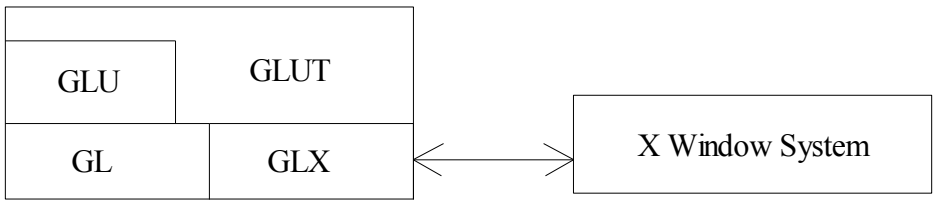
Des outils permettant de gérer l'intervention directe de l'utilisateur au niveau de la scène, tant via le clavier que via la souris, sont aussi disponibles.

OpenGL est un logiciel d'interface avec le matériel (*hardware)* graphique. Afin de rester indépendant de ce matériel, la librairie OpenGL ne contient que des commandes élémentaires. En particulier, aucune commande de gestion des fenêtres d'affichage ni d'écoute des interventions de l'utilisateur ne sont directement disponibles. Il faut donc disposer d'une librairie complémentaire spécifique au système de *windowing*, pour faire le lien avec celui-ci. Il s'agit de GLX pour le système de *windowing* X, de WGL pour Micosoft Windows, de PGL pour IBM OS/2, d'AGL pour Apple.

D'autres librairies complémentaires ont été développées pour faciliter le travail d'écriture du code source des applications OpenGL. L'OpenGL *Utility Librairy* (GLU) contient une serie de

fonctionnalités construites sur base des commandes élémentaires OpenGL. Un autre complément pratique est l'OpenGL *Utility Toolkit* (GLUT), une librairie proposant une séries de fonctions qui permettent de gérer les interactions avec le système de *windowing* et les interventions de l'utilisateur. Les programmes écrits dans le cadre de ce cours feront systématiquement appels à un certain nombre de ces instructions, qui seront décrites pendant les séances de travaux pratiques. GLUT permet aussi de dessiner quelques objects de base en 3D tels que sphères, cubes, tores, théières...

Nous pouvons schématiser la structure de ces différentes librairies comme suit :



La syntaxe spécifique à OpenGL respecte les règles suivantes :

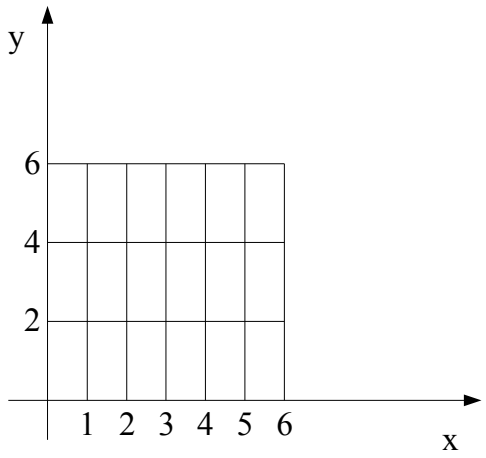
- les noms de fonctions commencent par `gl`, `glu`, ou `glut` suivant la librairie à laquelle elles appartiennent, puis le reste du nom avec une majuscule pour chaque début de mot (ex : `glClear()`, `glutSwapBuffers()`);
- les constantes pré-définies sont en majuscules et commencent par `GL_`, `GLU_` ou `GLUT_` (ex : `GL_PROJECTION`, `GLUT_LEFT_BUTTON`);
- les types de variables pré-définis commencent par `GL` ou `GLU`, puis le reste du nom avec une majuscule pour chaque début de mot à partir du deuxième (ex : `GLdouble`, `GLUquadricObj`).

2. Modèle élémentaire en 2 dimensions

Pour dessiner une scène élémentaire en 2D, on commence par la représenter dans un système de coordonnées abstrait (coordonnées de dessin) en utilisant les formes élémentaires que sont les *primitives graphiques*. Il faut aussi spécifier la couleur de chaque élément de la scène. On définit finalement une zone d'écran dans laquelle une portion choisie de la scène sera projetée pour apparaître à l'utilisateur.

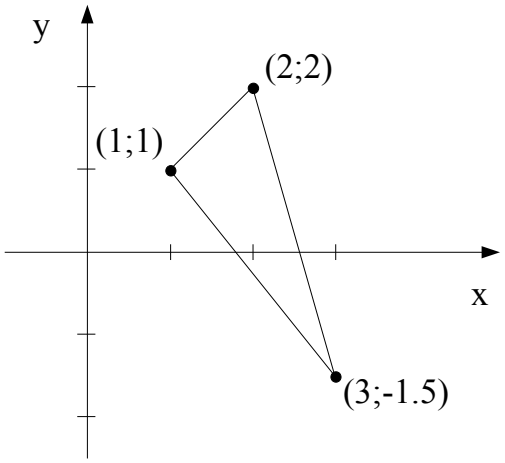
2.1. Les coordonnées de dessin

Les coordonnées de dessin sont un système de références pour la construction du modèle. Il s'agit d'un simple système d'axes : les échelles des axes et l'emplacement de l'origine du système sont choisis de façon à faciliter le travail pour le cerveau humain; ils sont indépendants de la représentation finale dans la fenêtre d'écran! Si l'on veut dessiner une grille quadrillée régulière, on choisira par exemple de faire correspondre l'origine avec le sommet inférieur gauche de la grille. Ensuite, la dimension horizontale de chaque rectangle élémentaire du quadrillage sera par exemple choisie comme correspondant à une unité de l'axe des abcisses (axes des *x*) et leur dimension suivant l'axe des ordonnées (axes des *y*) sera définie en conséquence : si un rectangle est deux fois plus haut que large, sa hauteur sera de deux unités suivant l'axe des *y*.



Une fois que la représentation du modèle est prête, on décrit en OpenGL les différentes primitives graphiques utilisées directement en coordonnées de dessin. Pour décrire le triangle ci-dessous qui relie les points (1;1), (3;-1.5) et (2;2), on utilise le code qui suit :

```
glBegin(GL_TRIANGLES);
   glVertex2d(1.0,1.0);
   glVertex2d(3.0,-1.5);
   glVertex2d(2.0,2.0);
glEnd();
```



Les instructions `glBegin()` et `glEnd()` indiquent le début et la fin de la définition de la primitive graphique, son type étant spécifié par la constante (`GL_TRIANGLES`). L'instruction `glVertex2d()` est répétée autant de fois qu'il y a de sommets en spécifiant l'emplacement de ceux-ci. Les différents types de primitives graphiques disponibles ainsi que l'ordre de spécification des sommets seront abordés plus en détail dans le chapitre 3.

Remarque : à la place de l'instruction `glVertex2d()` *on aurait pu tout aussi bien utiliser* `glVertex2f()` *: de nombreuses instructions OpenGL se terminent par une lettre spécifiant le type de ses arguments numériques. Le* `d` *indique que les arguments seront des "double" et le* `f` *qu'il s'agira de "float". Le* `2d` *qui termine* `glVertex2d()` *n'a donc ici rien à voir avec le fait de travailler en 2 dimensions.*

2.2.Le modèle de couleur

Le *buffer* des couleurs définit la couleur de chaque pixel de la fenêtre d'écran en utilisant un modèle de couleurs. Un certain nombre de bits sont prévus pour stocker l'information relative à chaque pixel. Il existe différents modèles permettant de définir la couleur en un point, dont nous ne retenons qu'un seul dans le cadre du cours : le modéle RGB (Red Green Blue) qui définit une couleur comme un mélange des trois composantes élémentaires *rouge*, *verte* et *bleue*. Les bits disponibles pour chaque pixel sont alloués aux trois composantes de manière aussi équilibrée que possible. Ils définissent trois nombres en virgule flottante, compris entre 0 et 1, qui déterminent les intensités respectives de rouge, de vert et de bleu dans la couleur désirée. Un 0 correspond à une intensité nulle et un 1 à une intensité maximale. On obtient entre autre les cas particuliers suivants:

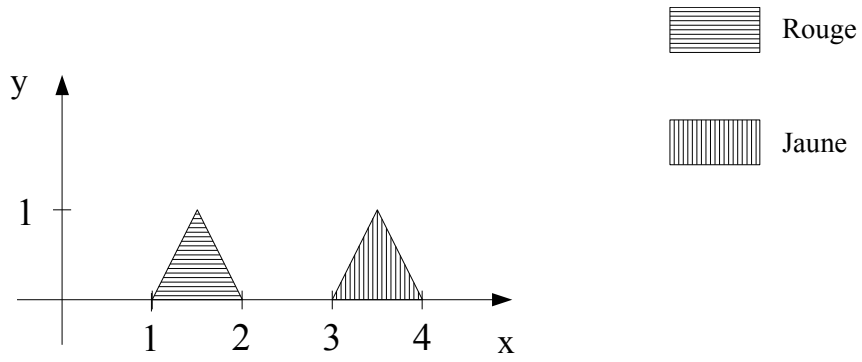
- (0,0,0) définit du noir,
- (1,1,1) définit du blanc,
- (1,0,0) définit du rouge,
- (0,1,0) définit du vert,
- (0,0,1) définit du bleu,
- (*g*,*g*,*g*) où $0<g<1$ définit une nuance de gris d'autant plus foncée que *g* est proche de 0.

Il existe aussi une quatrième composante à ce modèle de couleur : la composante *alpha* (A) utilisée pour obtenir des mélanges de couleurs provenant de plusieurs sources. Nous n'utilisons pas les mélanges pour l'instant. Le modèle s'appelle donc modèle RGBA et nous laisserons la composante *alpha* à zéro.

La première opération relative à la couleur est de définir une couleur de fond pour notre modèle, en appliquant cette couleur à l'ensemble du *buffer* des couleurs. On utilise pour cela l'instruction `glClearColor(R,G,B,A)`. Pour obtenir par exemple un fond vert clair, on écrira `glClearColor(0.0,0.5,0.0,0.0)`.

La définition d'une couleur de dessin fait appel à l'instruction `glColor3f(R,G,B)`, où le "`3f`" indique que la fonction reçoit trois paramètres de type *float*. Une fois une couleur spécifiée de la sorte, elle sera appliquée à toutes les primitives définies après sa spécification, jusqu'à ce qu'on spécifie une autre couleur. Le code suivant dessine un premier triangle rouge, suivi d'un triangle jaune de même forme mais décalé vers la droite :

```
glBegin(GL_TRIANGLES);
   glColor3f(1.0,0.0,0.0);// rouge
   glVertex2d(1.0,0.0);
   glVertex2d(2.0,0.0);
   glVertex2d(1.5,1.0);
   glColor3f(0.0,1.0,1.0);// jaune
   glVertex2d(3.0,0.0);
   glVertex2d(4.0,0.0);
   glVertex2d(3.5,1.0);
glEnd();
```



Si l'on veut colorier différentes parties de l'intérieur d'une primitive graphique à l'aide de plusieurs couleurs différentes, on doit définir au moins autant de primitives qu'il y a de zones de couleurs différentes. Il faut ensuite définir chacune de ces nouvelles primitives graphiques en spécifiant la couleur désirée pour chacune des zones. Le principe est de décomposer le modèle en autant de primitives graphiques que nécessaire.

Il est aussi possible d'obtenir des dégradés de couleur. Pour cela, on associe des couleurs différentes aux différents sommets d'une primitive graphique. En mode RGB, OpenGL détermine alors automatiquement la couleur de chaque pixel définissant l'intérieur de la primitive graphique de façon à obtenir un dégradé continu entre les couleurs associées aux différents

sommets. Cependant, ceci ne s'opère que si le modèle de dégradé est activé (ce qui est le cas par défaut) à l'aide de l'instruction `glShadeModel(GL_SMOOTH)` appelée dans la fonction "`display`" introduite dans la section 2.4 ci-dessous. Lorsque le modèle de dégradé est désactivé (ce qui s'obtient par l'appel de `glShadeModel(GL_FLAT)`), une seule couleur de remplissage est utilisée pour chaque primitive graphique.

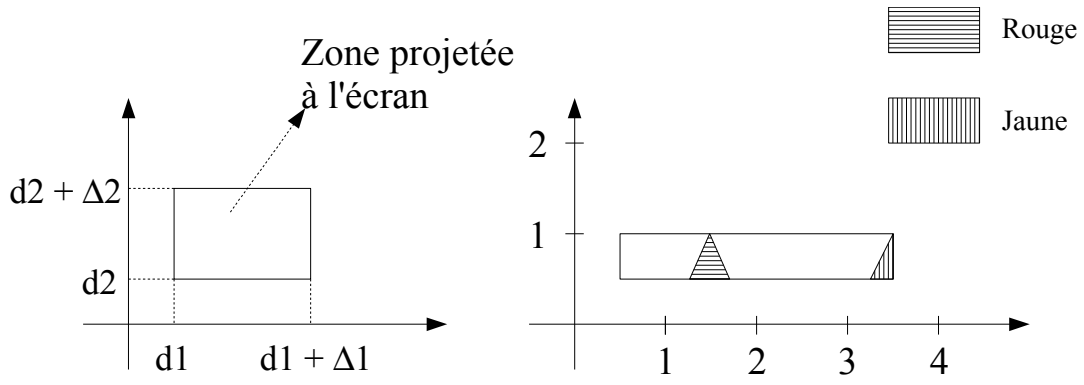
Lorsque le modèle de dégradé est activé, l'exemple suivant dessine un triangle dont une base est jaune et dont la couleur de remplissage passe du jaune au rouge au fur et à mesure que l'on s'approche du troisième sommet :

```
glBegin(GL_TRIANGLES);
   glColor3f(0.0,1.0,1.0);// jaune
   glVertex2d(2.0,0.0);
   glVertex2d(1.0,0.0);
   glColor3f(1.0,0.0,0.0);// rouge
   glVertex2d(1.5,1.0);
glEnd();
```

2.3.Définir la zone de projection à l'écran

Une fois que la description du modèle est terminée, il faut définir la partie du modèle, ou zone, qui doit être projetée à l'écran. Dans le cas d'un modèle à deux dimensions, il existe une instruction simple dans la librairie GLU permettant de définir une zone rectangulaire devant apparaître à l'écran : `gluOrtho2D()`. Cette fonction prend 4 paramètres, le premier et le troisième définissent le coin inférieur gauche de la zone à projeter, tandis que le deuxième et le quatrième paramètres définissent son coin supérieur droit. Ces 4 paramètres sont définis en coordonnées de dessin (les mêmes que celles utilisées pour définir le modèle).

La zone définie par `gluOrtho2D(d1,d1+Δ1,d2,d2+Δ2)` est illustrée dans le cadre de gauche ci-dessous, tandis que le cadre de droite montre la zone qui sera projetée si l'instruction `gluOrtho2D(0.5,3.5,0.5,1.0)` est appelée pour le modèle des triangles colorés proposé dans le paragraphe précédent.



La projection de la zone choisie dans la fenêtre d'écran se fait automatiquement. Le chapitre 4 sera consacré à la définition et à la gestion de la fenêtre d'écran.

2.4.En pratique

Dans l'exemple présenté ci-dessus, les instructions définissant le modèle (c-à-d. les triangles rouge et jaune) doivent apparaître dans la fonction appelée par la fonction `glutDisplayFunc()`. Classiquement on utilise le nom "`display`" pour cette fonction. D'autre part, l'instruction `gluOrtho2D()` apparaît dans une fonction que nous nommerons "`viewport`", elle-même appelée dans "`display`" avant la définition du modèle. Nous détaillerons la fonction "`viewport`" dans le chapitre 4. On a donc la structure suivante :

```
void display(void){
    ...
    glShadeModel(GL_FLAT);
    glClearColor(0.0,0.5,0.0,0.0)// fond vert clair

    viewport();

    glBegin(GL_TRIANGLES);
      glColor3f(1.0,0.0,0.0);// rouge
      glVertex2d(1.0,0.0);
      glVertex2d(2.0,0.0);
      glVertex2d(1.5,1.0);
      glColor3f(0.0,1.0,1.0);// jaune
      glVertex2d(3.0,0.0);
      glVertex2d(4.0,0.0);
      glVertex2d(3.5,1.0);
    glEnd();
    ...
}

void viewport(){
    ...
    gluOrtho2D(0.5,3.5,0.5,1.0);// définit la zone projetée à l'écran
    ...
}

int main(int argc, char** argv){
    ...
    glutDisplayFunc(display);
    ...
}
```

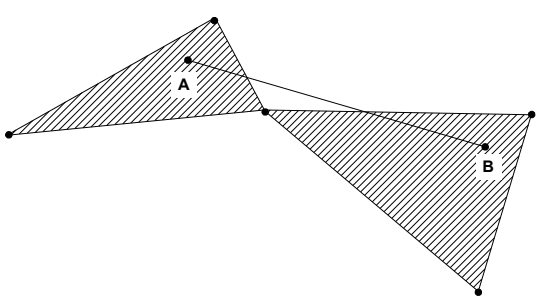
3. Les primitives graphiques

Une scène apparaissant à l'écran peut soit provenir d'une image existante, soit être une projection d'un modèle construit par exemple en OpenGL. Les images ont une provenance externe : il peut s'agir par exemple d'une photographie digitalisée, d'une image construite par un programme quelconque puis sauvegardée, ... Dans la suite, nous nous intéresserons exclusivement à la définition d'un modéle graphique comme un travail de dessin. Il s'agit de combiner des formes élémentaires appelées primitives graphiques élémentaires (points, lignes, triangles, ...), et de leur associer des couleurs et un modèle de lumière. Ce modèle pourra être observé, en tout ou en partie, d'un point quelconque de l'espace en 3 dimensions, et le résultat projeté dans la fenêtre d'écran.

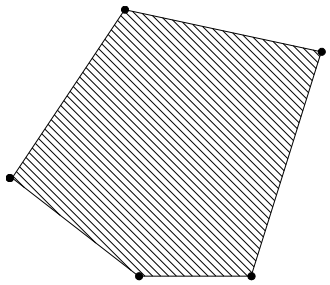
Un modèle en 2 dimensions pourrait être par exemple une fresque colorée de formes géométriques. Une fois la fresque définie en OpenGL, on peut définir une partie rectangulaire de celle-ci, sur laquelle on désire se focaliser. Cette partie, ou zone, rectangulaire peut être orientée suivant un sens quelconque, comme si l'on tournait la fresque sur elle-même par rapport à l'oeil d'un observateur, puis projetée dans une fenêtre apparaissant à l'écran.

3.1. Les primitives graphiques élémentaires

Les primitives graphiques élémentaires disponibles en OpenGL sont des points, des segments de droites liant deux points, des triangles, des quadrilatères et des polygones. Les primitives élémentaires doivent être convexes. C'est-à-dire que quels que soient deux points choisis à l'intérieur d'une primitive graphique, le segment qui les lie doit être lui-même entièrement compris à l'intérieure de cette primitive graphique. Ce concept est illustré ci-dessous : le pentahèdre dessiné dans le dessin de gauche est non convexe, comme le montre le segment AB. D'autre part, le pentahèdre du dessin de droite est convexe. Une figure non convexe devra être décomposée en autant de primitives convexes que nécessaire. Par exemple, le pentahèdre du dessin de gauche ci-dessous doit être défini en utilisant au minimum deux triangles (un triangle étant toujours convexe !).

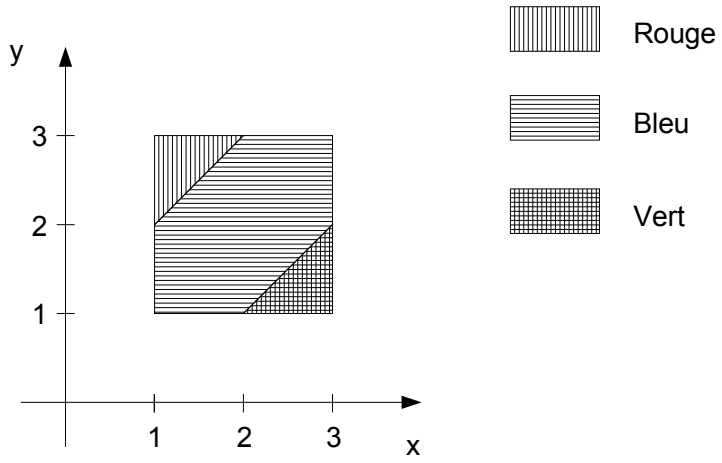


NON CONVEXE



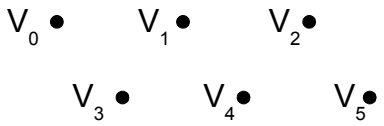
CONVEXE

Une figure convexe peut elle-même devoir être décomposée en plusieurs primitives graphiques lorsque sa composition n'est pas homogène : par exemple si cette figure convexe prend plusieurs couleurs différentes. Dans l'illustration ci-dessous, le carré tricolore doit être décomposé en deux triangles et un hexahèdre de façon à pouvoir définir la couleur adéquate pour chacun d'eux.

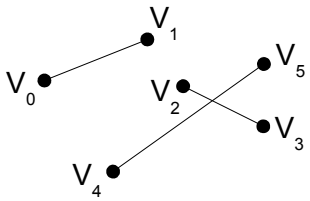


Les instructions `glBegin()` et `glEnd()` indiquent le début et la fin de la définition d'une primitive graphique, son type étant spécifié par une constante passée en paramètre à `glBegin()`. Entre ces deux instructions, l'instruction `glVertex2d()` (modèle en 2 dimensions) ou `glVertex3d()` (modèle en 3 dimensions) est répétée autant de fois qu'il y a de sommets à définir, en spécifiant l'emplacement de ceux-ci en coordonnées de dessin. Les différents types de primitives graphiques disponibles en OpenGL sont les suivants :

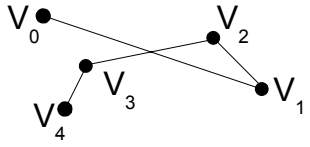
➔ GL_POINTS : points isolés les uns des autres;



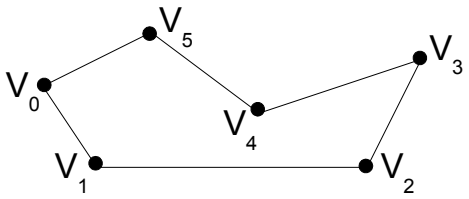
➔ GL_LINES : segments de droite isolés les uns des autres. Le premier segment est défini par les 2 premiers sommets, le deuxième par les 2 sommets suivants, etc;



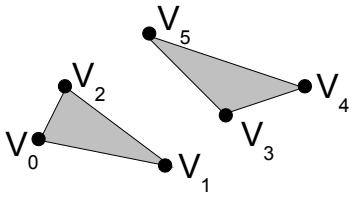
➔ GL_LINE_STRIP : ligne brisée définie en reliant les sommets entre eux. Le premier et le dernier sommet sont traités comme "isolés";



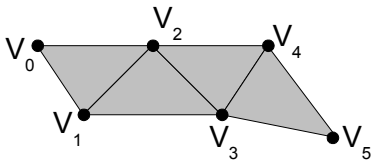
➔ GL_LINE_LOOP : ligne brisée définie en reliant les sommets entre eux. Le premier et le dernier sommet sont reliés entre eux pour former une boucle;



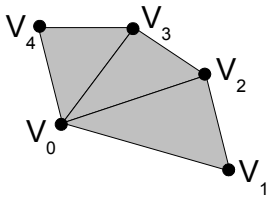
GL_TRIANGLES : triangles isolés les uns des autres. Le premier triangle est défini par les 3 premiers sommets, le deuxième par les 3 sommets suivants, etc;



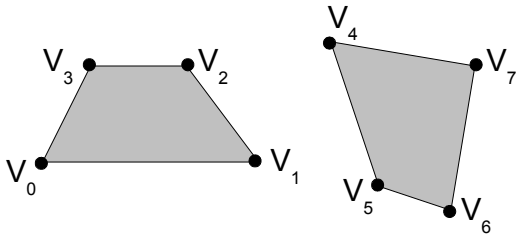
➔ GL_TRIANGLE_STRIP : triangles se développant "en drapeau". Le premier triangle est défini par les 3 premiers sommets (sommets 1 à 3), le deuxième par les sommets 2 à 4, etc. Chaque nouveau triangle est défini en utilisant le dernier côté du triangle précédent plus un nouveau sommet;



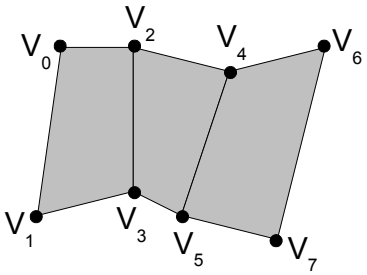
➔ GL_TRIANGLE_FAN : triangles se développant "en éventail". Le premier triangle est défini par les 3 premiers sommets (sommets 1-2-3), le deuxième par les sommets 1-3-4, le troisième par les sommets 1-4-5, etc. Tous les triangles ont le premier sommet en commun et chaque nouveau triangle est défini en utilisant le dernier côté du triangle précédent plus un nouveau sommet;



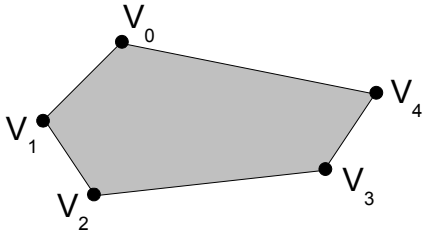
➔ GL_QUADS : quadrilatères convexes isolés les uns des autres. Le premier quadrilatère est défini par les 4 premiers sommets, le deuxième par les 4 sommets suivants, etc. Les sommets sont reliés dans leur ordre d'apparition. Le premier et le dernier sommet de chaque groupe de 4 sont reliés entre eux;



➔ GL_QUAD_STRIP : quadrilatères convexes accolés les uns aux autres. Le premier quadrilatère est défini par les 4 premiers sommets (1 à 4), le deuxième par les sommets 3 à 6, etc. Chaque groupe de 4 sommets définit un quadrilatère en les reliant de la façon suivante : premier-deuxième-quatrième-troisième-premier.



➔ GL_POLYGON : polygone convexe défini en reliant les sommets dans leur ordre d'apparition. Le premier et le dernier sommet sont reliés entre eux.



L'exemple qui précède la présentation des différents types de primitives graphiques pourrait se programmer (entre autres) de la façon suivante :

```
glBegin(GL_TRIANGLES);
   glColor3f(0.4,0.8,0.0);//dominante verte
   glVertex2d(1.0,2.0);
   glVertex2d(2.0,3.0);
   glVertex2d(1.0,3.0);
```

```
    glColor3f(0.7,0.0,0.3);//dominante rouge
    glVertex2d(2.0,1.0);
    glVertex2d(3.0,1.0);
    glVertex2d(3.0,2.0);
  glEnd();
  glBegin(GL_POLYGON);
    glColor3f(0.0,0.1,0.5);//dominante bleue
    glVertex2d(1.0,1.0);
    glVertex2d(2.0,1.0);
    glVertex2d(3.0,2.0);
    glVertex2d(3.0,3.0);
    glVertex2d(2.0,3.0);
    glVertex2d(1.0,2.0);
  glEnd();
```

Il existe aussi une fonction permettant de définir un rectangle sur base de deux sommets opposés (x1,y1) et (x2,y2). Il s'agit de `glRectf(x1,y1,x2,y2)`.

3.2. Caractéristiques élémentaires

Il est possible de définir la dimension d'un point, le motif d'une ligne ou le mode d'affichage d'un polygone.

La dimension d'un point et la largeur d'une ligne sont définies en nombre de pixels via les instructions `glPointSize(s)` et `glLineWidth(s)`, respectivement. Elles prennent un "float" strictement supérieur à zéro comme paramètre. La dimension par défaut est de 1 pixel.

L'instruction `glLineStipple(GLint factor, GLushort pattern)` permet de définir un motif alterné (du type pointillé) pour le rendu des segments de droites. Chacun des 16 bits de l'"unsigned short" `pattern` définit le caractère allumé (1) ou éteint (0) du pixel correspondant. Le bit d'ordre le plus bas (le plus à droite) caractérise le pixel le plus à gauche. Le motif ainsi défini est répété pour construire le segment de droite. La longueur de chaque séquence de pixels allumés/éteints est multipliée par un facteur `factor`. Considérons les deux instructions suivantes : `glLineStipple(1,14649)` et `glLineStipple(3,14649)`. Le motif correspondant est "1001110010011100" et le rendu d'un segment de droite avec chacun des deux motifs est illustré ci-dessous (avec un grossissement) :

Factor = 1 :

Factor = 3 :

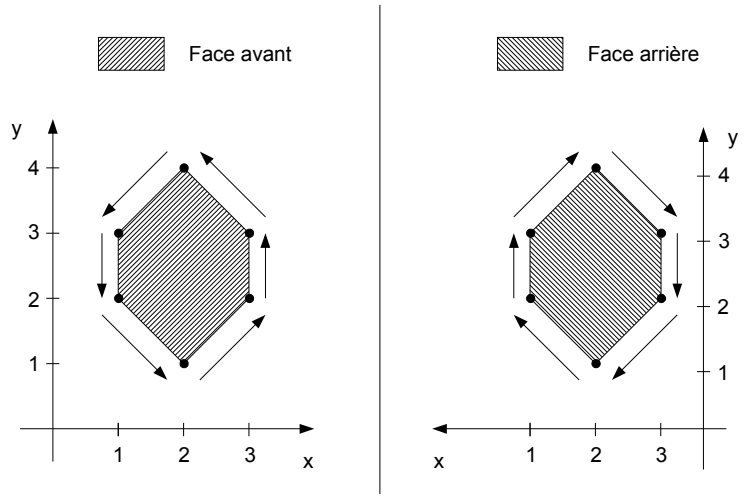
Remarque : pour des raisons d'efficacité, tout une série de possibilités offertes par OpenGL sont désactivées par défaut. On appelle "états de base" les états qu'il est nécessaire d'activer pour pouvoir utiliser une de ces possibilités. La commande d'activation est `glEnable(...)` *et la commande de désactivation est* `glDisable(...)`. *Dans l'exemple ci-dessus, on devra utiliser* `glEnable(GL_LINE_STIPPLE)` *avant la première instruction* `glLineStipple(...)`. *Dès que*

l'état de base peut être désactivé (`glDisable(GL_LINE_STIPPLE)` *dans notre exemple), il est vivement conseillé de le faire pour garantir une vitesse maximale de rendu des images. Dans la suite chaque état de base sera signalé lors de sa première apparition.*

Lors de la définition d'un polygone, il est possible de définir plusieurs modes de rendu. Ce choix se fait en utilisant l'instruction `glPolygonMode(face,mode)` qui vient avant l'appel de `glBegin(GL_POLYGON)`.

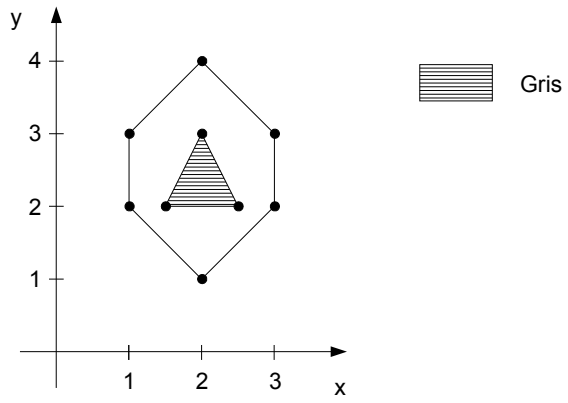
Le paramètre `mode` définit le type d'affichage du polygone. Il prend la valeur `GL_POINT` pour l'affichage des sommets seuls, `GL_LINE` pour l'affichage des côtés seuls (mode dit "fil de fer") et `GL_FILL` pour un polygone rempli par la dernière couleur définie.

Dans un modèle 3D, il est possible de définir des caractéristiques différentes pour chacune des deux faces d'un polygone. Le paramètre `face` définit la ou les faces concernée(s) : il prend les valeurs `GL_FRONT`, `GL_BACK` ou `GL_FRONT_AND_BACK` suivant que le choix s'applique à la face avant seule, à la face arrière seule ou aux deux faces. **Par défaut, la face avant est la face définie par un parcours anti-horlogique des sommets du polygone dans l'ordre de leur définition en considérant un axe des *x* dirigé vers la droite et un axe des *y* dirigé vers le haut.** Ceci est illustré par le graphique ci-dessous : si les points sont définis dans l'ordre proposé par les flêches la face avant sera celle visible lorsque l'on se trouve face à la figure de gauche, tandis que la face arrière sera celle visible lorsque l'on se trouve face à la figure de droite.



Par défaut, le mode est `GL_FILL` pour les deux faces. Dans l'illustration ci-dessous, on ne désire faire apparaître que le pourtour d'un hexaèdre régulier entourant un triangle. On pourra utiliser les instructions suivantes :

```
glBegin(GL_TRIANGLES);
   glColor3f(0.5,0.5,0.5);// gris
   glVertex2d(1.5,2.0); glVertex2d(2.5,2.0); glVertex2d(2.0,3.0);
glEnd();
glPolygonMode(GL_FRONT,GL_LINE);
glBegin(GL_POLYGON);
   glVertex2d(2.0,1.0); glVertex2d(3.0,2.0); glVertex2d(3.0,3.0);
   glVertex2d(2.0,4.0); glVertex2d(1.0,3.0); glVertex2d(1.0,2.0);
glEnd();
```



Il est possible d'agir sur d'autres caractéristiques d'un polygone (orientation de la face avant, motif de remplissage, suppression de l'affichage de certains bords ou d'une ou des deux faces), mais leur description dépasse le cadre de cette section.

3.3. Quadriques

Les primitives graphiques élémentaires introduites ci-dessus permettent de construire n'importe quel objet dont les contours sont des segments de droite, mais qu'en est-il lorsque l'on veut dessiner un pourtour ayant une courbure, comme par exemple un cercle, une ellipse ou un ballon de rugby? Il sera alors nécessaire de définir l'objet soit point par point, soit à l'aide d'un grand nombre de segments de droites qui donnent l'impression de courbure recherchée lorsqu'ils sont reliés entre eux.

Si l'on désire dessiner un cercle de rayon *r* centré à l'origine, une approche naïve consistera à définir un certain nombre de points de son périmètre et de les relier par des segments de droite :

```
glBegin(GL_LINE_LOOP);
   for(int x=0; x<360; x+=10)
      glVertex2d(r*cos(x),r*sin(x));
glEnd();
```

Plus le pas (`x` ci-dessus) sera petit, plus le nombre de segments sera élevé et plus l'impression de cercle sera bonne. Cette approche est cependant à proscrire car elle requiert le calcul d'un sinus et d'un cosinus à chaque itération. Il existe des algorithmes beaucoup plus performants qui définissent chaque point du cercle en fonction du précédent. OpenGL propose de construire des disques, sphères et cylindres (quadriques) en utilisant de tels algorithmes.

L'obtention d'un objet de type quadrique nécessite trois étapes :

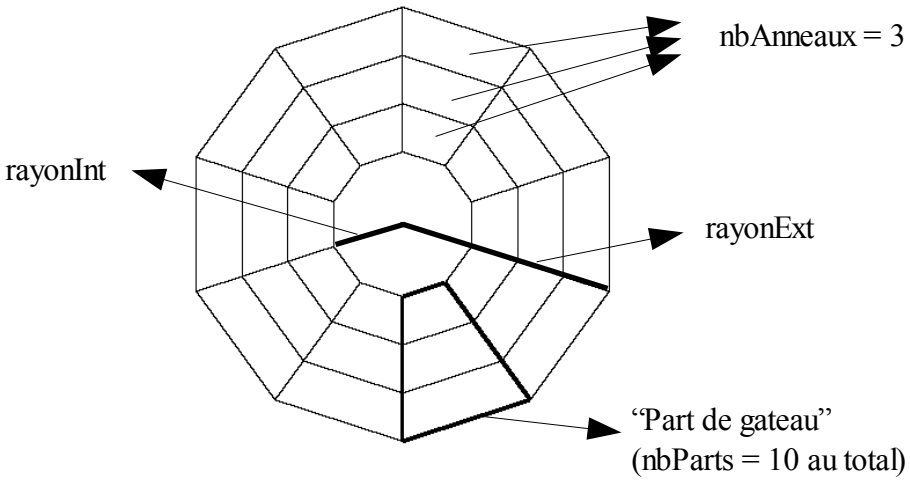
1. Déclarer et instancier une structure de donnée qui contiendra toutes les informations nécessaires à la gestion de la quadrique. Un telle structure est contenue dans un objet de type `GLUquadricObj` et son instanciation se fait via la commande `gluNewQuadric()`. On libère la mémoire utilisée par l'instruction `gluDeleteQuadric(...);`
2. Définir les attributs de la quadrique : mode d'affichage, orientation, vecteurs normaux. Seul le mode d'affichage nous intéresse ici. Il est défini par `gluQuadricDrawStyle(qobj,`

`drawStyle)`, où `qobj` est un pointeur vers la structure définie en 1 ci-dessus et `drawStyle` peut prendre les valeurs `GLU_POINT`, `GLU_LINE`, `GLU_SILHOUETTE` ou `GLU_FILL`. `GLU_POINT` affiche les sommets uniquement, `GLU_LINE` affiche en mode "fil de fer" toutes les primitives élémentaires qui forment la quadrique, `GLU_SILHOUETTE` est similaire à `GLU_LINE` mais les segments séparant deux faces coplanaires ne sont pas affichés, enfin `GLU_FILL` affiche en mode "plein" toutes les primitives élémentaires qui forment la quadrique;

3. Appeler la fonction qui va générer les sommets et autres attributs de la quadrique. Il existe 4 instructions différentes pour générer les 4 types de quadriques disponibles : `gluSphere(...)`, `gluCylinder(...)`, `gluDisk(...)` et `gluPartialDisk(...)`. Nous allons détailler ci-dessous l'utilisation de `gluDisk(...)`, tandis que le lecteur consultera le manuel de référence OpenGL pour les 3 autres.

Voyons plus en détail la commande permettant de définir un disque : `gluDisk(disque,rayonInt,rayonExt,nbParts,nbAnneaux)` définit un disque centré à l'origine des coordonnées de dessin (en (0,0)), de rayon intérieur `rayonInt` et de rayon extérieur `rayonExt`. Le paramètre `disque` pointe vers la structure de donnée nécessaire. Le paramètre `nbParts` indique le nombre de sommets qui, reliés entre eux par des segments de droite, représenteront les cercles intérieurs et extérieurs. Ce paramètre peut aussi être vu comme un nombre de "parts de gateau" divisant le disque. Finalement, pour la gestion de dégradés de couleur et des modèles de lumière, un disque est divisé en formes géométriques élémentaires délimitées par les "parts de gateau" introduites ci-dessus d'une part, et `nbAnneaux` cercles concentriques intermédiaires entre les cercles intérieur et extérieur d'autre part. L'ensemble de ces formes élémentaires sont visibles sur le disque illustré ci-dessous en mode "fil de fer", obtenu via les instructions :

```
GLUquadricObj *disque; //étape 1
disque=gluNewQuadric(); //étape 1
gluQuadricDrawStyle(disque,GLU_LINE); //étape 2
gluDisk(disque,1.0,3.0,10,3); //étape 3
```



3.4. Comment déplacer un objet

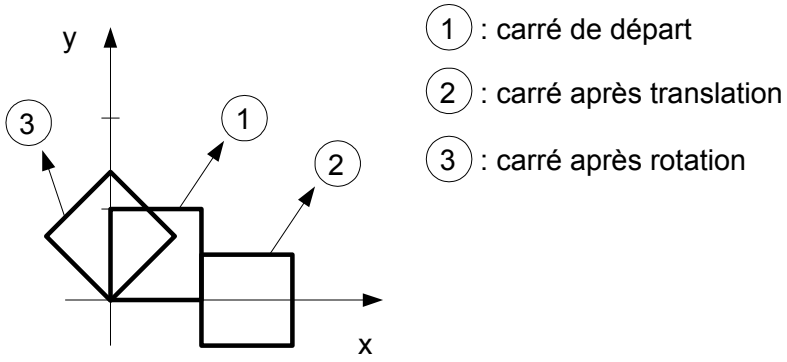
On désire souvent faire apparaître un même objet à plusieurs endroits différents d'un modèle, ou encore afficher un modèle plusieurs fois d'affilée en déplaçant légèrement un objet donné pour obtenir un effet de mouvement. La technique consiste à définir l'objet à un endroit fixe du système de coordonnées et à le déplacer en lui appliquant des translations et/ou des rotations pour l'amener à l'endroit désiré.

OpenGL offre la possibilité de translater un objet d'un endroit à un autre en définissant un déplacement suivant l'axe des abcisses combiné à un déplacement suivant l'axe des ordonnées (en 2D). Il est aussi possible de faire tourner un objet d'un certain angle autour d'un axe. Nous allons brièvement décrire ces deux types de transformations, dont le mécanisme sera détaillé dans le chapitre 5. Dans la suite, il sera nécessaire de raisonner en 3 dimensions : l'axe des abcisses sera noté *x*, l'axe des ordonnées *y*, et l'axe de profondeur perpendiculaire au plan formé par *x* et *y* sera noté *z*. Partant de l'origine (*x=0,*y=*0,*z=0), le sens positif de chaque axe sera le suivant : vers la droite pour les *x*, vers le haut pour les *y* et sortant du plan *x*-*y* vers l'observateur pour les *z*.

Pour translater un objet, on utilise l'instruction `glTranslated(Δx,Δy,Δz)` avant d'appeler sa définition. Cela a pour effet de le déplacer du delta (Δ) spécifié le long de chacun des 3 axes. En deux dimensions, le Δz sera mis à zéro puisque l'on ne quitte pas le plan *x*-*y*.

Pour faire pivoter un objet, on utilise l'instruction `glRotated(angle,x,y,z)` avant d'appeler sa définition. Cela a pour effet de lui appliquer une rotation de `angle` degrés dans le sens anti-horlogique autour de l'axe passant par l'origine du repère et par le point (`x`,`y`,`z`). Un `angle` négatif correspond à une rotation dans le sens horlogique. En deux dimensions, la rotation se fera autour de l'axe *z*, c'est à dire que l'on appellera `glRotated(angle,0,0,z)`, où `z` peut prendre n'importe quelle valeur non nulle; cependant on utilise classiquement `z=1.0`.

Illustrons ces déplacements à l'aide d'un exemple simple : nous disposons d'une fonction qui construit un carré de côté unitaire dont le coin inférieur gauche correspond à l'origine du repère et dont les côtés sont parallèles aux axes. Ce carré est d'abord dessiné à sa position de définition, puis en subissant une translation d'une unité vers la droite et d'une demi-unité vers le bas, et finalement en subissant une rotation de 45° dans le sens anti-horlogique dans le plan *x*-*y* (autour de l'axe *z*).

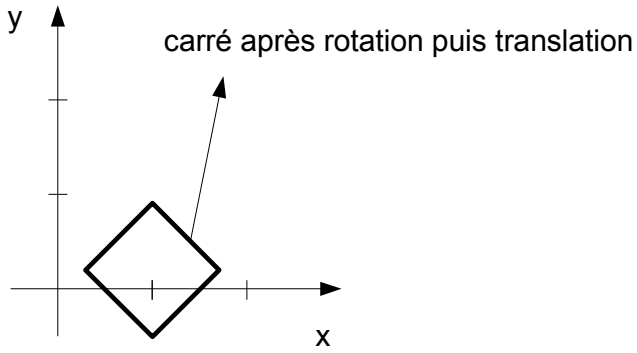


Voici le code correspondant :

```
void carréUnité(void)
  {
    glPolygonMode(GL_FRONT;GL_LINE); //mode fil de fer
    glColor3d(0.0,0.0,0.0);// noir
    glBegin(GL_POLYGON); //définition du carré
      glVertex2d(0.0,0.0);
      glVertex2d(1.0,0.0);
      glVertex2d(1.0,1.0);
      glVertex2d(0.0,1.0);
    glEnd();
  }
  ...
  void display(void)
  {
    ...
    carréUnité(); // carré de départ
    glPushMatrix();
    glTranslated(1.0,-0.5,0.0);
    carréUnité(); // carré après translation
    glPopMatrix();
    glRotated(45.0,0.0,0.0,1.0);
    carréUnité(); // carré après rotation
    ...
  }
```

Remarques :

1. *Dans le cas présent, il est préférable de remplacer le bloc d'instructions définissant le carré par l'instruction* `glRectf(0.0,0.0,1.0,1.0)` *introduite à la fin de la section 3.1. Cette instruction est plus simple et en outre optimale pour la définition de rectangles ;*
2. *Les instructions* `glPushMatrix()` *et* `glPopMatrix()` *apparaissant dans le code ci-dessus permettent dans ce cas-ci d'annuler l'effet de la translation avant d'opérer la rotation. Si l'on avait omis ces instructions, le dernier carré aurait été affiché après avoir subi la rotation puis la translation, comme illustré ci-dessous. Nous reviendrons sur ces instructions au chapitre 5.*


y
carré après rotation puis translation
x

4. Fenêtre d'écran et viewport

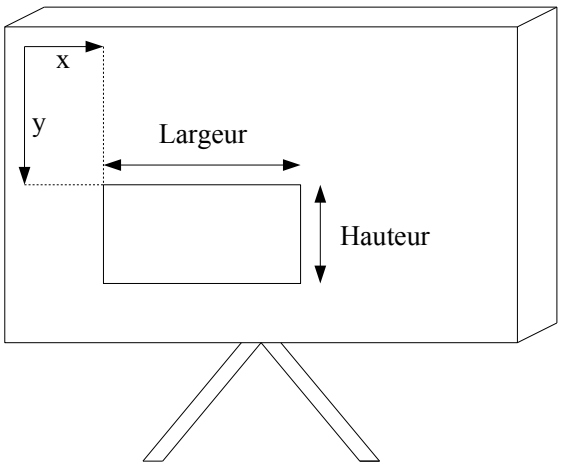
Une fois qu'un modèle a été construit en coordonnées de dessin, il faut décider de ce qui sera affiché à l'écran, ce qui nécessite plusieurs étapes qui sont expliquées dans le présent chapitre. D'une part, l'on définit une *fenêtre d'écran* dans laquelle seront affichées des vues du modèle, cette fenêtre pouvant elle-même être divisée en plusieurs sous-fenêtres appelées *viewports*. Chacune de ces viewports peut afficher une vue choisie du modèle, en parallèle avec ce qu'affichent les autres viewports. D'autre part, il faut définir pour chaque viewport la partie du modèle que l'on désire afficher et le type de projection qu'OpenGL doit utiliser pour faire correspondre cette partie du modèle avec la viewport.

Deux autres sujets liés à l'affichage seront aussi abordés : la gestion du redimensionnement de la fenêtre d'écran par l'utilisateur d'une part, et l'identification d'une primitive graphique du modèle de dessin par transformation inverse de coordonnées d'écran sélectionnées via la souris d'autre part.

4.1. La fenêtre d'écran

Les caractéristiques de la fenêtre apparaissant à l'écran sont au départ spécifiées à l'aide de fonctions de la librairie GLUT. Il faut spécifier les dimensions (largeur et hauteur) de la fenêtre ainsi que la position de son coin supérieur gauche avant de demander son affichage.

Les dimensions de la fenêtre d'écran, tout comme les coordonnées d'écran, sont calculées en pixels. Il est important de noter qu'*en coordonnées d'écran l'axe des ordonnées (y) pointe vers le bas*. L'origine du repère est donc le coin supérieur gauche de la fenêtre d'écran, chaque pixel étant localisé par un couple de coordonnées (*x*,*y*) où un *x* positif désigne un nombre de pixels vers la droite et un *y* positif désigne un nombre de pixels vers le bas. Le positionnement du coin supérieur gauche de la fenêtre d'écran est quant à lui spécifié en nombre de pixels vers la droite et vers le bas à partir du premier pixel en haut à gauche du moniteur (l'écran de l'ordinateur).



Les 3 commandes "glut" permettant d'afficher une fenêtre d'écran sont appelées dans la fonction `main(...)` :

- `glutInitWindowSize(largeur,hauteur)` définit la taille de la fenêtre d'écran;
- `glutInitWindowPosition(x,y)` définit `(x,y)`, l'emplacement du coin supérieur

gauche de la fenêtre d'écran;

- `glutCreateWindow(char *nom)` demande la création de la fenêtre d'écran. On peut, via `nom`, donner un titre à la fenêtre d'écran. Cette instruction renvoie aussi un entier identifiant la fenêtre d'écran qui est ouverte.

4.2.La viewport et la matrice de projection

L'affichage ne se fait pas directement dans la fenêtre d'écran, mais dans une zone ou partie rectangulaire de celle-ci, appelée *viewport*. Nous considérons ici le cas simple d'un modèle 2D avec l'oeil de l'observateur sur une ligne perpendiculaire au plan du modèle. La partie du modèle que l'on veut afficher sera projetée dans la viewport suivant la direction définie par la vue de l'observateur. Il faut donc spécifier 3 choses : l'emplacement et les dimensions de la viewport, la partie (rectangulaire) du modèle que l'on veut afficher, et le type de projection à utiliser pour faire la correspondance entre le modèle et la viewport. OpenGL gère alors la transformation de la zone choisie du modèle (niveau abstrait, en coordonnées de dessin) pour obtenir les caractéristiques de chacun des pixels définissant la viewport dans le buffer des couleurs (niveau concret).

Les instructions OpenGL qui permettent de définir une viewport et son contenu doivent apparaître dans la fonction dite "*reshape(...)*". L'intérêt de cette fonction sera expliqué dans la section 4.3.
Les dimensions de la viewport sont définies via l'instruction `glViewport(vp1,vp2,w,h)`, où (`vp1`,`vp2`) définit le coin inférieur gauche de la viewport et *w* et *h* définissent sa largeur et sa hauteur. **`vp1`, `vp2`, *w* et *h* doivent être fournis en pixels et correspondre à un système d'axes où *y* pointe vers le haut, *x* pointe vers la droite et l'origine du repère correspond au coin inférieur gauche de la fenêtre d'écran.** La définition de la partie du modèle à afficher et du type de projection se fait via les 3 instructions ci-dessous :

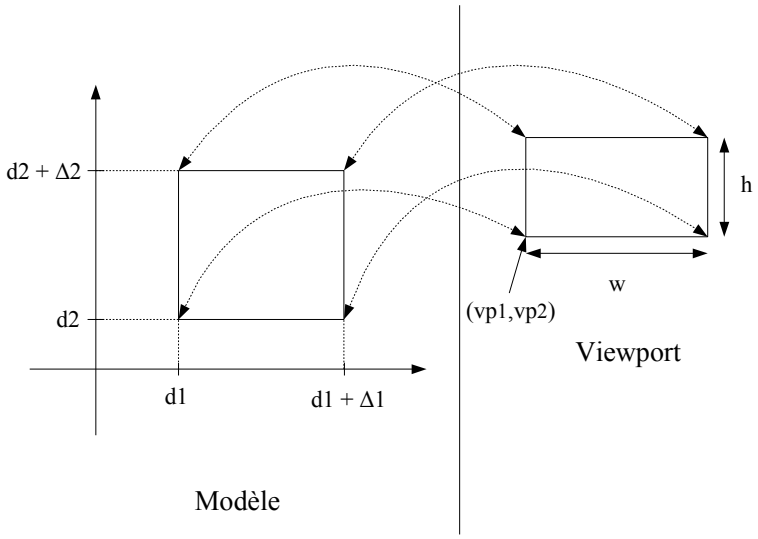
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(d1,d1+Δ1,d2,d2+Δ2);
```

L'instruction `glMatrixMode()` indique à OpenGL que l'on change de type de matrice de travail. Il existe deux types de matrices de travail : les matrices de projection qui définissent la projection utilisée pour projeter le modèle dans la viewport, et les matrices de transformation géométrique qui définissent les transformations à appliquer à un ensemble de primitives graphiques dans le modèle de dessin (via `glTranslated(...)` ou `glRotated(...)` par exemple). Le passage d'un type de matrice à l'autre se fait en passant les constantes `GL_PROJECTION` et `GL_MODELVIEW` à `glMatrixMode()`, respectivement. Cette instruction est nécessaire chaque fois que l'on doit donner des instructions concernant une projection après des instructions concernant des transformations géométriques du modèle de dessin, et vice versa.

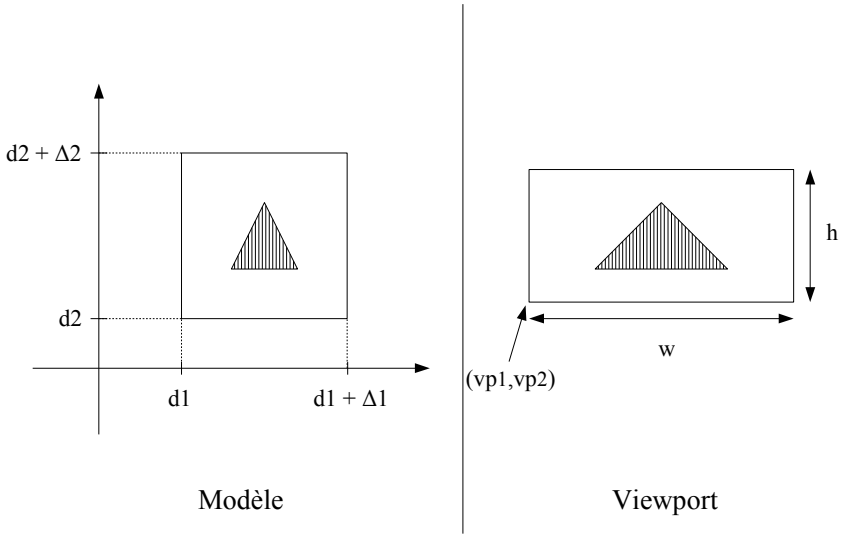
L'instruction `glLoadIdentity()` permet quant à elle de réinitialiser la matrice courante à l'identité si nécessaire, sans quoi les instructions de projection ou de transformation sont combinées dans la matrice courante. Ce mécanisme sera étudié en détail un peu plus tard.

Finalement, `gluOrtho2D(d1,d1+Δ1,d2,d2+Δ2)` définit à la fois le type de projection et la zone du modèle de dessin que l'on veut projeter. L'instruction `gluOrtho2D(...)` définit une projection orthogonale pour un modèle en 2 dimensions, suivant une direction perpendiculaire au plan du modèle de dessin. Les points (`d1,d2`) et (`d1+Δ1,d2+Δ2`) définissent les coins inférieur gauche et supérieur droit de la zone à projeter, en coordonnées de dessin.

Effectuer une projection orthogonale telle qu'elle est décrite ci-dessus nécessite deux opérations : une translation et un changement d'échelle. La translation met en correspondance le coin inférieur droit de la zone à projeter avec celui de la viewport. Le changement d'échelle permet de passer des unités de dessin aux pixels.



Ce dernier entraînera une distorsion de l'image si le rapport entre la largeur et la hauteur de celle-ci, appelé *aspect ratio* et noté *w/h*, n'est pas respecté. C'est-à-dire si $(\Delta 1/\Delta 2) \neq (w/h)$. Imaginons par exemple que la largeur de la viewport soit deux fois plus grande que sa hauteur (w=2h) tandis que la largeur et la hauteur de la zone à projeter sont identiques ($\Delta 1=\Delta 2$). L'image subira alors un effet d'étirement dans la sens de la largeur.



4.3.Redimensionnement de la fenêtre d'écran par l'utilisateur

Lorsqu'une fenêtre d'écran est affichée, l'utilisateur a la possibilité de modifier ses dimensions, c'est-à-dire d'allonger ou de raccourcir la largeur et/ou la hauteur de la fenêtre. Ces redimensionnements sont gérés automatiquement par OpenGL, via un système classique de gestion d'évènements, l'utilisateur utilisant la souris pour indiquer ses desiderata. Pour que ceci soit possible, il faut que les instructions de définition des viewports et des projections se trouvent dans une même fonction, cette dernière étant passée en paramètre de l'instruction `glutReshapeFunc(...)` de la fonction `main(...)`. Cette fonction est appelée `reshape(...)` par convention, et reçoit deux paramètres entiers qui sont la largeur et la hauteur de la fenêtre d'écran, en pixels. Au départ, ces deux paramètres sont les dimensions initiales de la fenêtre, puis ils changent automatiquement chaque fois que l'utilisateur la redimensionne. Pour pouvoir utiliser ces dimensions dans le reste du code, leurs valeurs sont récupérées dans des variables globales. Le code se structure de la manière suivante :

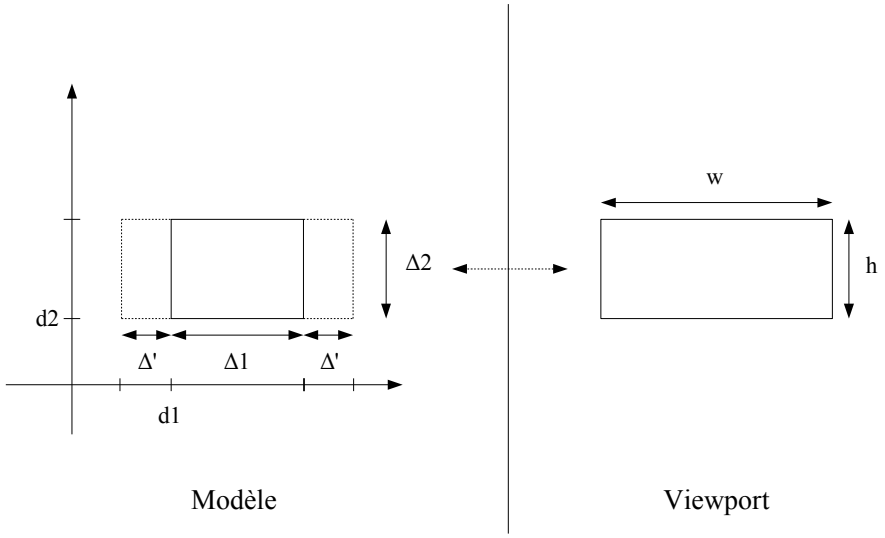
```
int main(int argc, char** argv)
{
   ...
   glutReshapeFunc(reshape);
   ...
}
...
static int w_global, h_global;
...
void reshape(int w, int h) // w et h sont la largeur et la hauteur de la
{                          // fenêtre d'écran (en callback)
   // récupération des dimensions de la fenêtre d'écran
   w_global=w;
   h_global=h;
}
```

La fonction `reshape()` est une fonction de type *callback* appelée chaque fois que l'utilisateur redimensionne la fenêtre. La fonction d'affichage de la fenêtre d'écran est appelée automatiquement après chaque appel de la fonction `reshape()`.

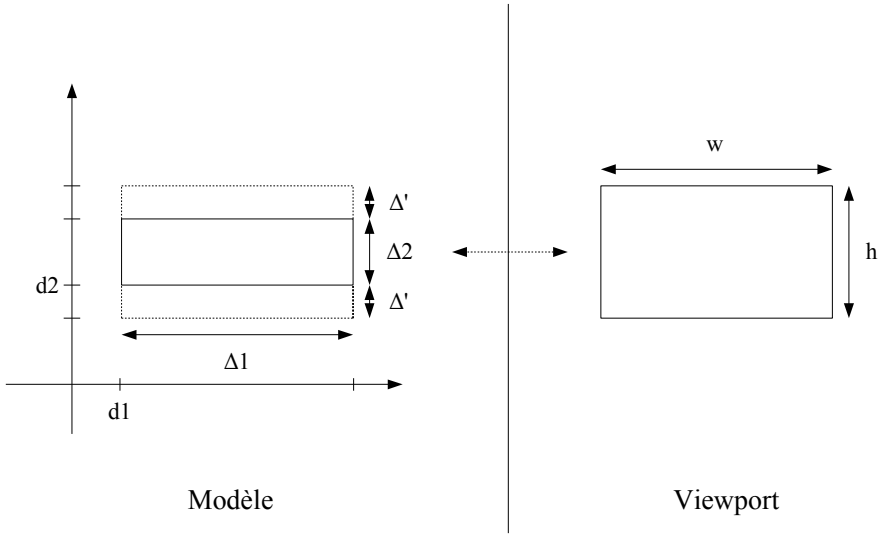
Le redimensionnement de la fenêtre par l'utilisateur peut lui aussi entraîner un problème de distorsion de ce qui est affiché. Imaginons que, tout comme dans l'exemple de la section 4.2, la largeur de la fenêtre soit doublée tandis que sa hauteur reste la même : les dimensions des viewports subiront le même changement et la zone du modèle à afficher subira un étirement dans le sens de la largeur sans que sa hauteur ne soit modifiée. L'utilisateur verra donc l'image s'étirer comme si la fenêtre était faite d'un matériau élastique, ce qui est rarement l'effet désiré ! Pour éviter cela, on s'arrange pour que l'*aspect ratio w/h* reste constant lors d'un redimensionnement de la fenêtre.

Une façon d'aborder le problème, c'est de décider qu'un redimensionnement de la fenêtre d'écran ne peut ni entraîner de distorsion, ni réduire la partie de la scène visible initialement. Pour cela, on décide que si le redimensionnement de la fenêtre entraîne une modification de l'*aspect ratio*, on augmentera la partie du modèle affichée à l'écran de telle sorte que ses dimensions correspondent au nouvel *aspect ratio*. Si $(w/h)>(\Delta 1/\Delta 2)$, on agrandira la dimension horizontale de la zone à projeter, et si $(w/h)<(\Delta 1/\Delta 2)$, on agrandira sa dimension verticale.

Envisageons d'abord le cas où $(w/h)>(\Delta 1/\Delta 2)$. Nous allons augmenter le $\Delta 1$ d'une quantité Δ' à gauche et à droite, de telle sorte que l'*aspect ratio* de la nouvelle zone à projeter corresponde à celui des nouvelles dimensions de la fenêtre d'écran : $\frac{\Delta 1+2\Delta'}{\Delta 2}=\frac{w}{h}$. On en déduit la valeur Δ' recherchée : $\Delta'=\frac{1}{2}\left(\frac{w}{h}\Delta 2-\Delta 1\right)$.



De manière équivalente, lorsque $(w/h)<(\Delta 1/\Delta 2)$ on augmente $\Delta 2$ d'une quantité Δ' vers le haut et vers le bas pour obtenir le nouvel *aspect* ratio : $\frac{\Delta 2+2\Delta'}{\Delta 1}=\frac{h}{w}$. On en déduit la valeur Δ' recherchée : $\Delta'=\frac{1}{2}\left(\frac{h}{w}\Delta 1-\Delta 2\right)$.



Le code ci-dessous traduit la solution dans le cas où la viewport posséde le même *aspect ratio* que la fenêtre d'écran :

```
void viewport()
{ // Le code ci-dessous ne tient pas compte des divisions entières !
  // w et h correspondent à w_global et h_global récupérés dans la
  // fonction reshape(...)
  ...
  if((w/h)>(Δ1/Δ2))
    gluOrtho2D(d1-1/2*((w*Δ2/h)-Δ1),d1+Δ1+1/2*((w*Δ2/h)-Δ1),d2,d2+Δ2);
  if((w/h)<=(Δ1/Δ2))
    gluOrtho2D(d1,d1+Δ1,d2-1/2*((h*Δ1/w)-Δ2),d2+Δ2+1/2*((h*Δ1/w)-Δ2));
  ...
  }
```

Lorsque la zone à afficher est carrée (Δ1=Δ2), les instructions ci-dessus se simplifient. Si de plus la zone à projeter est disposée symétriquement par rapport à l'origine, c'est-à-dire qu'elle va de (-a,-a) à (a,a) en coordonnées de dessin, la simplification est encore plus marquée. Voici ce que deviendrait le code dans ces deux cas de figure :

```
void viewport()
{ // Le code ci-dessous ne tient pas compte des divisions entières !
  // w et h correspondent à w_global et h_global récupérés dans la
  // fonction reshape(...)
  ...
  // Δ1=Δ2
  if(w>h) gluOrtho2D(d1-(w-h)*(Δ1/2)/h,d1+(w+h)*(Δ1/2)/h,d2,d2+Δ2);
  if(w<=h) gluOrtho2D(d1,d1+Δ1,d2-(h-w)*(Δ2/2)/w,d2+(h+w)*(Δ2/2)/w);
  ...
```

```
    // Δ1=Δ2 et zone symétrique par rapport à l'origine
    if(w>h) gluOrtho2D(-(w/h)*a,(w/h)*a,-a,a);
    if(w<=h) gluOrtho2D(-a,a,-(h/w)*a,(h/w)*a);
    ...
}
```

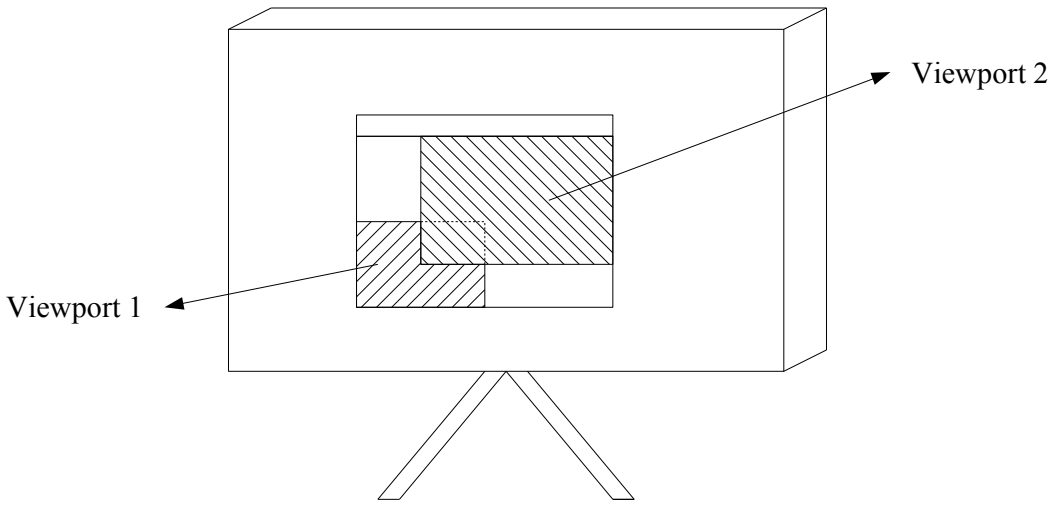
4.4.Principes de gestion de plusieurs viewports

Pour gérer plusieurs viewports simultanément dans la même fenêtre d'écran, il faut utiliser l'instruction `glViewport(...)` autant de fois que nécessaire avec les dimensions des différentes viewports désirées. Après chaque `glViewport(...)`, on appelera les instructions de projection pour la viewport courante. D'autre part, on appelera dans la fonction `display` les instructions qui caractérisent la viewport désirée avant les instructions relatives à la définition de ce qui doit être affiché dans chaque viewport. Si plusieurs viewports se recouvrent l'une l'autre, c'est la dernière viewport définie qui sera affichée à l'écran pour ce qui concerne la zone de conflit. Le code suivant donne un exemple de deux viewports se partageant la fenêtre d'écran :

```
static int w_global, h_global
...
void display(void)
{
   viewport1(w_global, h_global);
   ...
   // instructions d'affichage pour première viewport
   ...
   viewport2(w_global, h_global);
   ...
   // instructions d'affichage pour seconde viewport
}
...
void viewport1(int w, int h)
{
   glViewport(0,0,w/2,h/2);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(...);
   glMatrixMode(GL_MODELVIEW); // passage à la matrice des transformations
                               // géométriques
   glLoadIdentity();
}
```

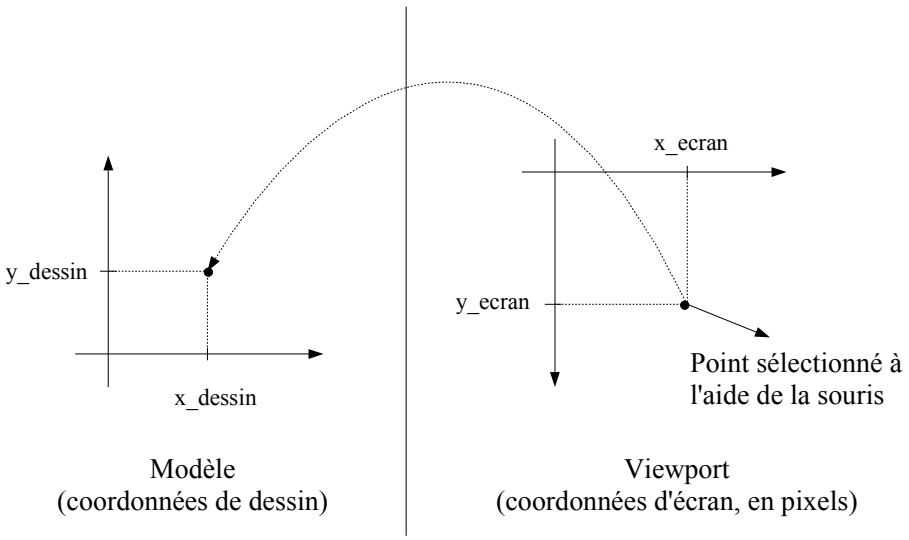
```
void viewport2(int w, int h)
{
   glViewport(w/4,h/4,3*w/4,3*h/4);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluOrtho2D(...);
   glMatrixMode(GL_MODELVIEW); // passage à la matrice des transformations
                               // géométriques
   glLoadIdentity();
   }
...
void reshape(int w, int h)
{
   // récupération des dimensions de la fenêtre d'écran
   w_global=w;
   h_global=h;
}
```

Dans l'exemple ci-dessus, une première viewport définit une zone rectangulaire du coin inférieur gauche au point situé au centre de la fenêtre d'écran. La seconde viewport définit un second rectangle du point (w/4,h/4) au coin supérieur droit de la fenêtre. Comme l'appel de `viewport2()` a lieu après celui de `viewport1()` dans la fonction `display()`, la zone commune aux 2 viewports (le carré allant de (w/4,h/4) à (w/2,h/2)) affichera ce qui est projeté dans `viewport2()`.



4.5.Sélection d'un object par inversion des coordonnées d'écran

Il existe une fonction *callback* gérée via la librairie *Glut* qui permet de récupérer les coordonnées d'écran lors d'une série d'évènements liés à l'utilisation de la souris. Ces coordonnées permettent de déterminer un objet sélectionné par l'utilisateur par un click de la souris. Pour cela, il faut déterminer les coordonnées de dessin qui correspondent aux coordonnées d'écran du point sélectionné. En fonction de ces coordonnées de dessin, on retrouve alors l'objet correspondant. Cette méthode est utilisable pour des modèle en deux dimensions car il existe alors une correspondance univoque entre le modèle et sa projection; par contre, il est beaucoup plus compliqué, lorsque c'est possible, d'utiliser ce principe lorsque le modèle est en trois dimensions. Nous verrons plus tard une méthode de sélection plus sophistiquée proposée par OpenGL pour gérer la sélection d'objets.



Dans cette section-ci, nous verrons d'abord la syntaxe de la fonction *callback* permettant de récupérer de l'information quant à des évènements liés à la souris. Ensuite, nous verrons comment retrouver l'information nécessaire à la transformation inverse de coordonnées d'écran en coordonnées de dessin.

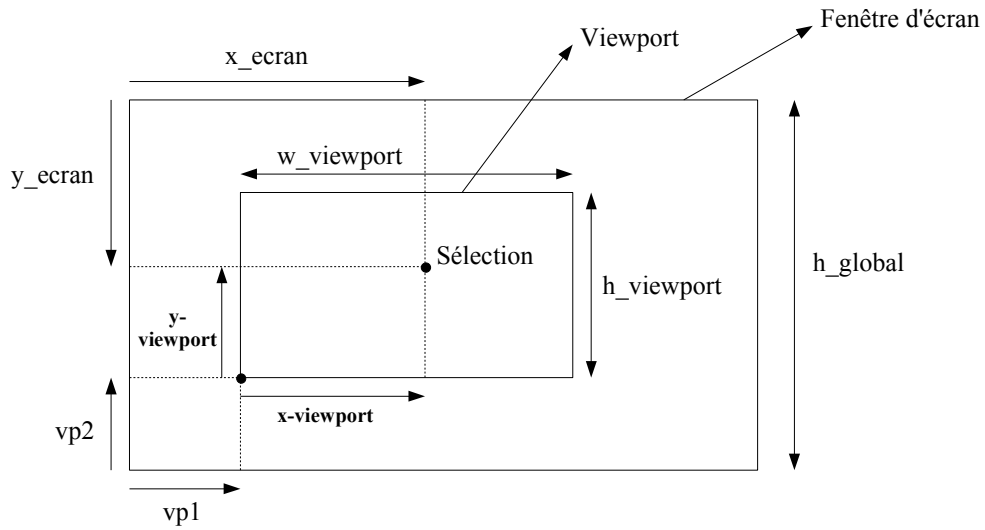
La fonction `glutMouseFunc()` reçoit un pointeur vers une fonction *callback* possédant 4 paramètres, qui permet de gérer les évènements liés à la souris. Le code se structure de la façon suivante :

```
int main(int argc, char** argv)
{
   ...
   glutMouseFunc(mouse);
   ...
}
...
```

```
void mouse(int button, int state, int x_ecran, int y_ecran)
{
   ...
   actions
   ...
}
```

Le premier paramètre (`button`) peut prendre les valeurs `GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` et `GLUT_RIGHT_BUTTON` selon le bouton activé. Le deuxième paramètre (`state`) peut prendre les valeurs `GLUT_UP` et `GLUT_DOWN` selon que le bouton reçoit une pression ou est relaché. Les deux derniers paramètres (`x_ecran` et `y_ecran`) indiquent l'emplacement de la souris en coordonnées d'écran (pixels) au moment de l'évènement. On peut donc définir différentes actions en fonction du type d'évènement et de la position de la souris au moment ou l'évènement a lieu.

Comment trouver les coordonnées de dessin correspondantes `x_dessin` et `y_dessin` ? Pour cela, il faut récupérer d'une part les paramètres définissant la viewport courante et d'autre part ceux qui définissent la partie du modèle qui est projetée dans cette viewport. Les coordonnées d'écran `x_écran` et `y_écran` sont alors transformées dans un premier temps en coordonnées relatives à la viewport, puis dans un second temps en coordonnées de dessin.



La récupération dans un vecteur des paramètres caractérisant la *dernière viewport définie* se fait via l'instruction `glGetIntegerv(...)`. Le code ci-dessous illustre cette récupération dans un vecteur appelé `viewport` :

```
GLint viewport[4];
glGetIntegerv(GL_VIEWPORT, viewport);
```

Les deux premiers éléments du tableau à une dimension `viewport` donnent l'abscisse et l'ordonnée du coin inférieur gauche de la viewport `vp1` et `vp2` (cf. section 4.2). Les deux éléments suivants donnent la largeur et la hauteur de la viewport, que nous noterons ici `w_viewport` et `h_viewport`. On récupère aussi les caractéristiques de la portion du modèle

projetée dans la viewport, soient `d1` et `d2` l'abscisse et l'ordonnée de son coin inférieur gauche en coordonnées de dessin, et `Δ1` et `Δ2` sa largeur et sa hauteur. Celles-ci sont données par exemple par les formules utilisées dans l'instruction `gluOrtho2D(...)` de la fonction `viewport()`. Dans l'exemple de la section 4.2, on avait simplement `gluOrtho2D(d1,d1+Δ1,d2,d2+Δ2)`.

Soient `x_viewport` et `y_viewport` les coordonnées de position de la souris en nombres de pixels vers la droite et vers le haut à partir du coin inférieur gauche de la viewport, et soit `h_global` la hauteur de la fenêtre d'écran disponible via la fonction `reshape`, on a :

`x_viewport = x_ecran – vp1,`

`y_viewport = h_global - vp2 - y_ecran` (on passe à des coordonnées vers le haut).

On obtient finalement les coordonnées de dessin en utilisant la correspondance entre la portion de modèle qui est projetée et la viewport. Cela nécessite, comme annoncé en fin de section 4.2, une translation de `d1` suivant *x* et de `d2` suivant *y*, précédée d'un changement d'échelle de `(Δ1/w_viewport)` en *x* et de `(Δ2/h_viewport)` en *y* :

`x_dessin = d1 + x_viewport*(Δ1/w_viewport),`

`y_dessin = d2 + y_viewport*(Δ2/h_viewport).`

Il suffit alors de comparer (`x_dessin,y_dessin`) avec la définition de la scène pour décider si un objet donné a été sélectionné par l'utilisateur.

5. Transformations géométriques en 2D

La translation et la rotation d'objets en deux dimensions a déjà été introduites dans la section 3.4. Dans ce chapitre-ci, nous verrons un troisième type de transformation, la transformation d'échelle. Nous introduirons ensuite le système de représentation matricielle utilisé par OpenGL pour gérer les transformations géométriques, ainsi que la façon de combiner entre elles plusieurs transformations par leur application successive à un même objet.

Pour rappel, toute instruction de transformation géométrique est appelée avant l'instruction qui définit l'objet auquel la transformation s'applique. La translation d'un objet utilise l'instruction `glTranslated(Δx,Δy,Δz)`, qui a pour effet de le déplacer du delta (Δ) spécifié le long de chacun des 3 axes. Pour faire pivoter un objet, on utilise l'instruction `glRotated(angle,x,y,z)`, qui a pour effet de lui appliquer une rotation de `angle` degrés dans le sens anti-horlogique autour de l'axe passant par l'origine du repère et par le point (`x,y,z`). Un `angle` négatif correspond à une rotation dans le sens horlogique.

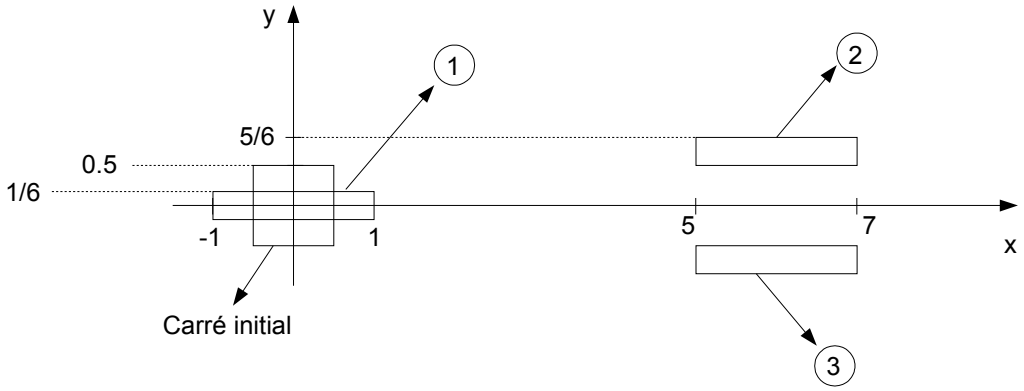
5.1. La transformation d'échelle

L'instruction `glscaled(Sx,Sy,Sz)` permet d'appliquer à un objet une transformation d'échelle de facteurs `Sx` , `Sy` et `Sz` suivant chacun des axes, respectivement. L'effet de cette transformation est de multiplier chacune des coordonnées de dessin par le facteur correspondant. Lorsqu'un facteur est négatif, la transformation d'échelle se double d'une réflexion par rapport au plan constitué par les deux autres axes. Lorsque le centre de gravité de l'objet correspond à l'origine du repère, on se ramène à un changement d'échelle classique : l'objet est allongé ou rétréci d'un facteur donné suivant chaque axe.

Exemple : dans le code ci-dessous, la fonction `carréUnité()` construit un carré unitaire centré à l'origine du repère. Trois cas de figures sont proposés : dans le premier, le carré subit un doublement suivant l'axe *x* et est rétréci des deux tiers suivant l'axe *y* ; le deuxième translate d'abord le carré de 2 unités vers le haut et 3 unités vers la droite avant de lui appliquer une transformation d'échelle de facteurs 2 et 1/3 suivant *x* et *y*, respectivement ; dans le troisième cas, le carré subit la même translation que dans le deuxième, puis subit une transformation d'échelle de facteurs 2 et -1/3 suivant *x* et *y*, respectivement. Remarquons que dans les cas 2 et 3 la translation est appelée après le changement d'échelle ; nous verrons dans la section 5.3 ce qu'il se passe si les deux transformations sont appelées dans l'ordre inverse.

```
void carréUnité(void)
  {
    glPolygonMode(GL_FRONT;GL_LINE); //mode fil de fer
    glColor3d(0.0,0.0,0.0);
    glRect(-0.5,-0.5,0.5,0.5); //définition du carré
  }
  ...
  glPushMatrix();
  glScaled(2.0,1/3.0,0.0);
  carréUnité(); // cas de figure 1
```

```
glTranslated(3.0,2.0,0.0);
carréUnité(); // cas de figure 2
glPopMatrix();
glPushMatrix();
glScaled(2.0,-1/3.0,0.0);
glTranslated(3.0,2.0,0.0);
carréUnité(); // cas de figure 3
...
```



Voyons à présent comment l'information relative à chaque type de transformation est stockée sous forme matricielle.

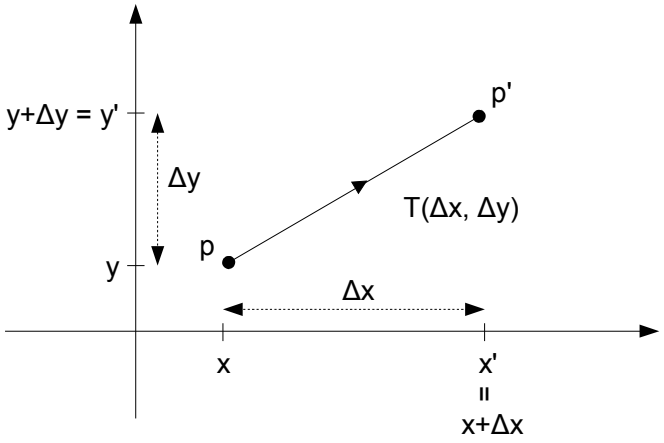
5.2.Représentation matricielle

5.2.1.Translation

On représente d'une part un point quelconque P de coordonnées x et y par le vecteur $P=\begin{bmatrix} x \\ y \end{bmatrix}$, et une translation de Δx suivant l'axe x et de Δy suivant l'axe y peut se représenter d'autre part par un vecteur $T=\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix}$. Tout point P subissant la translation T deviendra le point P', tel que défini ci-dessous, après transformation :

$$P' = P + T$$

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} x + \Delta x \\ y + \Delta y \end{bmatrix}$$

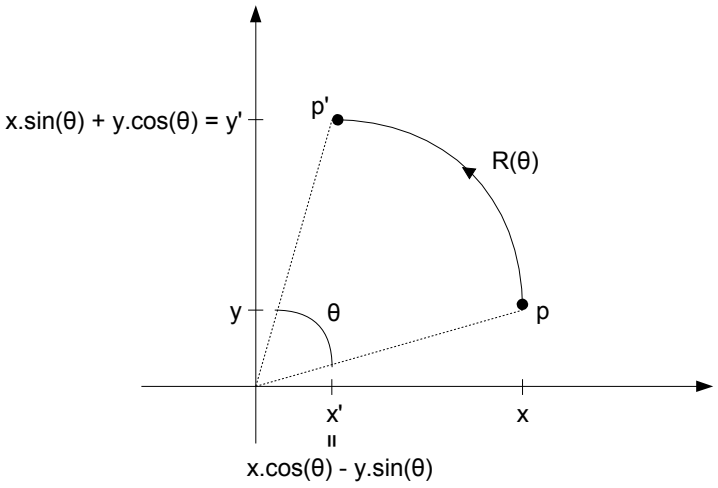


5.2.2.Rotation autour de l'origine du repère

Lorsqu'un point subit une rotation d'un angle θ autour de l'axe *z*, la transformation prend la représentation matricielle $R=\begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}$, où un θ positif correspond à une rotation dans le sens anti-horlogique. Tout point *P* subissant la rotation *R* deviendra le point *P'*, tel que défini ci-dessous, après transformation :

$$P' = R.P$$

$$P' = \begin{bmatrix}x' \\ y'\end{bmatrix} = \begin{bmatrix}\cos\theta & -\sin\theta \\ \sin\theta & \cos\theta\end{bmatrix}.\begin{bmatrix}x \\ y\end{bmatrix} = \begin{bmatrix}x.\cos\theta - y.\sin\theta \\ x.\sin\theta + y.\cos\theta\end{bmatrix}$$

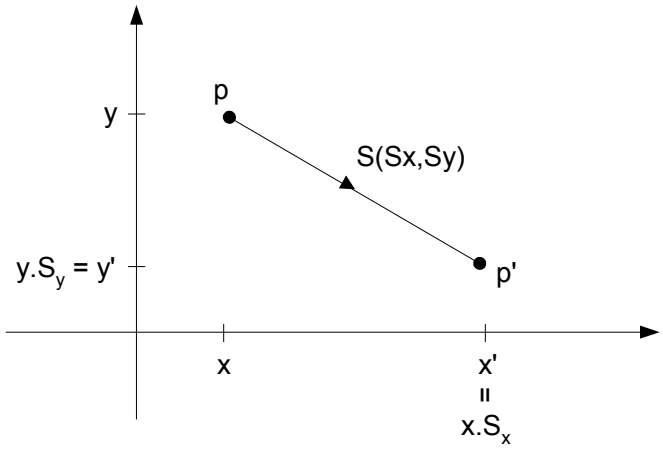


5.2.3.Transformation d'échelle (*scaling*)

Lorsqu'un point subit une transformation d'échelle de facteurs *Sx* suivant l'axe *x* et *Sy* suivant l'axe *y*, la transformation prend la représentation matricielle $S=\begin{bmatrix}Sx & 0 \\ 0 & Sy\end{bmatrix}$. Tout point *P* subissant la transformation d'échelle *S* deviendra le point *P'*, tel que défini ci-dessous, après transformation :

$$P' = S.P$$

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix} . \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x.Sx \\ y.Sy \end{bmatrix}$$



5.2.4.Coordonnées homogènes

Suivant les définitions ci-dessus, la translation se traduit par une somme *(P' = P + T)* tandis que la rotation et la transformation d'échelle se traduisent par des produits (*P' = R.P* et *P' = S.P*, respectivement). Afin que la translation puisse se traduire aussi par un produit, on introduit les coordonnées homogènes : un point quelconque *P* dans un repère à 2 dimensions sera représenté par un vecteur de longueur 3, ou la 3ième coordonnée est égale à 1 : $P = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$. On peut alors définir la matrice de translation $T = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix}$ et le point *P'* après transformation s'obtient en effectuant le produit *T.P*. On a dès lors pour chacune des trois transformations élémentaires en coordonnées homogènes :

- Translation :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = T(\Delta x ; \Delta y) \; . \; P$$

- Rotation :

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = \quad R(\theta) \quad . \; P$$

- Tranformation d'échelle :

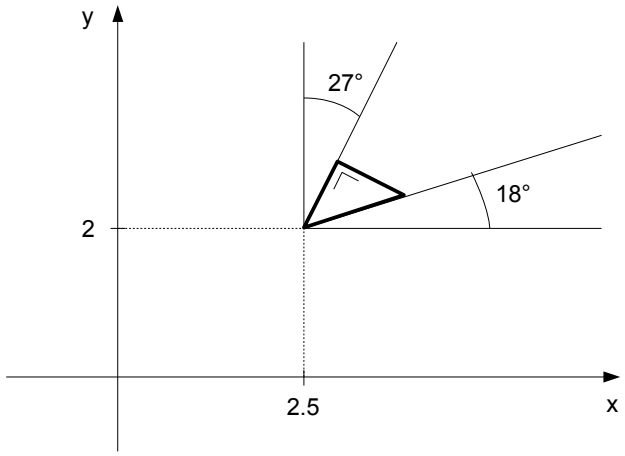
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = S(Sx;Sy) \ . \ P$$

L'introduction des coordonnées homogènes permet donc de représenter toute transformation élémentaire par un produit matriciel. En pratique, OpenGL travaille avec des coordonnées homogènes de dimension 4 : *x*, *y* et *z* plus une 4ième coordonnée égale à 1.0 par défaut.

5.3. Composition de plusieurs transformations

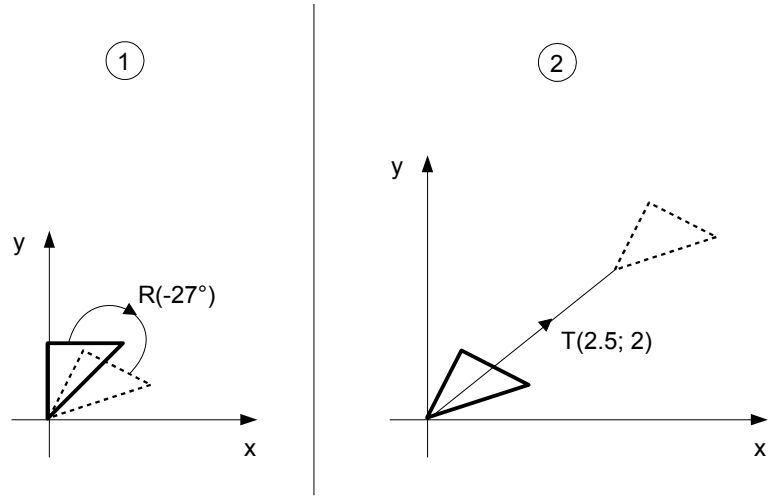
Les transformations géométriques permettent de déplacer un objet afin de le positionner à l'endroit voulu de la scène. Pour ce faire, un seul type de transformation élémentaire sera le plus souvent insuffisant : il faudra appliquer au minimum 2 transformations de types différents (par exemple une rotation suivie d'une translation) et parfois plus.



Imaginons que l'on veuille dessiner un triangle rectangle isocèle dont l'hypoténuse est de longueur $\sqrt{2}$, de sorte que l'angle droit soit orienté vers le haut, que l'hypoténuse forme un angle de 18° avec l'horizontale et que le sommet inférieur gauche coïncide avec le point (2.5;2) du repère. Pour le cerveau humain, une façon simple de procéder sera de définir un triangle identique de sommets (0;0), (1;1) et (0;1) puis de lui appliquer une rotation de 45-18=27° dans le sens horlogique (*R(-27°)*), suivie d'une translation de 2.5 unités suivant *x* et de 2 unités suivant *y* (*T(2.5;2)*), pour l'amener à l'endroit désiré.

Cela se traduit sous forme matricielle par *P'=T(2.5;2).R(-27°).P*, ou encore :

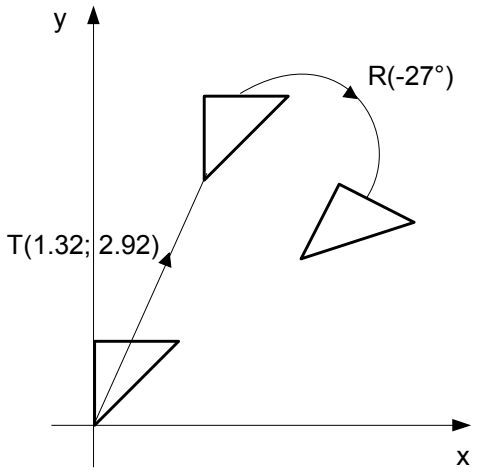
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 2.5 \\ 0 & 1 & 2.0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} \cos(-27°) & -\sin(-27°) & 0 \\ \sin(-27°) & \cos(-27°) & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$



Il est important de remarquer que la première transformation (*R(-27°)*) se trouve directement à gauche de *P* dans le produit matriciel, la translation venant elle-même à gauche de la rotation. Dans le code OpenGL, la rotation apparaîtra juste avant l'appel du triangle initial et la translation se placera avant la rotation ; ou encore la translation appraîtra la première dans le code, suivie de la rotation, suivie finalement de l'appel du triangle initial. Le code en sera :

```
...
void dessineTriangle(void)
{
   glBegin(GL_TRIANGLES);
      glVertex2d(0.0,0.0);
      glVertex2d(1.0,1.0);
      glVertex2d(0.0,1.0);
   glEnd();
}
...
void display(void)
{
   ...
   glTranslated(2.5,2.0,0.0);
   glRotated(-27.0,0.0,0.0,1.0);
   dessineTriangle();
   ...
}
```

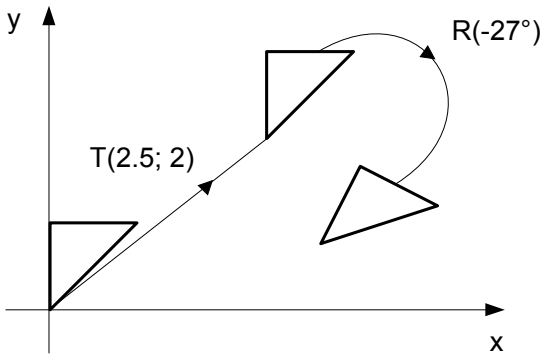
Que se passe-t'il si l'on décide d'appliquer d'abord une translation, suivie d'une rotation? On pourra par exemple utiliser la translation *T(2.5*cos(27°) - 2*sin(27°);2.5*sin(27°) + 2*cos(27°))* suivie de la rotation *R(-27°)*. On réalise que cette approche est moins élégante, et même à rejeter au profit de la première solution qui ne requiert pas les 4 appels à des fonctions trigonométriques.



Cette deuxième solution se traduirait au niveau du code OpenGL par :

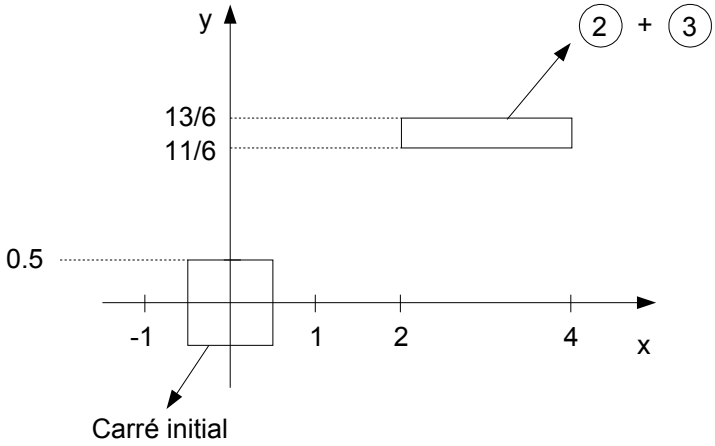
```
void init(void)
{
   ...
   cos27 = cos(M_PI*27.1/180);
   sin27 = sin(M_PI*27.1/180);
}
...
void display(void)
{
   ...
   glRotated(-27,0.0,0.0,1.0);
   glTranslated(2.5*cos27-2.0*sin27,2.5*sin27+2.0*cos27,0.0);
   dessineTriangle();
   ...
}
```

Inverser la rotation et la translation de la première solution, c'est-à-dire effectuer la transformation *P'=R(-27°).T(2.5;2).P*, ne donne pas le résultat recherché mais bien celui illustré ci-dessous :



De manière générale, deux transformations élémentaires *M1* et *M2* ne sont pas commutatives : appliquer d'abord *M1* suivie de *M2* ne donne pas le même résultat que *M2* suivie de *M1*, ou encore $M2.M1.P \neq M1.M2.P$. Il existe cependant 4 cas dans lesquels *M1* et *M2* seront commutatives : si elles sont toutes les deux du même type (*T*, *R* ou *S*) ou encore si l'une est une rotation et l'autre une transformation d'échelle telle que *Sx=Sy*.

Illustrons une seconde fois cette non commutativité en reprenant l'exemple de la section 5.1. Si l'on permutte la translation et la transformation d'échelle dans les cas de figure 2 et 3, on obtient *P'=T(3.0;2.0).S(2.0;0.33).P* et *P'=T(3.0;2.0).S(2.0;-0.33).P*, respectivement. Les deux primitives se superposent, pour un résultat bien différent de celui obtenu en section 5.1 avec *P'=S(2.0;0.33).T(3.0;2.0).P* et *P'=S(2.0;-0.33).T(3.0;2.0).P*.



5.4.La matrice de "modelview"

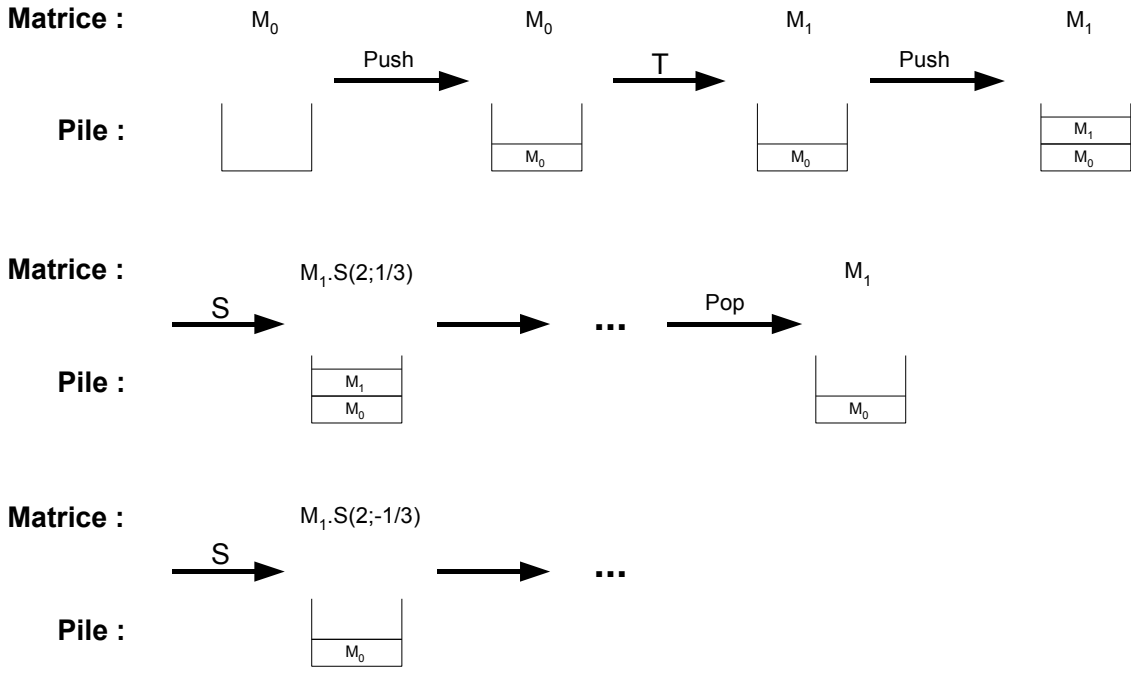
Comment OpenGL procède-t'il en pratique pour gérer les transformations demandées? La réponse a été évoquée dans la setion 4.2 : OpenGL utilise une matrice dite de "*modelview*" dans laquelle sont enregistrées les transformations géométriques successives. Au départ, cette matrice est initialisée à l'identité (matrice contenant des 1 sur sa diagonale et des 0 ailleurs) par l'appel de l'instruction `glLoadIdentity()`. Ensuite, chaque fois qu'une instruction du type `glTranslated`, `glRotated` ou `glScaled` est rencontrée, la matrice de *modelview* est multipliée à droite par la matrice de transformation géométrique correspondante. Enfin, chaque fois qu'une primitive graphique est rencontrée, les sommets de celle-ci sont transformés par multiplication à droite de la matrice de *modelview* par le vecteur *P* correspondant. Ces différentes opérations concernent le modèle uniquement, les calculs relatifs à la projection vers la fenêtre d'écran ayant lieu par la

suite.

La matrice de *modelview* peut à tout moment être stockée dans une pile, via l'instruction `glPushMatrix()`. A l'inverse, la matrice de *modelview* peut aussi être remplacée par la matrice se trouvant au sommet de cette pile via l'instruction `glPopMatrix()`. On peut ainsi récupérer la matrice dans un état antérieur lorsque cela permet de faire l'économie d'une ou plusieurs transformations géométriques. La profondeur de la pile varie d'un système et d'une version d'OpenGL à une autre, et peut être obtenue via l'instruction `glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, GLint *depth)`. Elle est au minimum de 32 pour la matrice de *modelview*.

Afin d'illustrer le fonctionnement de cette pile, analysons le code relatif au dernier exemple de la section 5.3, qui se présente comme suit :

```
...
glPushMatrix();
glTranslated(3.0,2.0,0.0);
glPushMatrix();
glScaled(2.0,0.33,0.0);
carréUnité(); // cas de figure 2
glPopMatrix();
glScaled(2.0,-0.33,0.0);
carréUnité(); // cas de figure 3
...
```



Le premier glPushMatrix() stocke l'état de la matrice avant la translation, soit M_0. Le second `glPushMatrix()` stocke le résultat du produit de M_0 par $T(3.0;2.0)$: $M_1=M_0.T(3.0;2.0)$. La matrice de *modelview* est ensuite multipliée par $S(2.0;0.33)$ avant d'être appliquée au premier carré. Ensuite, `glPopMatrix()` remplace la matrice de *modelview* courante par M_1 avant

d'appliquer la seconde transformation d'échelle, ce qui évite de répéter l'instruction de translation (identique pour les deux carrés). A la fin du code, la matrice de *modelview* contient $M_1.S(2.0; -0.33)$ tandis que M_0 est disponible en haut de la pile.

Remarque : il faut faire *attention de ne pas dépasser la capacité de la pile car OpenGL ne donne aucun avertissement dans ce cas, se contentant de laisser la pile en état, sans effectuer les* `glPushMatrix()` *demandés !*

5.5. Cisaillement (shearing)

A compléter.

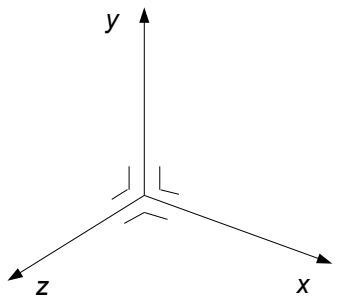
6. Modèle en 3D : définition, transformations et projections

La généralisation à des modèles en 3 dimensions est assez simple en ce qui concerne la définition de la scène et les transformations géométriques. Par contre, la restitution à l'écran d'une zone d'un modèle 3D est plus compliquée : il faut définir la partie de la scène que l'on veut projeter à l'écran (qui est maintenant un volume en 3D), le type de projection à utiliser pour représenter ce volume sur un support 2D et la position de l'oeil du spectateur par rapport à la scène.

Après une rapide généralisation en 3D de ce qui a déjà été discuté en 2D, ce chapitre donnera une définition succinte de ce qui caractérise une projection d'un volume 3D sur un support 2D. Différents types de projections simples seront abordés, ainsi que les différentes étapes du processus pratique de projection. Finalement, le modus operandi d'OpenGL sera expliqué.

6.1. Une coordonnée supplémentaire

Pour pouvoir travailler en 3D, il suffit d'introduire une nouvelle coordonnée (notée *z*) qui ajoutée à *x* et *y* permettra de définir de manière univoque tout point de l'espace par rapport au système de coordonnées (*x*,*y*,*z*). Nous travaillons avec un système *orthonormé dextrogyre* : le vecteur unitaire est de même longueur suivant chaque axe, les 3 axes forment l'un avec l'autre des angles de 90 degrés et la direction des *z* positifs est définie comme la direction d'enfoncement d'un tire-bouchon pour droitier lorsqu'il effectue une rotation des *x* positifs vers les *y* positifs.



Les coordonnées homogènes sont dès lors de dimension 4, la quatrième étant notée *w* et de valeur égale à 1.0 par défaut. L'instruction de définition d'un sommet d'une primitive graphique devient `glVertex3d(x,y,z)`. La surface d'un objet 3D se construit à l'aide des primitives graphiques introduites au chapitre 2, les sommets de celles-ci pouvant être définis dans l'espace 3D. Les sommets relatifs à un même quadrilatère ou polygone doivent cependant se trouver dans un même plan, et toujours définir une forme convexe.

Les transformations géométriques répondent aux mêmes principes qu'en 2D, en tenant compte des possibilités offertes par la coordonnée supplémentaire *z* :

- Translation :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = T(\Delta x ; \Delta y ; \Delta z) \; . \; P$$

- Rotation anti-horlogique autour de l'axe z :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_z(\theta) \; . \; P$$

- Rotation anti-horlogique autour de l'axe x :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_x(\theta) \; . \; P$$

- Rotation anti-horlogique autour de l'axe y :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = R_y(\theta) \; . \; P$$

- Tranformation d'échelle :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$P' = S(Sx ; Sy ; Sz) \; . \; P$$

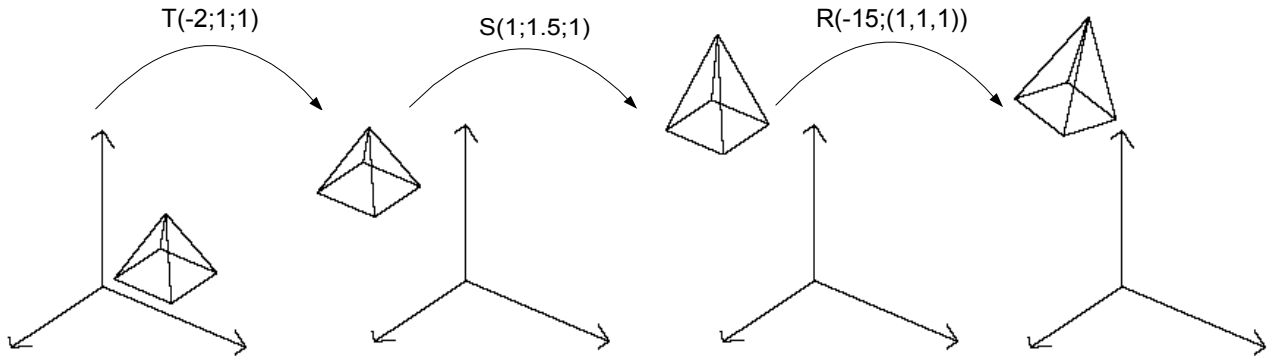
Une rotation autour d'un axe quelconque se décompose toujours en un maximum de 5 rotations "élémentaires" telles que définies ci-dessus. Le principe est de faire d'abord correspondre l'axe de rotation avec l'un des axes du système de coordonnées (ce qui requiert un maximum de 2 rotations), ensuite d'opérer une rotation d'angle et de sens requis autour de cet axe principal, et enfin de procéder à la tranformation inverse de celle qui a permis d'obtenir la correspondance avec un des axes du système de coordonnées.

En OpenGL, *T(Δx,Δy,Δz)* se traduit par `glTranslated(Δx,Δy,Δz)` et *S(Sx*,S*y*,S*z*) par `glScaled(Sx,Sy,Sz)`. La rotation anti-horlogique d'un angle θ, exprimé en degrés, autour d'un axe quelconque dont la direction est indiquée par le vecteur (*x*,*y*,*z*) se traduit quant à elle par `glRotated(θ,x,y,z)`.

L'exemple ci-dessous fait subir les transformations suivantes à une pyramide régulière : *T(-2;1;1)* suivie de *S(1;1.5;1)* suivie finalement d'une rotation de 15° dans le sens horlogique autour de l'axe défini par le vecteur (*1*,*1*,*1*). La rotation est programmée de deux façons : d'une part en indiquant directement l'axe de rotation quelconque dans `glRotated()` et d'autre part en la décomposant suivant le principe exposé plus haut.

```
void pyramide(void)
{
  glPolygonMode(GL_FRONT_AND_BACK;GL_LINE); //mode fil de fer
  glColor3d(0.0,0.0,0.0);
  glBegin(GL_TRIANGLE_FAN);//définition des 4 faces avec le côté visible
                           //vers l'extérieur
    glVertex3d(1.5,2.0,0.5);
    glVertex3d(1.0,1.0,0.0);
    glVertex3d(1.0,1.0,1.0);
    glVertex3d(2.0,1.0,1.0);
    glVertex3d(2.0,1.0,0.0);
    glVertex3d(1.0,1.0,0.0);
  glEnd();
  //définir la base n'est pas nécessaire si on travaille en mode fil de fer
}
...
glPushMatrix();
rotation();
glScaled(1.0,1.5,1.0);
glTranslated(-2.0,1.0,1.0);
pyramide();
glPopMatrix();
...
void rotation(void)
{
  //soit première possibilité :
  glRotated(-15.0,1.0,1.0,1.0);

  //soit deuxième possibilité, utilisant uniquement des rotations autour
  des axes définissant le repère :
  // 5. opère la transformation inverse de 1.
  glRotated(-45.0,0.0,1.0,0.0);
  // 4. opère la transformation inverse de 2.
  glRotated(-45.0,0.0,0.0,1.0);
```

```
    // 3. opère une rotation horlogique de 15° autour de l'axe y
    glRotated(-15.0,0.0,1.0,0.0);
    // 2. fait correspondre le vecteur (1,1,0) avec l'axe y
    glRotated(45.0,0.0,0.0,1.0);
    // 1. amène le vecteur (1,1,1) dans le plan (x,y) -> (1,1,0)
    glRotated(45.0,0.0,1.0,0.0);
}
```



La deuxième possibilité proposée dans la fonction `rotation()` ci-dessus n'est bien sur pas l'unique façon de travailler avec des rotations "élémentaires". Elle est proposée ici dans un but illustratif ; en pratique la solution la plus compacte doit être privilégiée, sans oublier de toujours commenter le code de manière synthétique et claire.

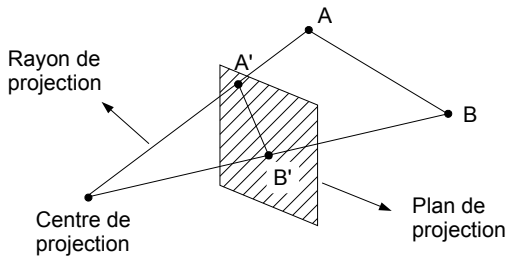
6.2. Caractéristiques d'une projection d'un volume 3D en 2D

On désire projeter une portion de la scène 3D sur une surface plane, le plan de projection 2D. Nous appellerons dans la suite cette portion le volume à projeter. La projection elle-même associe à chaque point du volume à projeter un point sur le plan de projection, de sorte que l'on obtienne la représentation 2D voulue du volume 3D.

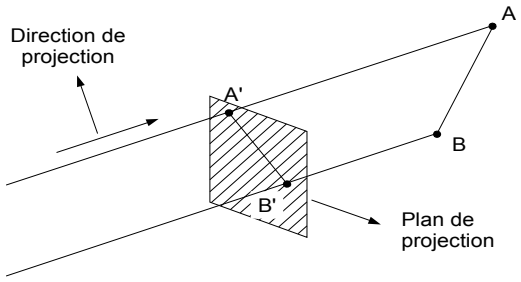
Toute projection est caractérisée par un ensemble de rayons de projection qui relient un centre de projection unique à chaque point du volume à projeter. Chaque rayon de projection possède un point d'intersection avec le plan de projection, et l'ensemble de ces intersections forme le résultat de la projection.

Il n'y a pas de contrainte particulière concernant l'emplacement du plan de projection : il peut se situer à l'avant ou à l'arrière du volume à projeter par rapport au centre de projection, ou même couper le volume. Sa situation et son inclinaison dans l'espace influenceront le résultat.

Il existe deux familles principales de projections, selon que le centre de projection se situe à distance finie ou infinie du volume à projeter. Dans le cas d'un centre de projection à distance finie, on parle de projection en *perspective*, et dans le cas d'un centre de projection à distance infinie (voir section 6.2.2), on parle de projection *parallèle*.



Projection en perspective



Projection parallèle

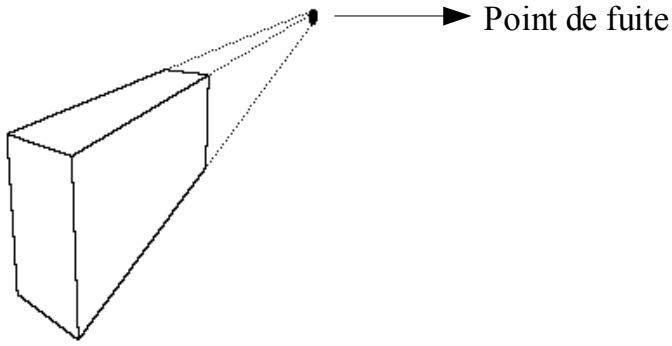
6.2.1.Projection en perspective

Lorsque l'on utilise ce type de projection, la dimension d'un objet dans le plan de projection est inversément proportionnelle à la distance de l'objet au centre de projection. Ainsi, plus un objet est éloigné, plus son apparence sera petite, ce qui confère aux projections en perpective un effet réaliste, se rapprochant de ce que le cerveau humain a l'habitude de percevoir via le système oculaire.

Les angles du volume de départ ne sont pas conservés en projection, sauf s'ils appartiennent à un plan parallèle au plan de projection. De même, des droites parallèles le restent uniquement si elles appartiennent à des plans parallèles au plan de projection. Dans le cas contraire, des droites parallèles entre elles se rejoignent toutes en un même point, appelé *point de fuite* associé à leur direction.

Lorsqu'un point de fuite correspond à la direction d'un axe principal (x, y ou z), on parle de *point de fuite principal*. Il y a entre 1 et 3 points de fuite principaux, selon le nombre d'axes principaux qui ont un point d'intersection avec le plan de projection.

Projection en perspective



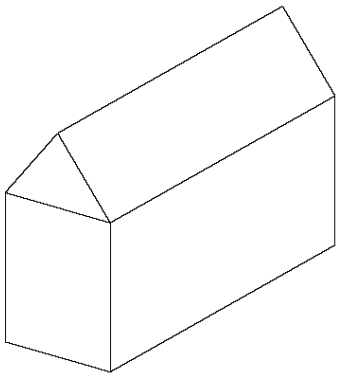
On peut donc reconnaître une projection en perspective à l'effet de perspective d'une part (plus un objet est éloigné, plus sa dimension apparente est petite), et à la présence de points de fuite d'autre part.

6.2.2.Projection parallèle

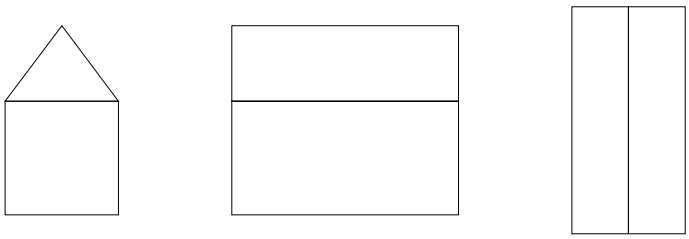
Lorsque les rayons de projection sont parallèles entre eux, suivant ce que l'on appelle la *direction de projection*, on parle de projection parallèle. Cela revient à considérer que le centre de projection se trouve à distance infinie du volume à projeter ; en effet, des droites parallèles en géométrie euclidienne se rejoignent à l'infini en géométrie projective. Lors d'une projection parallèle, des parallèles dans le volume de départ restent parallèles en projection, mais les angles finis ne sont pas conservés à l'exception de ceux qui appartiennent à des plans parallèles au plan de projection.

Une projection parallèle est dite *orthographique* si le plan de projection est perpendiculaire à la direction de projection. Dans le cas contraire, la projection parallèle est dite *oblique*. La projection orthographique est la plus utilisée en pratique, et la seule projection parallèle proposée par OpenGL. Elle est moins "réaliste" que la projection en perspective, mais elle permet de prendre des mesures à l'échelle suivant certaines directions, d'où son intérêt dans une série de domaines.

Projection parallèle



En dessin industriel, on représente couramment un objet à l'aide de trois "vues" : une vue de face, une vue de côté et une vue du haut. Ces trois vues sont en fait des projections orthographiques de l'objet. Pour une orientation bien choisie de l'objet, elles permettent de connaître toutes les distances et les angles nécessaires à la représentation de l'objet en 3D.



Vue de face

Vue de coté

Vue du haut

6.2.3.Les étapes de construction d'une projection

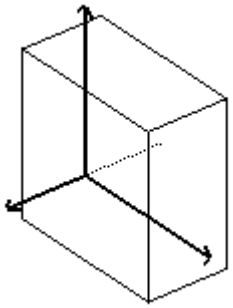
Comment, à partir d'un point en coordonnées de dessin, obtient-on un point résultant de la projection désirée en coordonnées d'écran?

Remarquons avant de continuer que seuls les sommets des primitives graphiques sont transformés, les sommets résultants étant ensuite reliés entre eux suivant les mêmes primitives que les sommets originaux.

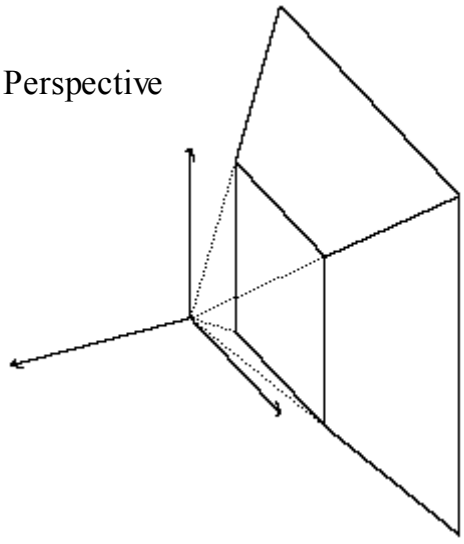
La transformation complète permettant d'obtenir une image dont l'affichage à l'écran corresponde à la représentation recherchée du volume à projeter peut se décrire comme une succession de tranformations similaires aux transformations géométriques. On peut décomposer le processus en étapes et associer à chacune d'elle une opération matricielle : tout comme pour les transformations géométriques, les calculs nécessaires font appels à des multiplications matricielles. Nous ne rentrerons pas dans les détails de ces calculs (cela déborderait du cadre du cours), mais nous allons décrire brièvement 5 étapes permettant d'appréhender la marche à suivre :

1. Définir le centre ou la direction de projection, ainsi que le plan de projection. Définir les dimensions d'un rectangle sur le plan de projection, tel que seules les intersections des rayons de projection avec la zone délimitée par ce rectangle doivent être prises en compte pour construire la projection. Définir aussi 2 plans parallèles au plan de projection, tels que seuls les objets compris entre ces deux plans doivent être projetés. L'ensemble de ces informations détermine le volume à projeter.
2. Transformer le volume défini à l'étape 1 en un *volume canonique de vue*, c'est-à-dire un volume délimité par les plans suivants :
   - $x=-1, x=1, y=-1, y=1, z=0, z=-1$, c'est-à-dire un parallèlépipède rectangle, avec l'axe z qui correspond à la direction de projection, pour une projection parallèle;
   - $x=-z, x=z, y=-z, y=z, z=-z_{min}, z=-1$, c'est-à-dire une pyramide tronquée, dont la base est perpendiculaire à la direction de l'axe z et parallèle au plan de projection et dont le sommet, situé à l'origine du repère, est le centre de projection, pour une projection en perspective.

Parallèle



Perspective



3. Eliminer tout ce qui se trouve en dehors du volume canonique de vue, c'est-à-dire tout ce qui n'appartient pas au volume à projeter.
4. Projeter l'intérieur du volume canonique de vue sur le plan de projection suivant le type de projection choisi. Le résultat se retrouvera à l'intérieur du rectangle défini à l'étape 1.
5. Etablir une correspondance (un mapping) entre le rectangle contenant le résultat de la projection et la viewport active.

L'étape 2 a pour principal intérêt de permettre un traitement très efficace à l'étape 3 qui lui succède, ce qui ne serait souvent pas le cas si l'on ne passait pas par un volume canonique de vue. Si l'on excepte quelques situations très particulières, le temps passé à l'étape 2 est largement récupéré à l'étape 3.

6.3.Le modus operandi en OpenGL

6.3.1.Introduction

Tout comme il existe une matrice de *modelview* permettant de gérer les transformations géométriques, il existe en OpenGL une matrice de projection via laquelle sont effectués les calculs relatifs au processus de projection. Lorsque l'on maîtrise les mécanismes sous-jacents, on peut travailler directement sur cette matrice, cependant la majorité des utilisateurs désirent éviter cette approche relativement fastidieuse. C'est pourquoi OpenGL propose 2 types de projection pour lesquelles il suffit de spécifier quelques paramètres simples, le reste du travail s'opérant de manière automatique.

Dans la suite, nous ne considérerons que ces deux projections "assistées". Leur définition nécessite deux étapes :

- d'abord le choix du point de vision, emplacement de l'oeil de l'observateur, et celui d'un point de la scène qui, en conjonction avec le premier, définit la ligne de vue. Le point de vision sera aussi le centre de projection en *perspective*, et la ligne de vue définira la direction de projection en *parallèle*. Il faut aussi spécifier l'orientation de la scène dans cette première étape, c'est-à-dire donner la direction qui pointera vers le haut;
- dans un second temps, on précisera le volume à projeter. Le plan de projection sera d'office perpendiculaire à la ligne de vue et situé entre le point de vue et le volume à projeter lorsque l'on utilise les projections "assistées" (ce qui est de loin le cas le plus courant en pratique). On définira d'une part les distances, le long de la ligne de vue, du point de vue au plan de projection et à un plan parallèle à ce dernier, et d'autre part les dimensions d'un rectangle du plan de projection qui contiendra le résultat de la projection. L'ensemble de ces informations, en conjonction avec celles données à l'étape précédente, est suffisant pour définir un volume à projeter univoque.

Voyons maintenant en détails les instructions OpenGL relatives à ces deux étapes.

6.3.2.Définition du point de vue et de la ligne de vue

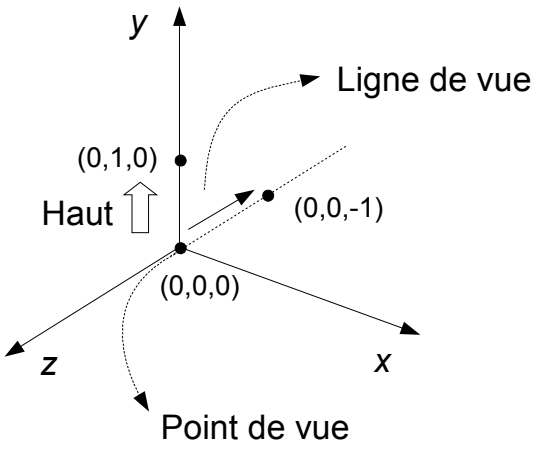
En OpenGL c'est l'instruction `gluLookAt()` qui permet de définir le point de vue et la ligne de vue, en coordonnées de dessin. La syntaxe en est :

```
gluLookAt(eye_x,eye_y,eye_z,center_x,center_y,center_z,up_x,up_y,up_z)
```

Les paramètres attendus sont des "double". Les trois premiers paramètres (`eye_x, eye_y, eye_z`) définissent le point de vue. Les trois suivantes (`center_x, center_y, center_z`) définissent un point vers lequel l'oeil regarde : le vecteur pointant de (`eye_x, eye_y, eye_z`) vers (`center_x, center_y, center_z`) définit la ligne de vue. Finalement, le vecteur pointant de (0, 0, 0) vers (`up_x, up_y, up_z`) indique l'orientation du haut de la scène.

Par défaut, le point de vue est à l'origine du repère et pointe dans la direction de l'axe *z* négatif, tandis que le haut de la scène est orienté vers les *y* positifs. Donc omettre l'instruction `gluLookAt()` revient à l'appel suivant :

```
gluLookAt(0.0,0.0,0.0,0.0,0.0,-1.0,0.0,1.0,0.0)
```



En fait, l'instruction `gluLookAt()` revient à opérer sur le modèle les transformations géométriques nécessaires pour obtenir la vision souhaitée. Elle est donc appelée lorsque la matrice de *modelview* est activée, et avant tout autre transformation : toutes les primitives graphiques définissant la scène subissent ainsi ensemble ces transformations. L'instruction `gluLookAt()` revient à appliquer une translation et une rotation à l'ensemble de la scène avant d'opérer le travail de projection. On placera donc généralement les trois instructions suivantes dans “display” avant de commencer à décrire le modèle :

```
...
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(...);
...
```

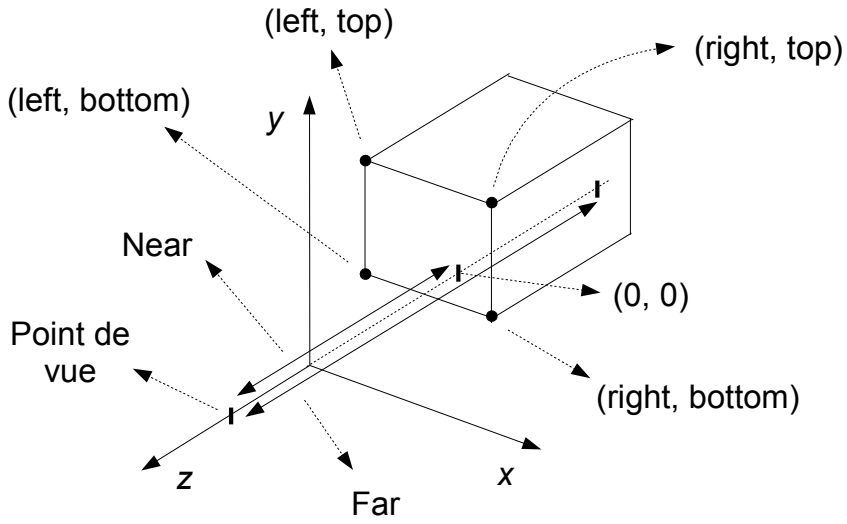
6.3.3.Définir les cractéristiques de projection

Il existe plusieurs possibilités pour caractériser la projection désirée. Avant tout, il faut passer en mode projection (`glMatrixMode(GL_PROJECTION)`) et initialiser la matrice de projection (`glLoadIdentity()`).

Pour une projection parallèle, on utilise l'instruction `glOrtho(...)` (ou sa version simplifiée `gluOrtho2D(...)` qui a été expliquée en section 2.3), dont voici la syntaxe :

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

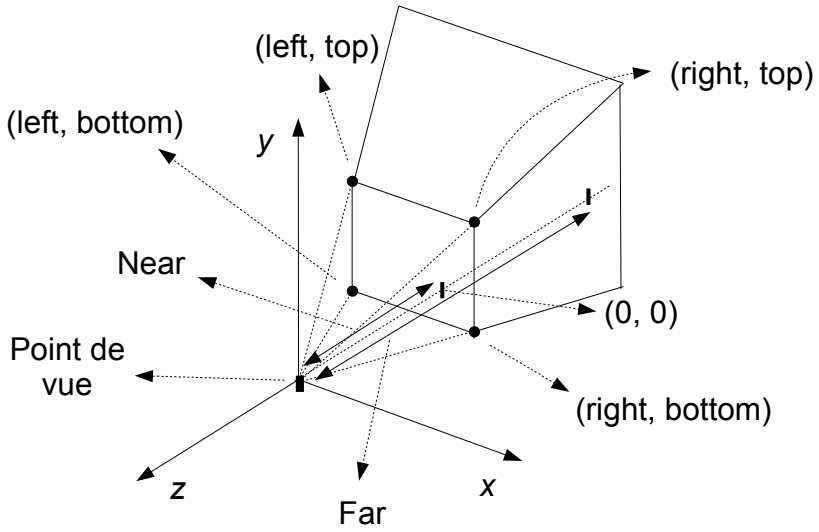
Les paramètres attendus sont des "double". Considérons un système de coordonnées en 2 dimensions dont l'origine se trouve à l'intersection de la ligne de vue et d'un plan perpendiculaire à la direction de projection, et dont les *y* positifs sont déterminés par l'orientation du haut de la scène dans `gluLookAt(...)`. Dans un tel système, les coordonnées, en unité de dessin, des coins inférieur gauche et supérieur droit du rectangle défini sur le plan perpendiculaire à la direction de projection sont donnés par (`left,bottom`) et (`right,top`). Les paramètres `near` et `far` représentent quant à eux les distances minimale et maximale, le long de la ligne de vue, entre le point de vision et les deux plans perpendiculaires à la direction de projection qui délimitent le volume à projeter. Ces deux derniers paramètres doivent donc être positifs.



Pour une projection en perspective, on utilise soit `gluPerspective(...)`, soit `glFrustum(...)`. L'instruction `glFrustum()` prend les mêmes paramètres que `glOrtho()` :

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

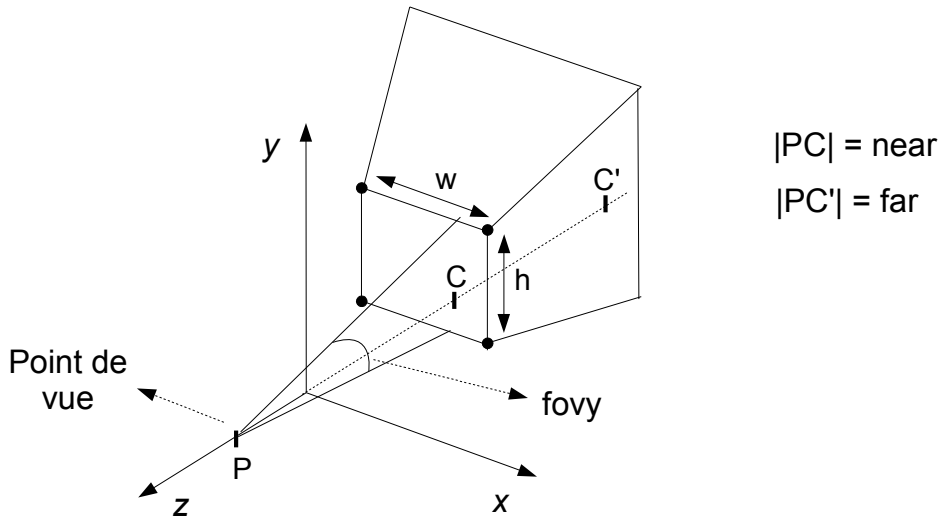
Les paramètres attendus sont des "double". Le rectangle définissant la zone de projection sur le plan perpendiculaire à la direction de projection, et situé à une distance `near` du centre de projection, est caractérisé de la même façon que dans `glOrtho()` ci-dessus. Cependant, le centre de projection est maintenant à distance finie du modèle, à l'emplacement du point de vision défini via `gluLookAt(...)`. Le volume à projeter, une pyramide tronquée, est déterminé en prolongeant les segments reliant le centre de projection aux sommets du rectangle défini ci-dessus, jusqu'au plan situé à une distance `far` du centre de projection.



L'instruction `gluPerspective(...)`, un peu plus compliquée à spécifier, se révèle souvent plus intuitive à utiliser :

```
gluPerspective(fovy,aspect,near,far)
```

Les paramètres attendus sont des "double". Cette instruction permet de définir un volume symétrique par rapport au centre de projection. Les paramètres `near` et `far` sont toujours les distances minimale et maximale du centre de projection aux cotés du volume à projeter perpendiculaires à la direction de projection. Le paramètre `aspect` définit l'"aspect ratio" (*w*/*h*) du rectangle défini sur les plans du volume de projection perpendiculaires à la direction de projection. Finalement, `fovy` est un angle définissant la hauteur du champ de vision.



Le choix de l'angle défini par `fovy` peut se faire en considérant la distance entre l'oeil et l'écran d'ordinateur d'une part, et la hauteur de la fenêtre d'écran d'autre part. L'angle permettant de sous-tendre la hauteur de la fenêtre à partir du point d'observation sera souvent un choix donnant un bon résultat. Par exemple, pour un utilisateur situé à 80 cm de l'écran et une fenêtre de 20 cm de haut, on pourra choisir un angle égal à *2arctg((20/2)/80)*, c'est-à-dire environ 14°.

Les instructions de projection se placent généralement dans la fonction "viewport", juste après la définition de la viewport qu'elles concernent. En effet, ces instructions sont souvent paramétrées par la largeur (*w*) et la hauteur (*h*) de la fenêtre d'écran pour éviter les problèmes de distorsion, et leurs valeurs sont directement disponibles dans la fonction "reshape". Cela permet d'ajuster le volume à projeter lorsque l'utilisateur modifie les dimensions de la fenêtre d'écran. Si l'on utilise `gluPerspective(...)` par exemple, le paramètre `aspect` sera souvent égal à *w*/*h*, ce qui est un grand avantage de cette instruction. On aura donc classiquement la structure suivante dans la fonction "viewport" :

```
void viewport()
{
  glViewport(...); //définition d'une viewport
  glMatrixMode(GL_PROJECTION); //activation de la matrice de projection
  glLoadIdentity(); //initialisation de la matrice de projection
  // si projection parallèle :
  glOrtho(...); ou gluOrtho2D(...);
  // si projection en perspective :
  glFrustum(...); ou gluPerspective(...);
  ...
}
```

7. Positionnement et animation

Lorsque l'on veut spécifier certains modèles, on remarque souvent une répétition de certains motifs à l'intérieur du modèle. Ces répétitions peuvent être exploitées pour simplifier la spécification, et ceci est particulièrement facilité en OpenGL par l'utilisation de la pile des matrices de transformations géométriques.

Nous allons illustrer sur un exemple simple l'utilisation de cette pile pour le positionnement des objets dans un tel modèle, ainsi que pour l'obtention d'un effet de mouvement. Avant cela, les deux façons d'aborder l'ordre de succession des transformations géométriques seront présentées. Pour terminer ce chapitre, on verra comment faire pour qu'un objet en masquant un autre soit bien dessiné devant celui qu'il cache lors d'une projection d'un modèle 3D.

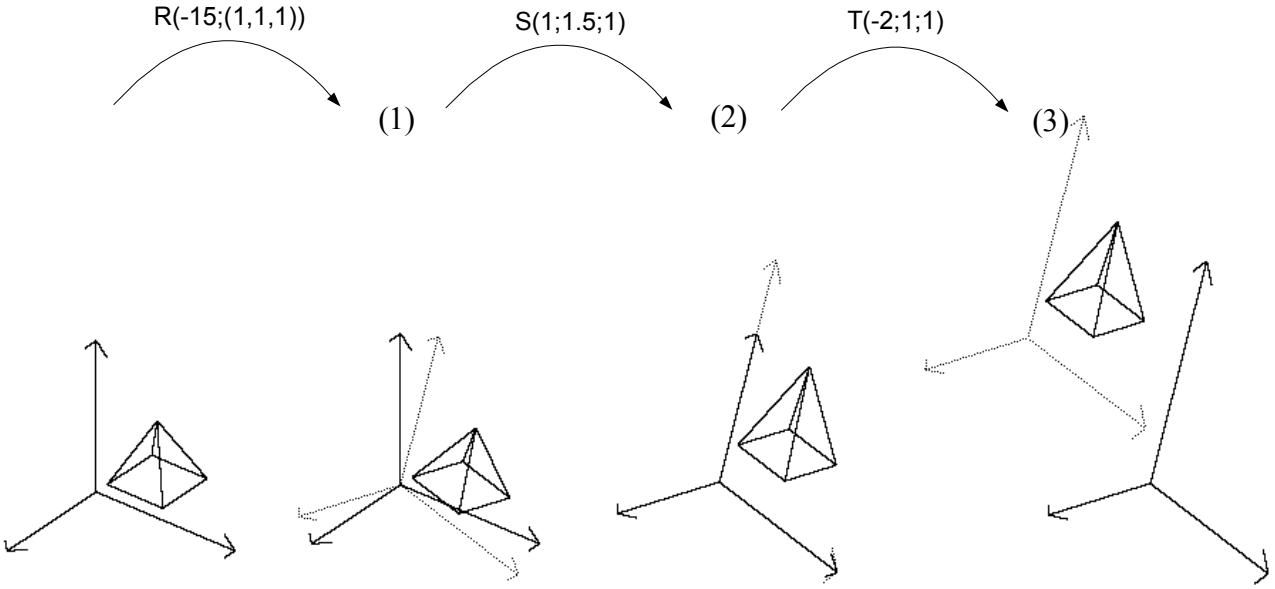
7.1. L'ordre de succession des transformations géométriques

Il existe deux façons de se représenter mentalement l'ordre de succession des tranformations géométriques : d'une part en considérant un repère fixe par rapport auquel l'objet subit les différentes transformations, et d'autre part en considérant un repère lié à l'objet et subissant les transformations en même temps que lui.

La première de ces deux représentation est celle que nous avons utilisée jusqu'à présent. Les axes du repère de coordonnées de dessin sont fixes et un objet subit une série de transformations successives qui lui donnent l'apparence et la position désirées par rapport au reste de la scène. Les instructions OpenGL qui décrivent les différentes transformations sont alors appelées dans le code dans l'ordre inverse de l'ordre dans lequel elles doivent être appliquées à l'objet, comme expliqué en détail au chapitre 5. Dans le cas de la pyramide exposé dans la section 6.1, les différentes étapes sont illustrées dans la figure proposée à la fin de cette même section.

Une autre façon de se représenter une succession de transformations géométriques est de considérer un système d'axes *lié* à l'objet qui subit les transformations. C'est-à-dire que le système d'axes subit les mêmes transformations que l'objet, relativement à lui-même. Lors d'une translation par exemple, l'objet est déplacé d'un vecteur (Δx,Δy,Δz) par rapport au système d'axes, et un nouveau système d'axes est obtenu en déplaçant le sytème d'axes précédant du même vecteur (Δx,Δy,Δz) par rapport à lui-même. La tranformation suivante se fera par rapport à ce nouveau système d'axes. Lorsque l'on raisonne de cette façon, les instructions OpenGL correspondant aux différentes transformations devront apparaître dans le code dans l'ordre dans lequel on se les représentent.

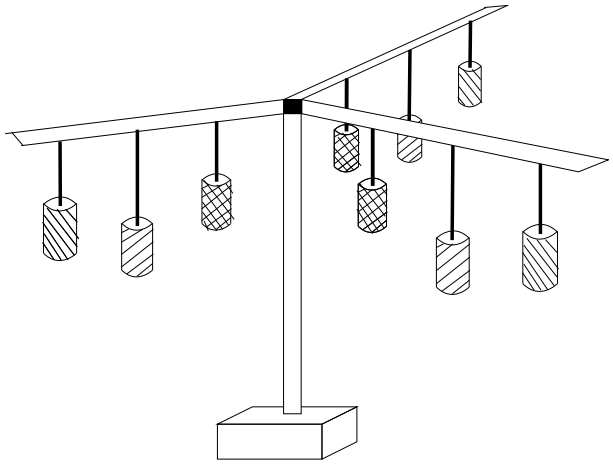
Reprenons l'exemple de la pyramide pour illustrer ce qui se passe lorsque l'on considère un repère lié à l'objet. Pour obtenir le résultat de l'exemple, on commence par effectuer la rotation horlogique de 15° autour de l'axe défini par le vecteur (*1*,*1*,*1*), qui s'accompagne d'une rotation identique du système d'axe lui-même (1). On opère ensuite une transformation d'échelle de facteur 1.5 suivant la direction de l'axe *y* dans le système d'axe ayant subi la rotation. En même temps que la pyramide est étirée dans cette direction, le vecteur unité suivant l'axe *y* est lui aussi multiplié par *1.5* : une unité suivant l'axe *y* vaut maintenant 1.5 fois une unité suivant l'axe *x* ou l'axe *z* (2). Finalement, on applique *T(-2;1;1)* dans le nouveau système d'axes, pour obtenir une pyramide de dimensions et d'emplacement identiques à celle de la figure proposée à la fin de la section 6.1 (3).



Chacun choisit le type de représentation qui lui convient le mieux. Il est intéressant d'avoir connaissance des deux types de représentation car il peut s'avérer utile de passer de l'un à l'autre suivant le cas de figure auquel on est confronté.

7.2.Positionnement d'objets dans un modèle avec "répétitions"

Imaginons un carrousel de foire ayant les caractéristiques suivantes : trois axes transversaux partent du sommet d'un mat central, et à chacune de ces traverses trois nacelles équidistantes représentant chacune un animal différent sont suspendues à des hauteurs différentes. Chaque ensemble "traverse + nacelles" est identique aux deux autres.



Supposons que l'on dispose déjà des fonctions *dessineMat()*, *dessineTraverse()*, *dessine-Nacelle(int typeAnimal)* et *dessineRaccordVertical()*. La fonction *dessineTraverse()* dessine un objet orienté vers les *x* positifs avec son extrémité gauche à l'origine du repère. Les fonctions *dessineMat()* et *dessineRaccordVertical()* dessinent ces objets orientés vers les *y* positifs et avec leur base centrée à l'origine du repère. De plus, *dessineRaccordVertical()* dessine un raccord de

hauteur 1. Dans la spécification du modèle, on va tenir compte des deux motifs répétitifs : la répétition de l'ensemble "traverse + nacelles" et la répétition "raccord vertical + nacelle" sur chaque traverse.

Si l'on choisit le sommet du mat comme origine du repère, on peut programmer la spécification comme suit (en considérant que la distance séparant 2 nacelles vaut 5) :

```
void dessineCarrousel(float hauteur[], int typeAnimal[])
// hauteur[] contient les distances entre la traverse et les nacelles
{
  int i;
  glPushMatrix();
  glTranslated(0.0,-10.0,0.0);//positionnement vertical
  dessineMat();
  glPopMatrix();//annule le positionnement vertical
  for(i=0;i<3;i++){
    glPushMatrix();
    //Amène l'ensemble "traverse+nacelles" dans l'orientation voulue
    glRotated(120.0*i,0.0,1.0,0.0);

    dessineTraversePlusNacelles(hauteur, typesAnimaux);
    //annule la rotation
    glPopMatrix();
  }
}
void dessineTraversePlusNacelles(float hauteur[], int typeAnimal[])
{
  int i;
  dessineTraverse();
  glPushMatrix();
  for(i=0;i<3;i++){
    glTranslated(5.0,0.0,0.0);//positionnement horizontal
    glPushMatrix();
    glTranslated(0.0,-hauteur[i],0.0);//positionnement vertical
    glPushMatrix();
    glScaled(1.0,hauteur[i],1.0);//établit la hauteur du raccord
    dessineRaccordVertical();
    glPopMatrix();//annule le glScaled(), spécifique au raccord
    dessineNacelle(typeAnimal[i]);
    glPopMatrix();//annule le positionnement vertical
  }
```

```
    glPopMatrix();//annule les positionnements horizontaux
}
```

Le principe est d'exploiter le plus possible la modularisation en travaillant par imbrication : on définit un objet général en imbriquant des objets de plus en plus particuliers. Le carrousel contient 3 traverses, contenant chacune 3 ensembles "raccord vertical + nacelle". De cette façon, on économise naturellement des opérations de transformation géométrique : chaque fois qu'une même série de transformations géométriques est commune au positionnement d'objets différents, elle peut être sauvegardée dans la pile des matrices de transformation via `glPushMatrix()`, puis récupérée au moment opportun via `glPopMatrix()`.

7.3.Animer le modèle

Pour donner une impression de mouvement, il est nécessaire et suffisant de respécifier le modèle autant de fois que nécessaire en plaçant les objets animés en des points successifs de leurs trajectoires, points suffisamment proches l'un de l'autre pour obtenir un effet de mouvement continu. L'affichage est constamment réactualisé à intervalle d'horloge machine, ce qui permet d'obtenir l'effet d'animation. C'est pour cette raison qu'en OpenGL tout est conçu pour que le processus permettant d'obtenir le rendu dans la fenêtre d'écran, à partir de la spécification du modèle et de la projection désirée, puisse se faire le plus rapidement possible.

Il existe deux instructions de la librairie GLUT qui permettent de gérer les animations. Il s'agit de `glutIdleFunc()` et `glutPostRedisplay()`. La première permet de spécifier une fonction qui doit être exécutée lorsqu'aucun travail n'est en cours ni aucun évènement en attente. La seconde indique que la fenêtre d'écran doit être rafraîchie, c'est-à-dire que la fonction "display" doit être appelée dès que possible. Ainsi, un incrément relatif à une spécification de mouvement peut être appliqué dans la fonction appelée par `glutIdleFunc()`, puis l'instruction `glutPostRedisplay()` est appelée dans cette même fonction pour rafraîchir la fenêtre d'écran avec une spécification du modèle dans laquelle l'objet en mouvement a été positionné à l'emplacement suivant de sa trajectoire.

Illustrons ceci sur base de l'exemple du carrousel, auquel nous allons donner une vitesse de rotation constante autour du mat. Nous allons utiliser la fonction `DessineCarrousel()` décrite en section 7.2, en en retirant cependant les instructions relatives à la spécification du mat : en effet, le mat devra rester fixe tandis que les trois ensembles "traverse + nacelles" seront en mouvement. Le code pourra se présenter comme suit :

```
static Glfloat hauteur[3]={2.0,3.0,4.0};
static Glint typeAnimal[3]={1,2,3};
static Glfloat spin=0.0;


void display(void)
{
   //réinitialisation du buffer à la couleur de fond
   glClear(GL_COLOR_BUFFER_BIT);


   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
```

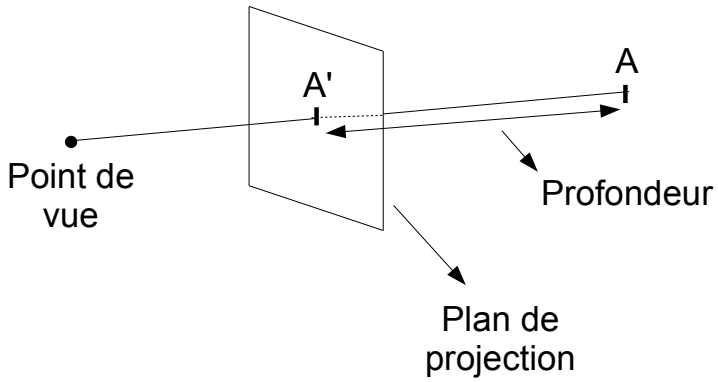
```
    glPushMatrix();
    glTranslated(0.0,-10.0,0.0);//positionnement vertical
    dessineMat();
    glPopMatrix();//annule le positionnement vertical
    glPushMatrix();
    glRotated(spin,0.0,1.0,0.0);//rotation autour de l'axe y (axe du mat)
    dessineCarrousel(hauteur,typeAnimal);
    //annule toutes les transformations géométriques depuis la rotation
      spécifiée deux lignes ci-dessus
    glPopMatrix();

    glFlush();//force l'exécution des commandes OpenGL déjà spécifiées
}
...
void spinDisplay(void)
//incrémente l'angle utilisé pour le mouvement (spin)
{
    spin+=2.0;
    if(spin>360.0)//limite la valeur de spin entre 0.0 et 360.0
       spin-=360.0;
    glutPostRedisplay();//demande le rafraîchissement de la fenêtre d'écran
}
...
int main(int argc, char **argv)
{
    ...
    //demande l'incrémentation de l'angle relatif au mouvement lorsqu'aucun
      travail n'est en cours ni aucun évènement en attente
    glutIdleFunc(spinDisplay);
    ...
}
```

L'instruction `glRotated(spin,0.0,1.0,0.0)` précédant `dessineCarrousel(...)`, le carrousel sera dessiné en subissant à chaque fois une rotation de 2 degrés (`spin+=2.0` dans `spinDisplay`) dans le sens anti-horlogique autour de l'axe *y* par rapport à sa position précédente. En fait, le carrousel est d'abord représenté dans sa position initiale, puis après avoir subi des rotations de 2 degrés , 4 degrés , 6 degrés , 8 degrés, etc, par rapport à cette position initiale. On peut modifier la vitesse soit en modifiant l'incrément d'angle, soit en utilisant une instruction / série d'instructions visant à ralentir le processus. On pourra par exemple inclure une boucle vide dans `display`, ou encore une instruction plus appropriée spécifique au système d'exploitation telle que `usleep(...)` sous Windows.

7.4.Le problème des surfaces cachées

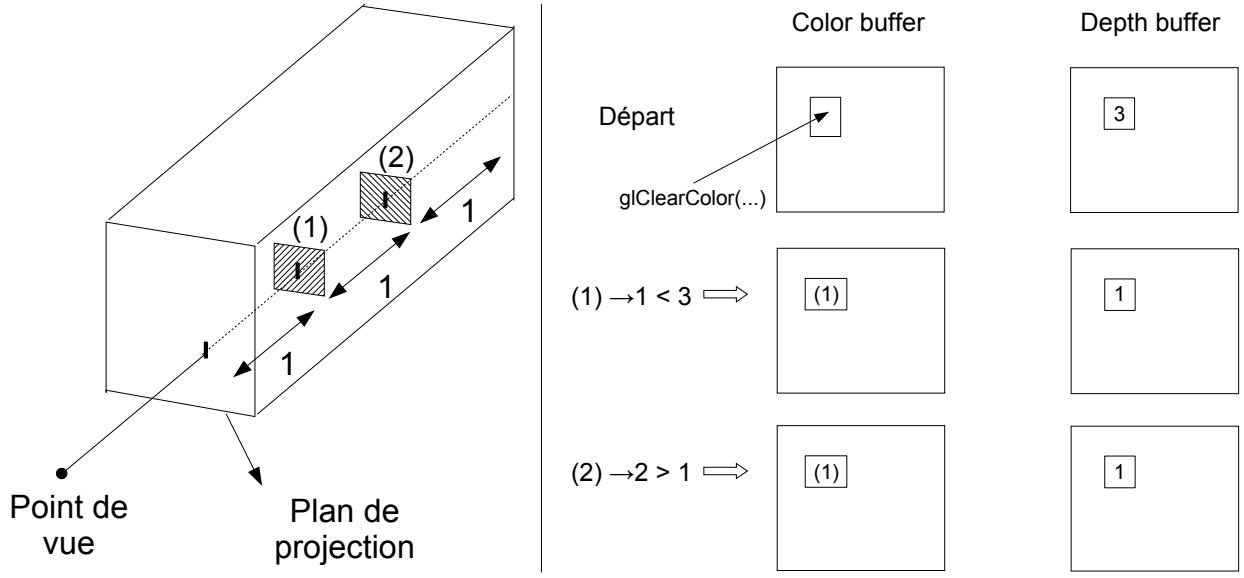
Par défaut, OpenGL ne vérifie pas si un objet en obstrue partiellement ou totalement un autre par rapport au point de vision. Si plusieurs objets se succèdent sur une même ligne de vue, c'est le dernier objet spécifié dans `display` qui apparaitra dans la fenêtre d'écran, même s'il est caché par un autre dans le modèle 3D.

Pour éviter ce problème, il faut activer un *buffer* supplémentaire, appelé *buffer* de profondeur (*depth buffer*). Ce *buffer* associe une profondeur à un pixel donné : il s'agit de la distance, le long de la ligne de vue, entre le plan de projection et l'endroit que le pixel représente.



Lors du travail de conversion d'une surface convexe définie par une primitive graphique en un ensemble de pixels colorés, une telle profondeur est calculée pour chaque pixel. Lorsque le *buffer* de profondeur est activé, la profondeur associée à chaque pixel est initialisée à une valeur égale à la distance maximum possible depuis le plan de projection. Ensuite, chaque fois qu'une information doit être écrite dans le *buffer* de couleur concernant un pixel donné, on compare d'abord la profondeur de ce pixel avec la profondeur déjà enregistrée dans le *buffer* de profondeur : si la nouvelle profondeur est inférieure, elle remplace la précédente et l'information concernant le pixel traité est inscrite dans le *buffer* de couleur ; dans le cas contraire, les informations concernant le pixel traité sont ignorées. De cette façon chaque pixel affiché à l'écran aura les caractéristiques de la primitive graphique la plus proche du point de vision.

Dans l'exemple ci-dessous, la surface (1) masque la surface (2) par rapport au point de vision. Imaginons que la spécification de (1) ait lieu avant celle de (2) dans `display` et que le *buffer* de profondeur soit activé. Au départ, le pixel relatif à la ligne de vue représentée sur la figure a une profondeur maximum, c'est-à-dire 3. Lorsque la spécification de (1) est rencontrée dans `display`, la profondeur correspondante (qui vaut 1) va remplacer la valeur courante du *buffer* de profondeur (qui vaut 3) car elle lui est inférieure. De plus, le *buffer* de couleur prendra les caractéristiques de la surface (1) pour ce pixel. Ensuite, lorsque la spécification de (2) est rencontrée dans `display`, la profondeur correspondante (qui vaut 2) étant supérieure à la valeur courante du *buffer* de profondeur (qui vaut 1), ni le *buffer* de profondeur, ni le *buffer* de couleur ne sont modifiés.



L'activation du *buffer* de profondeur augmente le temps nécessaire au traitement puisqu'il faut implémenter un *buffer* supplémentaire et faire les tests de comparaison des profondeurs chaque fois qu'un pixel est traité. Il y a cependant une compensation partielle puisqu'il y a écriture dans le *buffer* de couleur seulement lorsque la primitve graphique concernée n'est pas masquée par une autre primitive déjà spécifiée. Par contre, lorsque le *buffer* de profondeur n'est pas activé, il y a écriture concernant un même pixel dans le *buffer* de couleur autant de fois qu'il y a de spécifications de primitives graphiques se trouvant sur une même ligne de vue.

L'utilisation du *buffer* de profondeur fait intervenir trois instructions en OpenGL :

1. Il faut ajouter la constante `GLUT_DEPTH` aux arguments de `glutInitDisplayMode(...)` : `glutInitDisplayMode(...|GLUT_DEPTH|...)`;
2. Il faut activer l'implémentation du *buffer* de profondeur via l'instruction `glEnable(GL_DEPTH_TEST)`, ce qui se fait généralement dans `init()`;
3. Il faut finalement initialiser le *buffer* de profondeur avec la profondeur maximum possible en ajoutant `GL_DEPTH_BUFFER_BIT` aux arguments de `glClear(...)` : `glClear(...|GL_DEPTH_BUFFER_BIT|...)`.

Moyennant ces trois instructions, les surfaces cachées seront systématiquement éliminées de l'image apparaissant dans la fenêtre d'écran.

8. Display lists

OpenGL offre la possibilité de stocker un ensemble de commandes successives dans une structure particulière, appelée *display list* (que l'on pourrait traduire littéralement par "liste d'affichage"). La création d'une *display list* peut être vue comme la mise en cache des instructions correspondantes, pour utilisation ultérieure.

8.1. Avantages et désavantages des display lists

Les principaux avantages des *display lists* sont les suivants :

- Une série d'instructions est définie une seule fois, au moment de la création de la liste, et peut être ensuite éxécutée autant de fois que l'on veut par un simple appel de la liste;
- De nombreuses instructions nécessitent un certain calcul lorsqu'elles sont appelées : par exemple, `glRotated(...)` entraîne la construction d'une matrice (4x4) dont certains éléments nécessitent l'évaluation de fonctions trigonométriques. Cette matrice multiplie ensuite la matrice des transformations géométriques. Si une série de transformations géométriques sont inclues dans une *display list*, le travail relatif à la construction des matrices nécessaires et certaines multiplications matricielles seront effectuées une seule fois, au moment de la création de la liste. Le résultat de ces opérations sera stocké, puis récupéré chaque fois que nécessaire pour définir le modèle. Les *display lists* permettent donc d'éviter la répétition de calculs lorsque certains types d'instructions ont un caractère répétitif;
- Sur certains systèmes d'exploitation, certaines informations contenues dans une *display list* peuvent être stockées dans un format directement utilisable par le *hardware*, ce qui permet d'améliorer encore les performances;
- Lorsque le logiciel graphique est utilisé via un réseau, c'est-à-dire qu'une série de clients émettent des demandes qui sont reçues et exécutées par un serveur, il est particulièrement intéressant de pouvoir stocker des commandes répétitives dans des *display lists*. On limite ainsi fortement la quantité d'information qui doit être échangée via le réseau.

Les *display lists* ont aussi deux limitations importantes, qu'il faut garder à l'esprit pour les utiliser toujours à bon escient :

- Le contenu d'une *display list* ne peut être changé une fois que celle-ci a été compilée. Par exemple, si une primitive graphique est stockée dans une *display list* en incluant la spécification de sa couleur, cette dernière ne pourra plus être changée. Chaque fois que la *display list* sera appelée, la primitive graphique sera affichée avec la couleur spécifiée au moment de la compilation de la liste, même si l'on spécifie une autre couleur juste avant l'appel de la liste. Lors de l'appel de la liste, c'est la dernière couleur spécifiée dans celle-ci qui devient la couleur courante au niveau de `display`.
- L'appel d'une *display list* requiert toujours un certain travail de localisation et de récupération de l'information stockée dans celle-ci. Il est donc désavantageux de définir une *display list* trop petite (contenant trop peu d'instructions), car le gain ne compensera pas ce travail.

8.2. Création et exécution d'une display list en OpenGL

Chaque *display list* est identifiée par un numéro unique. La première chose à faire lorsque l'on veut créer des *display lists* est de demander à OpenGL quels numéros d'identification on peut utiliser. Cela se fait via l'instruction `glGenLists(nbLists)` qui renvoie un entier non signé représentant le premier numéro d'identification de `nbLists` numéros consécutifs. Si l'on désire créer 3 *display lists*, on écrira par exemple `firstId=glGenLists(3);`. Si l'entier assigné à `firstId` vaut 23, cela signifie que les 3 numéros d'identifications disponibles pour les 3 *display lists* sont 23, 24 et 25. Si 0 (zéro) est assigné à `firstId`, cela signifie qu'il n'y a pas assez de numéros d'identification disponibles.

La création d'une *display list* débute par l'instruction `glNewList(listId, mode)` et se termine par l'instruction `glEndList()`. Le paramètre `listId` désigne le numéro d'identification de la liste, tandis que `mode` peut prendre les valeurs `GL_COMPILE` ou `GL_COMPILE_AND_EXECUTE` selon que l'on désire que les instructions composant la liste soient simplement compilées lors de sa définition (le plus courant), ou bien qu'elles soient aussi éxécutées une première fois. Certaines instructions ne peuvent pas faire partie d'une *display list* ; il s'agit de toutes les instructions renvoyant une valeur ou l'adresse d'un objet, ou encore des instructions qui font intervenir l'état d'un client dans une configuration en réseau. Le lecteur se réfèrera au manuel de référence d'OpenGL pour une liste exhaustive des instructions interdites. Toutes les instructions relatives à la définition d'une primitive graphique et de ses caractéristiques ou à des opérations sur l'une ou l'autre des matrices de travail sont bien sûr autorisées.

L'éxécution (ou appel) d'une *display list* se fait via l'instruction `glCallList(listId)`, ou `listId` identifie la liste à éxécuter. Elle a lieu directement ou indirectement à partir de la fonction `display`.

Deux autres instuctions peuvent se révéler utiles lors de l'utilisation de *display lists* : `glIsList(listId)` renvoie `GL_TRUE` ou `GL_FALSE` selon que `listId` est oui ou non l'identifiant d'une liste ; `glDeleteLists(listId, nbLists)` efface `nbLists` listes dont les identifiants sont consécutifs et commencent à `listId`.

Pour terminer, voici un exemple illustrant la création et l'éxécution de *display lists*. Nous reprenons le modèle du carrousel introduit au chapitre 7, en considérant que toutes les nacelles sont identiques. Nous allons créer 3 *display lists* pour les 3 éléments répétitifs du modèle que sont les traverses, les raccords verticaux et les nacelles. Ces listes seront ensuite appelées chaque fois que nécessaire dans le modèle.

```
static int traverse, raccord, nacelle;//Pour stocker les IDs des listes
void init(void)
{
   traverse=glGenLists(3);//obtenir 3 IDs pour les 3 listes
   //Les 3 IDs sont consécutifs
   raccord=traverse+1;
   nacelle=traverse+2;
```

```
      //création de la liste pour une traverse
      glNewList(traverse,GL_COMPILE);
        //instructions de définition d'une traverse
      glEndList();

      //création de la liste pour un raccord
      glNewList(raccord,GL_COMPILE);
        //instructions de définition d'un raccord
      glEndList();

      //création de la liste pour une nacelle
      glNewList(nacelle,GL_COMPILE);
        //instructions de définition d'un nacelle
      glEndList();
    }
  void dessineCarrousel(float hauteur[])
    // hauteur[] contient les distances entre la traverse et les nacelles
    {
      int i;
      glPushMatrix();
      glTranslated(0.0,-10.0,0.0);//positionnement vertical
      dessineMat();
      glPopMatrix();//annule le positionnement vertical
      for(i=0;i<3;i++){
        glPushMatrix();
        //Amène l'ensemble "traverse+nacelles" dans l'orientation voulue
        glRotated(120.0*i,0.0,1.0,0.0);

        dessineTraversePlusNacelles(hauteur)
        glPopMatrix();//annule la rotation
      }

    void dessineTraversePlusNacelles(float hauteur[])
    {
      int i;
      glCallList(traverse);//appelle la liste relative à une traverse
      glPushMatrix();
```

```
    for(i=0;i<3;i++){
      glTranslated(5.0,0.0,0.0);//positionnement horizontal
      glPushMatrix();
      glTranslated(0.0,-hauteur[i],0.0);//positionnement vertical
      glPushMatrix();
      glScaled(1.0,hauteur[i],1.0);//établit la hauteur du raccord
      glCallList(raccord);//appelle la liste relative à un raccord
      glPopMatrix();//annule le glScaled(), spécifique au raccord
      glCallList(nacelle);//appelle la liste relative à une nacelle
      glPopMatrix();//annule le positionnement vertical
    }
    glPopMatrix();//annule les positionnements horizontaux
  }
```

La définition des listes se fait au niveau de la fonction `init`, puis celles-ci sont appelées dans le code exactement comme si l'on appelait des "sous-fonctions" de `display`.

9. Lumière et matière

La perception visuelle du monde qui nous entoure est le résultat de la sollicitation des cellules de l'oeil par les photons qui y arrivent. Ces photons sont des particules d'énergie émises par diverses sources de lumière et qui sont perçues par l'oeil soit directement, soit après réflexion sur les surfaces de différents matériaux. Suivant la structure d'un matériau, certaines fréquences de cette énergie photonique sont absorbées et d'autres réfléchies par ce matériau : c'est la composition du spectre énergétique atteignant l'oeil qui détermine l'apparence du matériau.

C'est ce processus d'une grande complexité que l'on désire imiter pour donner une apparence la plus réaliste possible aux images rendues à l'écran. Les modèles de lumière et matière développés dans ce but sont des approximations plus ou moins abouties du processus réel. Ces modèles définissent différentes caractéristiques de la lumière et des matériaux, puis se donnent un certain nombre de règles de calcul de la couleur d'un pixel donné. Dans le système RGB d'OpenGL, la couleur d'un pixel est définie par l'intensité des trois composantes de base (rouge, vert, bleu), elle-même dépendant des caractéristiques des sources de lumière éclairant la scène et des matériaux composant cette dernière.

Dans un premier temps, nous passerons en revue les différentes composantes de la lumière qui sont prises en compte dans le modèle d'éclairage proposé par OpenGL. Puis nous décrirons comment définir les différentes caractéristiques d'une source de lumière, avant de dire un mot concernant les caractéristiques des matériaux. On expliquera ensuite comment définir un modèle d'éclairage et comment mettre une source lumineuse en mouvement en OpenGL. Finalement, on abordera quelques principes du calcul mis en oeuvre pour obtenir la couleur d'un pixel donné, sur base des différents paramètres définis dans le modèle.

9.1. Les composantes de la lumière

Le modèle considère que la lumière influençant une scène se décompose en 4 composantes indépendantes : *ambiante*, *diffuse*, *spéculaire* et *émissive*. Les trois premières composantes caractérisent d'une part les sources dont la lumière éclaire la scène en influençant les objets qui réfléchissent directement ou indirectement cette lumière, et d'autre part la façon dont les objets eux-même réfléchissent la lumière. La composante émissive ne caractérise quant à elle que des objets donnés et influence uniquement la couleur de ces objets.

La composante *ambiante* prend en compte une lumière dispersée, qui a été réfléchie tant de fois que sa source est impossible à localiser. Dans une pièce d'habitation, toute lumière émise est réfléchie de nombreuses fois sur différentes surfaces, principalement les murs, plafond et plancher : la composante ambiante y est donc importante.

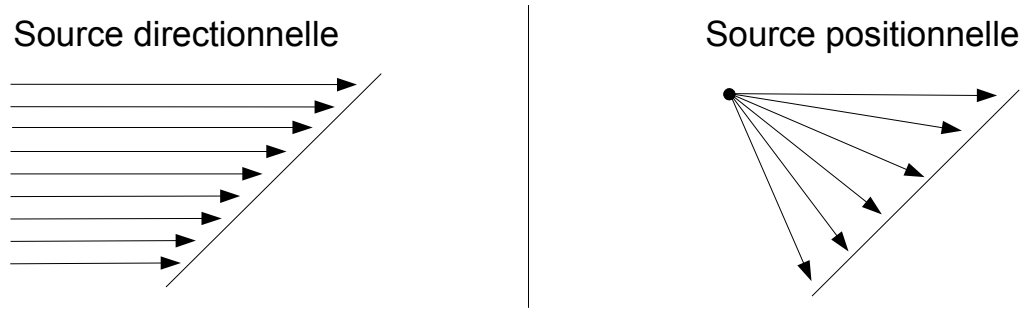
La composante *diffuse* prend en compte une lumière provenant d'une direction spécifique, et reflétée de façon égale dans toutes les directions par l'objet atteint. Plus l'angle de frappe de ce type de lumière sur la surface atteinte est proche d'un angle droit, au mieux cette surface est éclairée. Au contraire, un rayon de lumière diffuse tangent à une surface l'éclairera de façon moins prononcée. L'emplacement de l'oeil de l'observateur n'a pas de conséquence puisque cette composante est reflétée de façon égale dans toutes les directions.

La composante *spéculaire* prend elle aussi en compte une lumière provenant d'une direction spécifique, mais une lumière qui est réfléchie par la surface de l'objet dans une direction privilégiée. Le résultat perçu dépendera cette fois de la position de l'oeil de l'observateur. Ce

type de lumière se traduit par un effet de reflet sur le matériau ou encore de brillance, variant suivant l'endroit d'où l'on regarde.

9.2.Définir une source de lumière

Par défaut, il est possible de définir 8 sources de lumière différentes dans un modèle OpenGL. Celles-ci sont identifiées par les constantes GL_LIGHT0 à GL_LIGHT7. Pour chaque source, il faut définir ses composantes ambiante, diffuse et spéculaire ainsi que sa position dans l'espace (en coordonnées de dessin). Une source peut être soit *directionnelle*, si elle est située à distance "infinie" de la scène (comme le soleil par exemple), soit *positionnelle*, si elle est située à un endroit bien précis de la scène. Dans le cas d'une source directionnelle, tous les rayons de lumière sont considérés comme parallèles entre eux : ils ont la même direction, indépendamment de l'orientation de la surface éclairée par la source. En revanche dans le cas d'une source positionnelle, l'angle avec lequel un rayon frappe chaque point d'une surface dépend à la fois de l'orientation de la surface éclairée par rapport à la source et de la position de la source. Chaque source positionnelle peut aussi être considérée comme un spot. Dans ce cas, il faudra spécifier 3 paramètres supplémentaires caractérisant le spot.



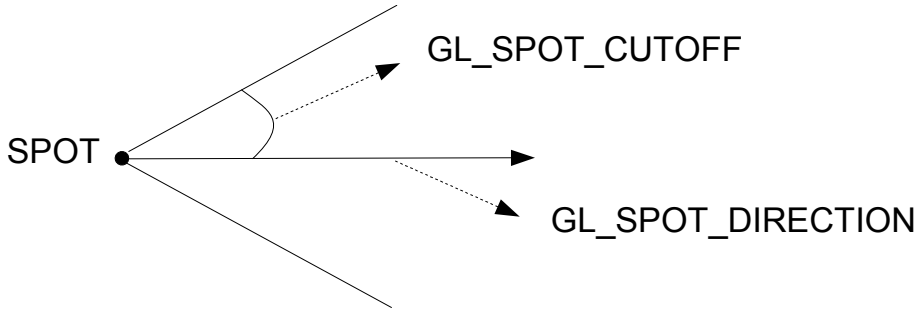
La spécification des différents paramètres définissant une source lumineuse se fait en utilisant soit `glLightf(light, characteristic, value)` soit `glLightfv(light, characteristic, value)`, répétée autant de fois que nécessaire. Le paramètre `light` identifie la source concernée (par exemple GL_LIGHT3). Le paramètre `characteristic` indique la caractéristique que l'on veut définir et le paramètre `value` la ou les valeurs que l'on veut lui assigner. Si une seule valeur doit être assignée, on utilise `glLightf(...)` où `value` est un scalaire (valeur unique). Si la caractéristique est définie par plusieurs valeurs, par exemple les 4 composantes RGBA d'une couleur, on utilise `glLightfv(...)` où `value` fait référence à un tableau contenant les différentes valeurs. Voici les différentes caractéristiques disponibles et leurs valeurs par défaut :

- GL_AMBIENT - composante ambiante de la source – défaut : (0.0,0.0,0.0,1.0) ;
- GL_DIFFUSE - composante diffuse de la source – défaut : (1.0,1.0,1.0,1.0) pour GL_LIGHT0 et (0.0,0.0,0.0,1.0) pour les autres ;
- GL_SPECULAR - composante spéculaire de la source – défaut : (1.0,1.0,1.0,1.0) pour GL_LIGHT0 et (0.0,0.0,0.0,1.0) pour les autres ;
- GL_POSITION – position dans l'espace – défaut : (0.0,0.0,1.0,0.0) ;
- GL_SPOT_DIRECTION – direction de la lumière du spot – défaut : (0.0,0.0,-1.0) ;
- GL_SPOT_EXPONENT – caractère concentré de la lumière – défaut : 0.0 ;

- GL_SPOT_CUTOFF – moitié de l'angle du cône de lumière – défaut : 180.0 ;
- GL_CONSTANT_ATTENUATION – facteur constant d'atténuation – défaut : 1.0 ;
- GL_LINEAR_ATTENUATION – facteur linéaire d'atténuation – défaut : 0.0 ;
- GL_QUADRATIC_ATTENUATION – facteur quadratique d'atténuation – défaut : 0.0.

On spécifie les composantes RGBA de la couleur de chaque composante (ambiante, diffuse et spéculaire) de la source via les constantes GL_AMBIENT, GL_DIFFUSE et GL_SPECULAR. La position de la source est spécifiée, en coordonnées de dessin, via GL_POSITION : si la coordonnée homogène (w) est égale à 0, il s'agit d'une source directionnelle et le vecteur reliant (x,y,z) à l'origine définit la direction de la lumière ; si la coordonnée homogène est non nulle, il s'agit d'une source positionnelle et le point (x,y,z) définit sa position dans l'espace.

Lorsqu'une source est définie comme un spot, elle émet un cône de lumière dans une direction spécifique obtenue en reliant l'origine du repère avec le point spécifié via GL_SPOT_DIRECTION. Le paramètre GL_SPOT_CUTOFF permet de définir la moitié de l'angle qui soustend le cône de lumière émis par le spot. Cette valeur peut varier entre 0.0 et 90.0 degrés, la valeur de 180.0 degrés étant réservée pour indiquer que la source n'est pas un spot. Le paramètre GL_SPOT_EXPONENT prend une valeur minimum de 0.0, correspondant à une répartition uniforme de la lumière dans le cône directionnel du spot. Plus la valeur de GL_SPOT_EXPONENT augmente à partir de 0.0, plus la lumière sera concentrée suivant la direction du spot (tel un rayon laser).



Les trois dernières caractéristiques (paramètres d'atténuation) sont utilisés uniquement pour des sources positionnelles : elles définissent comment l'intensité de la lumière s'atténue lorsque l'on s'éloigne de la source. La contribution de la lumière provenant d'une source positionnelle est multipliée par un facteur d'atténuation donné par la formule $\frac{1}{k_c + k_l d + k_q d^2}$, où d représente la distance entre le point éclairé et la source en coordonnées de dessin et k_c, k_l et k_q sont des paramètres dont les valeurs sont fixées via GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION et GL_QUADRATIC_ATTENUATION, respectivement. Comme on considère qu'une source directionnelle se trouve à distance infinie de la scène, l'atténuation n'intervient pas pour ce type de source.

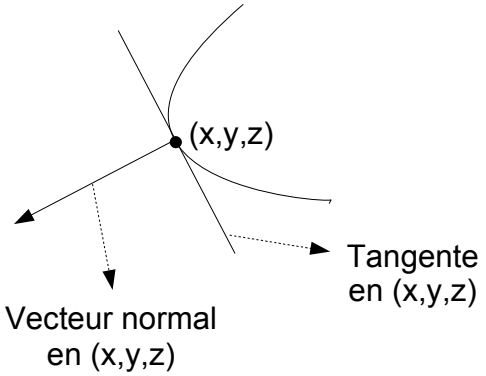
9.3.Définir les caractéristiques d'un matériau

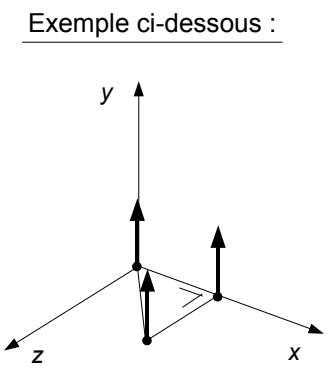
Pour chaque primitive graphique, il faut spécifier dans quelle mesure chacune des composantes RGB des composantes ambiante, diffuse et spéculaire de lumière sont réfléchies. Lorsque nous voyons un objet de couleur rouge, par exemple, c'est parce que cet objet absorbe les fréquences correspondant aux autres couleurs lorqu'une lumière l'atteint, et qu'il ne réfléchit

que des fréquences correspondant à du rouge. Pour chaque type de lumière atteignant une surface donnée d'un objet, OpenGL permet de spécifier dans quelle mesure chaque composante RGB de cette lumière est reflétée : une valeur de 0 signifie que la composante de couleur est totalement absorbée par le matériau, et une valeur de 1 correspond à une réflexion totale de la composante de couleur correspondante.

On peut aussi définir une composante *émissive* pour un matériau, ce qui permet de simuler l'émission de lumière par un objet donné. La composante émissive est totalement indépendante du reste du modèle d'éclairage : elle n'a aucun effet sur l'apparence des autres objets de la scène, ni d'influence sur la façon dont la lumière est reflétée par l'objet qui possède une composante émissive. Cela permet par exemple de représenter une lampe dans un modèle : on définira un objet représentant la lampe en lui associant une composante émissive correspondant à la couleur que la lampe doit avoir et l'on ajoutera au modèle une source de lumière positionnée au centre de cette lampe et dont les caractéristiques définissent le type de lumière que la lampe émet. Cela peut aussi permettre de définir un effet de phosphorescence.

La composante *spéculaire* d'une lumière est reflétée dans une direction privilégiée qui dépend d'une part de la direction d'où provient le rayon lumineux, et d'autre part de la direction du vecteur orthogonal à la surface du matériau à l'endroit éclairé. Il en va de même pour l'intensité de la composante *diffuse* : elle dépend de l'angle entre le rayon provenant de la source de lumière et le vecteur orthognal à la surface éclairée. Ce vecteur orthogonal à la surface du matériau s'appelle vecteur *normal* et doit être fourni à OpenGL en chaque sommet de chaque primitive graphique sur laquelle on veut utiliser la composante spéculaire et/ou diffuse de la lumière.





Définir le vecteur *normal* en un sommet se fait via l'instruction `glNormal3d(x,y,z)` dont l'appel précède celui de la définition du sommet correspondant. Le vecteur reliant l'origine au point `(x,y,z)` indique la direction du vecteur *normal* au sommet. Dans l'exemple qui suit, on définit un triangle dans le plan (*x,z*) et son vecteur normal pointant vers les *y* positifs :

```
glBegin(GL_TRIANGLES);
   glNormal3d(0.0,1.0,0.0);// vecteur normal identique pour les 3 sommets
   glVertex3d(0.0,0.0,0.0);
   glVertex3d(1.0,0.0,1.0);
   glVertex3d(1.0,0.0,0.0);
   glEnd();
```

Les vecteurs normaux sont donc définis en même temps que la primitive graphique, et subissent de ce fait les mêmes transformations géométriques que cette dernière. Pour des raisons techniques, les vecteurs normaux doivent être de longueur 1 ; dès lors, si l'on ne fournit pas les points adéquats dans `glNormal(...)` ou si l'on fait subir à la primitive graphique des transformations géométriques modifiant la longueur des vecteurs normaux (transformations d'échelle ou cisaillement), il faut demander à OpenGL de procéder aux ajustements nécessaires. Cela se fait en activant la normalisation automatique des vecteurs normaux à l'aide de l'instruction `glEnable(GL_NORMALIZE)`. En n'oubliant pas de la désactiver dès qu'elle n'est plus nécessaire.

Remarques :

- *un vecteur normal à une surface plane dont on connaît 3 points non alignés p1, p2 et p3 s'obtient en effectuant le produit vectoriel (p1-p2)x(p2-p3). Le résultat de ce produit vectoriel doit encore être normalisé avant d'être introduit en paramètre à* `glNormal(...)` *;*
- *pour rappel, le résultat du produit vectoriel de deux vecteurs* $(x_1, y_1, z_1)x(x_2, y_2, z_2)$ *est donné par* $(y_1z_2-z_1y_2 ; z_1x_2-x_1z_2 ; x_1y_2-y_1x_2)$.

Une dernière caractéristique des matériaux concerne l'intensité de l'éclat de lumière qui se forme lors de la réflexion de la composante spéculaire. Cet éclat peut être très concentré sur une petite surface ou plus diffus sur une surface plus importante. Cette caractéristique est gérée via le paramètre GL_SHININESS, compris entre 0.0 et 128.0 : plus celui-ci est élevé, plus l'éclat est intense.

Les différentes cractéristiques du matériau sont spécifiées via `glMaterialf(face, characteristic, value)`, où le paramètre `face` définit la ou les faces concernées (comme dans `glPolygonMode(...)` par exemple), `characteristic` désigne le paramètre concerné et `value` la ou les valeurs prises par ce paramètre. Tout comme avec `glLightfv(...)`, il existe une version `glMaterialfv(...)` de cette instruction. Voici un résumé des différentes caractéristiques disponibles avec leurs valeurs par défaut :

- GL_AMBIENT – réflexion de la composante ambiante – défaut : (0.2,0.2,0.2,1.0) ;
- GL_DIFFUSE – réflexion de la composante diffuse – défaut : (0.8,0.8,0.8,1.0) ;
- GL_AMBIENT_AND_DIFFUSE – pour spécifier les mêmes valeurs pour les composantes ambiante et diffuse ;
- GL_SPECULAR – réflexion de la composante spéculaire – défaut : (0.0,0.0,0.0,1.0) ;
- GL_SHININESS – caractère concentré de l'éclat – défaut : 0.0 – max : 128.0 ;
- GL_EMISSION – composante émissive du matériau – défaut : (0.0,0.0,0.0,1.0).

Les caractéristiques du matériaux se définissent de la même manière que la couleur, en répétant l'instruction `glMaterialf(...)` autant de fois que nécessaire avant chaque primitive graphique. Lorsque le modèle d'éclairage est activé, ce sont les propriétés définies par `glMaterialf(...)` qui sont utilisées par OpenGL pour calculer la couleur de chaque pixel en fonction des sources lumineuses actives, tandis que les instructions `glColor3f(...)` n'ont aucun effet dans ce cas.

9.4.Définir un modèle d'éclairage

Une fois que les propriétés des différentes sources de lumière et des matériaux composant la scène ont été spécifiées, il reste à définir les caractéristiques globales du modèle d'éclairage avant d'activer le mode d'éclairage et les sources désirées.

En plus des composantes ambiantes des différentes sources, le modèle incorpore une composante ambiante globale, indépendante des sources. La scène sera donc éclairée par cette lumière ambiante générale, même lorsque toutes les sources sont désactivées.

L'effet produit par la réflexion d'une lumière spéculaire est différent suivant que l'oeil de l'observateur est censé se trouver à l'intérieur de la scène (on parle alors d'observateur *local*), ou si l'observateur se trouve à une certaine distance de la scène (observateur à *l'"infini"*). Ceci est du au fait que la composante spéculaire perçue dépend de la direction reliant le point éclairé et le point de vision. Pour un observateur à l'infini, cette direction est identique pour tout sommet de la scène, tandis qu'elle doit être calculée pour chaque sommet dans le cas d'un observateur local. Dans ce dernier cas de figure, les calculs relatifs à la lumière spéculaire sont plus longs, mais l'effet est plus réaliste.

Lorsque seules les faces extérieures des objets sont visibles, il est inutile de calculer la couleur des faces intérieures des primitives graphiques (GL_BACK). Il est possible de spécifier à OpenGL s'il faut prendre en compte les deux faces des primitives graphiques, ou seules les faces extérieures.

Finalement, il est possible de spécifier si l'effet spéculaire doit être pris en compte avant ou après la mise en oeuvre des "textures" (voir le manuel OpenGL pour une introduction aux "textures"). Lorsqu'ils sont pris en compte après la mise en oeuvre des "textures", l'effet est plus réaliste mais les calculs plus longs.

La définition des caractéristiques du modèle d'éclairage se fait via l'appel de l'instruction `glLightModel*(characteristic, value)`, autant de fois que nécessaires. Les différentes caractéristiques et leurs valeurs sont :

- GL_LIGHT_MODEL_AMBIENT – composante ambiante globale de la lumière – défaut : (0.2,0.2,0.2,1.0) ;
- GL_LIGHT_MODEL_LOCAL_VIEWER – prend la valeur GL_TRUE pour un observateur *local* et GL_FALSE (défaut) pour un observateur à *l'"infini"* ;
- GL_LIGHT_MODEL_TWO_SIDE – prend la valeur GL_TRUE pour un traitement des deux faces des primitives graphiques et GL_FALSE (défaut) pour les faces extérieures seules ;
- GL_LIGHT_MODEL_COLOR_CONTROL – prend la valeur GL_SINGLE_COLOR (défaut) pour la prise en compte de l'effet spéculaire avant les "textures" et GL_SEPARATE_SPECULAR_COLOR dans le cas contraire.

Un fois que le modèle d'éclairage a été spécifié, il reste à l'activer à l'aide de l'instruction `glEnable(GL_LIGHTING)` et à "allumer"/"éteindre" les sources de lumières désirées à tout moment à l'aide de `glEnable(GL_LIGHT*)`/`glDisable(GL_LIGHT*)`, où l'astérisque prend les valeurs 0 à 7 selon la source. On peut alors admirer (ou déplorer) l'effet obtenu...

9.5.Animer les sources de lumière

Lorsque l'on spécifie la position d'une source de lumière (ou sa direction s'il s'agit d'une source directionnelle) à l'aide de `glLightfv(...)` appliquée au paramètre GL_POSITION, celle-ci est définie en coordonnées de dessin et subit les transformations géométriques précédant sa spécification de la même manière que les sommets des primitives graphiques. Il est donc possible de définir une source statique ou une source qui parcourt une certaine trajectoire par rapport à des objets fixes de la scène, ou encore une source subissant les même déplacements relatifs que ceux d'un objet en mouvement ou que ceux du point de vision (l'oeil de l'observateur). Tout dépend de l'endroit où l'on place l'instruction `glLightfv(...)` dans le code, et des transformations géométriques qui la précèdent. Les principaux cas de figures sont décrits ci-après.

Pour qu'une source reste fixe par rapport au système de coordonnées de dessin, il faut spécifier sa position après `gluLookAt(...)` et en lui faisant subir uniquement des transformations géométriques ne dépendant pas d'un incrément de mouvement. Si l'instruction `gluLookAt` n'est pas appelée, la position se spécifie juste après l'appel de `glLoadIdentity`.

Pour qu'une source suive une trajectoire donnée par rapport à un objet fixe, il faut spécifier sa position après `gluLookAt(...)` et en lui faisant subir des transformations géométriques incluant un incrément de mouvement par rapport à l'objet fixe.

Pour qu'une source subisse les mêmes déplacements relatifs qu'un objet en mouvement (c'est-à-dire que la source suit la même trajectoire que l'objet, mais avec un décalage dans l'espace par rapport à l'objet), il faut spécifier sa position après `gluLookAt(...)`, en lui faisant subir d'une part des transformations géométriques sans incrément de mouvement (pour la décaler par rapport à l'objet qu'elle doit suivre), combinés d'autre part avec des transformations incluant les mêmes incréments de mouvement que ceux subis par l'objet.

Pour qu'une source subisse les mêmes déplacements relatifs que ceux du point de vision, il faut spécifier sa position *avant* `gluLookAt(...)`, en lui faisant subir des transformations géométriques sans incrément de mouvement (pour la décaler par rapport au point de vision).

Dans les deux derniers paragraphes ci-dessus, les transformations géométriques sans incrément de mouvement sont facultatives. Il existe d'autres cas de figures plus compliqués, mais ceux qui précèdent sont suffisants pour comprendre les principes à appliquer.

9.6.Un exemple illustratif

Dans l'exemple ci-dessous, 3 sources de lumière différentes sont définies et donc disponibles pour éclairer la scène. L'une est située au point de vision tandis que les deux autres sont placées à des positions fixes. Deux sont positionnelles et une directionnelle. Une des sources positionnelles est un spot.

Deux objets sont définis : une pyramide dont les vecteurs normaux sont précisés pour chaque sommet et un cube obtenu à l'aide de l'instruction `glutSolidCube(longueurCoté)`. Les fonctions de la librairie GLUT définissant des objets 3D incluent la spécification des vecteurs normaux en chaque sommet de l'objet.

Comme d'habitude, les instructions n'apparaissant qu'une seule fois sont placées dans `init` tandis que les autres sont appelées directement ou indirectement dans `display`. Lorsqu'une caractéristique d'éclairage n'est pas précisée, c'est la valeur par défaut qui est utilisée.

Lorsqu'une composante de lumière est noire cela signifie que cette composante n'a pas d'effet dans le modèle d'éclairage.

```
...
static GLfloat position0[]={0.0,0.0,0.0,1.0};//source positionnelle
static GLfloat position1[]={5.0,6.0,0.0,1.0};//source positionnelle
static GLfloat position2[]={-2.0,1.0,0.0,0.0};//source directionnelle
...
void init(void)
// définition des caractéristiques des sources lumineuses
// et du modèle d'éclairage + display list pour la pyramide
{
   ...
   GLfloat blanc[]={1.0,1.0,1.0,1.0};
   GLfloat noir[]={0.0,0.0,0.0,1.0};
   GLfloat jaune[]={1.0,1.0,0.0,1.0};
   GLfloat bleuClair[]={0.0,1.0,1.0,1.0};
   GLfloat rouge[]={1.0,0.0,0.0,1.0};
   GLfloat directionSpot[]={0.0,-1.0,0.0};
   // lumière 0 : lumière ambiante jaune
   glLightfv(GL_LIGHT0, GL_AMBIENT, jaune);
   glLightfv(GL_LIGHT0, GL_DIFFUSE, noire);
   glLightfv(GL_LIGHT0, GL_SPECULAR, noire);
   // lumière 1 : lumière diffuse rouge + spéculaire blanche + spot
   glLightfv(GL_LIGHT1, GL_AMBIENT, noire);
   glLightfv(GL_LIGHT1, GL_DIFFUSE, rouge);
   glLightfv(GL_LIGHT1, GL_SPECULAR, blanc);
   glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, directionSpot);
   glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 10.0);
   glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 2.5);
   glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 3.0);
   // lumière 2 : lumière ambiante bleue claire
   glLightfv(GL_LIGHT2, GL_AMBIENT, bleuClair);

   // observateur local -> le point de vision est initialement en
   // (0.0,0.0,0.0) pour les calculs du modèle d'éclairage
   glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER,GL_TRUE);
   glEnable(GL_LIGHTING);// éclairage permanent
   glEnable(GL_LIGHT0);// lumière 0 permanente
```

```
    glNewList(pyramide,GL_COMPILE);
      glBegin(GL_TRIANGLES);// les 4 côtés de la pyramide
        glNormal3d(-1.0/sqrt(1.25),0.5/sqrt(1.25),0.0);
        glVertex3d(1.5,2.0,0.5);
        glVertex3d(1.0,1.0,0.0);
        glVertex3d(1.0,1.0,1.0);
        glNormal3d(0.0,0.5/sqrt(1.25),1.0/sqrt(1.25));
        glVertex3d(1.5,2.0,0.5);
        glVertex3d(1.0,1.0,1.0);
        glVertex3d(2.0,1.0,1.0);
        glNormal3d(1.0/sqrt(1.25),0.5/sqrt(1.25),0.0);
        glVertex3d(1.5,2.0,0.5);
        glVertex3d(2.0,1.0,1.0);
        glVertex3d(2.0,1.0,0.0);
        glNormal3d(0.0,0.5/sqrt(1.25),-1.0/sqrt(1.25));
        glVertex3d(1.5,2.0,0.5);
        glVertex3d(2.0,1.0,0.0);
        glVertex3d(1.0,1.0,0.0);
      glEnd();
      glBegin(GL_POLYGON);// la base de la pyramide
        glNormal3d(0.0,-1.0,0.0);
        glVertex3d(2.0,1.0,1.0);
        glVertex3d(1.0,1.0,1.0);
        glVertex3d(1.0,1.0,0.0);
        glVertex3d(2.0,1.0,0.0);
      glEnd();
    glEndList();
    ...
}

  void display(void)
  {
    ...
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // lumière 0, positionnelle, superposée au point de vision
    glLightfv(GL_LIGHT0,GL_POSITION,position0);
    gluLookAt(...);
```

```
    // lumières 1 (positionnelle) et 2 (directionnelle) fixes
    glLightfv(GL_LIGHT1,GL_POSITION,position1);
    glLightfv(GL_LIGHT2,GL_POSITION,position2);
    ...

    // propriétés et placement du cube, éclairé par les lumières 0 et 1

    glEnable(GL_LIGHT1);
    // absorption totale du rouge et réflexion totale du vert et du bleu pour
    // les composantes ambiante et diffuse
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,bleuClair);

    // réflexion totale du rouge et absorption totale du vert et du bleu pour
    // la composante spéculaire
    glMaterialfv(GL_FRONT,GL_SPECULAR,rouge);
    glMaterialf(GL_FRONT,GL_SHININESS,25.0);
    glPushMatrix();
    glTranslated(5.0,4.0,0.0);
    glutSolidCube(3.0);
    glPopMatrix();
    glDisable(GL_LIGHT1);

    // propriétés de la pyramide, éclairée par les lumières 0 et 2

    glEnable(GL_LIGHT2);
    // absorption totale de toutes les couleurs pour la composante diffuse
    glMaterialfv(GL_FRONT,GL_DIFFUSE,noir);
    // réflexion totale de toutes les couleurs pour la composante spéculaire
    glMaterialfv(GL_FRONT,GL_SPECULAR,blanc);
    glMaterialf(GL_FRONT,GL_SHININESS,100.0);
    glCallList(pyramide);
    glDisable(GL_LIGHT2);
    ...
}
...
```

L'exemple ci-dessus a pour but d'illustrer la marche à suivre pour mettre en oeuvre un modèle d'éclairage. Le lecteur curieux pourra tester cet exemple sur machine et essayer de comprendre le résultat obtenu pour différents points de visions.

9.7.Principes de calcul d'un modèle d'éclairage

Dans cette section, nous décrivons brièvement quelques unes des règles suivies par OpenGL pour calculer l'intensité de chacune des trois composantes de couleur RGB en chaque sommet utilisé dans la définition de la scène. Si le modèle de couleur est GL_SMOOTH, l'intensité en un point quelconque d'une primitive graphique sera calculée en interpolant les intensités aux différents sommets définissant la primitive. S'il sagit de GL_FLAT, c'est soit l'intensité du premier sommet (pour GL_POLYGON), soit celle du dernier sommet (pour tous les autres types de primitive) qui définit l'intensité de la primitive graphique dans son ensemble. Il est aussi à noter qu'OpenGL ne tient pas compte du fait que certains objets en masquent d'autres vis-à-vis d'une source de lumière, et a fortiori ne gère pas les ombres. OpenGL ne tient pas non plus compte de la lumière reflétée par un objet sur d'autres objets.

Chaque composante de couleur RGB est traitée séparément et les calculs suivent les mêmes formules pour chacune d'elles. De manière générale, la couleur en un sommet est calculée comme la somme des contributions de sa composante émissive, de la composante ambiante générale et des composantes ambiante, diffuse et spéculaire de chaque source en ce sommet. Mise à part celle de la composante émissive, chaque contribution dépend à la fois d'une source et de la caractéristique d'un matériau en un sommet : l'intensité d'une composante du modèle est calculée comme le produit de l'intensité de la source par celle qui caractérise la réflexion du matériau. Le résultat du calcul global est ramené à 0 s'il est négatif et à 1 s'il est supérieur à 1.

La contribution de la composante émissive est simplement l'intensité de cette composante.

La contribution de la composante ambiante globale est le produit de l'intensité de cette composante par l'intensité correspondante de réflexion par le matériau. Si la composante ambiante globale est caractérisée par (R1, G1, B1) et la composante ambiante du matériau par (R2, G2, B2), la résultante sera (R1∗R2, G1∗G2, B1∗B2). Le même principe est appliqué pour les trois composantes d'une source de lumière.

La contribution de la composante ambiante d'une source est le produit de l'intensité de la source par l'intensité de réflexion du matériau. Pour la contribution de la composante diffuse d'une source, le produit des intensités de la source et du matériau est multiplié par un facteur quantifiant dans quelle mesure la lumière frappe de manière directe le sommet éclairé. Pour la contribution de la composante spéculaire d'une source, le produit des intensités de la source et du matériau est multiplié par un facteur quantifiant dans quelle mesure la lumière frappe de manière directe le sommet éclairé et tenant compte du caractère concentré de l'éclat de lumière produit et du caractère local ou non de l'observateur.

La contribution globale d'une source de lumière en un sommet est calculée comme la somme des contributions des composantes ambiante, diffuse et spéculaire, le tout multiplié par le facteur d'atténuation dans le cas d'une source positionnelle, et par un facteur résumant l'effet de spot le cas échéant. Si la source est un spot, ce dernier facteur prend en compte le fait que le sommet éclairé soit à l'intérieur du cône de lumière ou pas, ainsi que sa distance au centre du cône et la variation d'intensité de la source lorsque l'on s'éloigne du centre du cône.

En résumé, on peut écrire pour un sommet donné :

$$intensité = emissif_{matériau} + ambiant_{global} * ambiant_{matériau} + \sum_{i=0}^{n} \left(\frac{1}{k_c + k_l d + k_q d^2} \right)_i * (effetSpot)_i * \left[\begin{array}{l} ambiant_{source} * ambiant_{matériau} \\ + facteur_{diffus} * diffus_{source} * diffus_{matériau} \\ + facteur_{spéculaire} * spéculaire_{source} * spéculaire_{matériau} \end{array} \right]_i$$

où l'indice *i* indexe les différentes sources lumineuses actives du modèle d'éclairage.

10.Sélection par intersection d'un volume avec le modèle

Il est souvent intéressant de pouvoir déterminer si tel ou tel objet d'un modèle 3D est contenu partiellement dans un volume spécifique de l'espace. Un exemple classique est de pouvoir détecter si l'utilisateur a sélectionné tel ou tel objet dans l'image rendue à l'écran. La complexité d'une telle détection augmente avec celle du modèle, pour devenir rapidement un problème réservé à l'expert.

Heureusement, OpenGL propose des outils permettant de systématiser une telle démarche. Ils permettent de déterminer rapidement quelles primitives graphiques entrent en contact avec un volume donné. Bien que ces outils offrent rarement la solution optimale, néanmoins le processus est suffisamment rapide pour de nombreuses applications.

Nous décrirons d'abord ces outils en dégageant les principes de fonctionnement du procédé. Nous passerons ensuite en revue les différentes étapes en OpenGL. Nous verrons finalement comment appliquer ce processus de sélection au problème de l'interaction avec l'utilisateur via la souris et nous illustrerons ce cas de figure par un exemple simple.

10.1.Le principe

On définit donc à un moment donné un volume spécifique d'intérêt, et l'on désire savoir si tel ou tel objet de la scène se trouve dans ce volume à ce moment précis. Pour cela, on va redéfinir des primitives graphiques qui interviennent dans la construction des objets cibles et tester si elles entrent en intersection avec le volume d'intérêt. Lorsqu'il y a intersection, une série d'informations utiles sera recueillie concernant les primitives concernées. Ces informations seront ensuite analysées pour en déduire quels objets cibles se trouvaient dans le volume d'intérêt et effectuer les actions correspondantes le cas échéant. Dans le cas d'une interaction avec l'utilisateur, une action simple pourrait être la modification de la couleur d'un objet sélectionné à l'aide de la souris.

La spécification du volume d'intérêt peut consister à se donner 6 plans traversant l'espace et délimitant ce volume. Il faudra aussi spécifier pour chaque plan dans quelle direction se trouve l'intérieur du volume d'intérêt. Comme OpenGL permet de définir un volume à projeter par exemple via `glOrtho(...)`, on pourra plus simplement utiliser cette instruction pour définir le volume d'intérêt. Ce dernier sera le plus souvent différent du volume projeté à l'écran.

Pour déterminer si telle ou telle primitive graphique est en intersection avec le volume d'intérêt, on leurs associera des identifiants entiers. Un tel identifiant sera appelé *nom* de la primitive graphique. L'approche que nous allons décrire consiste à stocker ces noms dans une pile, appelée *pile des noms*, et à récupérer le contenu de la pile dans une mémoire tampon lorsque les primitives dont les noms sont dans la pile sont en intersection avec le volume d'intérêt. Le processus d'ajoût et de retrait d'un objet dans une structure de type pile est particulièrement approprié pour traité des modèles hiérarchisés tel que celui présenté au chapitre 7.

Il faudra tester si chaque primitive graphique est en intersection avec le volume d'intérêt. Dans la suite, nous parlerons de *touche* pour désigner le fait que le test est positif (c'est-à-dire lorsqu'il y a intersection). Cette opération de test n'est pas triviale, heureusement OpenGL permet de faire ces tests automatiquement. OpenGL permet en effet de passer du mode de *rendu*, utilisé lorsque l'on veut afficher une portion de scène à l'écran, au mode de *sélection*,

spécifique aux opérations de sélection des objets présents dans un volume donné. Le mode de rendu est le mode par défaut, que nous avons exclusivement utilisé jusqu'à présent. Lorsque l'on est en mode de sélection, OpenGL teste automatiquement si les primitives graphiques sont "sélectionnées" dans le volume d'intérêt et quand c'est le cas, OpenGL envoie de manière tout aussi automatique une série d'informations, encore appelée *enregistrement*, vers une zone tampon pré-définie.

Cette zone tampon, appelée *buffer de selection*, reçoit un enregistrement chaque fois que la pile des noms est manipulée ou que l'on quitte le mode de sélection et qu'une touche a eu lieu depuis la dernière manipulation de la pile des noms ou depuis l'entrée en mode de sélection. Si un objet est composé de plusieurs primitives on pourra faire une seule manipulation sur la pile concernant cet objet : ainsi, un seul enregistrement sera envoyé vers le buffer de sélection si l'objet est en intersection avec le volume d'intérêt, même si plusieurs des primitives graphiques le composant ont fait une touche. A l'inverse, il sera possible d'identifier une primitive graphique par une combinaison de plusieurs noms en poussant les noms correspondants dans la pile avant l'appel de la définition de la primitive. Chaque enregistrement contient les informations suivantes :

- le nombre de noms présents dans la pile de noms au moment de la touche ;
- les profondeurs minimum et maximum des sommets des primitives ayant fait une touche depuis le dernier enregistrement ;
- le contenu de la pile des noms au moment de la touche, en démarrant par le nom se trouvant le plus bas dans la pile.

Lorsque l'on quitte le mode de sélection, il reste à analyser l'information recueillie dans le buffer de sélection et agir en conséquence.

10.2.La pile des noms

La pile des noms est disponible en OpenGL uniquement en mode de sélection. Il s'agit d'une structure de donnée de type pile classique permettant de stocker des entiers non signés. L'instruction `glInitNames()` initialise la pile, c'est-à-dire qu'une pile vide est mise à disposition. L'instruction `glPushName(name)` permet de pousser l'entier non signé `name` dans la pile tandis que son pendant `glPopName()` permet de retirer le nom se trouvant en haut de la pile. Finalement, l'instruction `glLoadName(name)` permet de remplacer le nom se trouvant en haut de la pile par `name`. Ces instructions sont sans effet lorsque l'on n'est pas en mode de sélection.

Imaginons que l'on ait une série de 5 objets définis dans 5 *display lists* consécutives démarrant avec l'identifiant `firstID`, et que l'on désire identifier ces objets par les entiers 1 à 5. Le code suivant permettra de placer dans la pile le nom de l'objet dont la *display list* est appelée, et lui seul, à chaque appel :

```
...
int i;
glInitNames();
glPushName(0);// place un nom quelconque en haut de la pile
...
```

```
for(i=0;i<5;i++){
   glLoadName(i+1);// échange le nom en haut de la pile avec "i+1"
   glCallList(firstID + i);
}
```

10.3.Le mode de sélection et son buffer

Le passage en mode de sélection se fait via `(void) glRenderMode(GL_SELECT);` tandis que le retour au mode de rendu se fait via `nbTouches = glRenderMode(GL_RENDER);`. Lorsque l'on quitte le mode de sélection pour retourner au mode de rendu, `glRenderMode(...)` renvoie un entier représentant le nombre de touches qui ont eu lieu pendant que le mode de sélection était activé.

Avant d'entrer en mode de sélection, il faut créer un tableau d'entiers non signés de taille suffisante, puis indiquer à OpenGL que ce tableau doit être utilisé comme buffer de sélection via `glSelectBuffer(taille, buffer)`, où `taille` est la dimension du tableau et `buffer` un pointeur vers le tableau. La dimension du tableau doit être suffisamment grande pour qu'il puisse contenir l'ensemble des enregistrements qui seront créés lors des touches. On aura donc typiquement :

```
GLuint bufferSelection[TAILLE];
glSelectBuffer(TAILLE, bufferSelection);
```

Si la dimension du buffer de sélection est insuffisante, `glRenderMode(GL_RENDER)` renverra un entier négatif.

Une fois que l'on est en mode de sélection, il faut définir le volume d'intérêt à l'aide d'une des 3 instructions disponibles (`glOrtho`, `glFrustum` ou `gluPerspective`), en mode de projection. On passe ensuite à la matrice de *modelview*. Il est alors **primordial** d'appeler l'instruction `gluLookAt(...)` adaptée car c'est elle qui positionne le point de vue et ce dernier est utilisé pour obtenir le volume de projection. Ensuite on définit les primitives graphiques à tester en conjugaison avec les transformations géométriques nécessaires, tout en manipulant la pile des noms. Lorsque toutes les primitives à tester ont été définies, on retourne en mode de rendu.

Dans le cas des 5 objets définis dans 5 *display lists* nous pourrions écrire :

```
...
GLuint bufferSelection[TAILLE];
...
glSelectBuffer(TAILLE, bufferSelection);
(void) glRenderMode(GL_SELECT);// passage en mode de sélection
glInitNames();
glPushName(0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(...);// définition du volume d'intérêt
```

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(...);
...
for(i=0;i<5;i++){
   glLoadName(i+1);
   glCallList(firstID + i);
}
glFlush();// force l'exécution de toutes les instructions déjà passées
nbTouches=glRenderMode(GL_RENDER);// retour en mode de rendu
...
```

Imaginons que les objets correspondants à la 1ière et à la 3ième *display lists* fassent une touche, quelle information trouverons-nous dans le buffer de sélection ? On trouvera les valeurs suivantes pour les 8 premiers éléments du tableau : 1; profondeurMinObjet1; profondeurMax-Objet1; 1; 1; profondeurMinObjet3; profondeurMaxObjet3; 3. La variable `nbTouches` aura quant à elle la valeur 2. On parcourt le tableau séquentiellement pour analyser le résultat de la sélection et récupérer l'information utile. Voyons ce que donne l'analyse point par point dans notre cas :

- `nbTouches` = 2 => 2 objets sélectionnés ;
- `bufferSelection[0]` = 1 => 1 nom présent dans la pile pour la première touche ;
- `bufferSelection[1]` = profondeur minimum du premier objet sélectionné ;
- `bufferSelection[2]` = profondeur maximum du premier objet sélectionné ;
- `bufferSelection[3]` = 1 => le premier objet sélectionné est l'objet 1 – ceci termine l'enregistrement concernant la première touche puisqu'il n'y avait qu'un seul nom présent dans la pile pour la première touche ;
- `bufferSelection[4]` = 1 => 1 nom présent dans la pile pour la 2ième touche ;
- `bufferSelection[5]` = profondeur minimum du 2ième objet sélectionné ;
- `bufferSelection[6]` = profondeur maximum du 2ième objet sélectionné ;
- `bufferSelection[7]` = 3 => le 2ième objet sélectionné est l'objet 3 – ceci termine l'enregistrement concernant la 2ième touche puisqu'il n'y avait qu'un seul nom présent dans la pile pour la 2ième touche – ceci termine aussi l'information utile présente dans le buffer de sélection puisque le nombre total de touches est de 2.

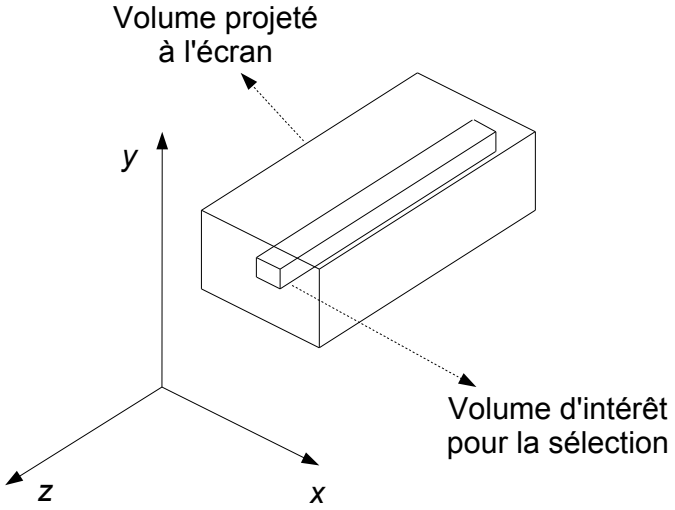
Remarques :

- *dans l'exemple ci-dessus, les 2 enregistrements sont envoyés vers le buffer de sélection lorsque l'instruction* `glLoadName(i+1)` *est rencontrée dans la boucle* `for(...)`*, aux moments où l'indice* `i` *prend les valeurs 1 et 3, respectivement ;*
- *les profondeurs présentes dans les enregistrements sont des entiers compris entre 0 et 2^{32}-1. Il s'agit chaque fois de la profondeur d'un sommet d'une primitive graphique, après transformation en coordonnées relatives au volume canonique de vue (ce qui donne un nombre entre 0 et 1, cf. section 6.2.3.), multipliée par 2^{32}-1 puis arrondie à l'entier non*

signé le plus proche.

10.4.Sélection à l'aide de la souris

Pour déterminer si certains objets sont sélectionnés à l'aide d'un clic de la souris, on conditionnera le passage au mode de sélection au type de clic attendu. Par exemple, on entre en mode de sélection lorsque l'utilisateur presse le bouton de gauche de la souris, ou bien lorsqu'il relache le bouton de droite... Le volume d'intérêt aura la forme d'un parallèlépipède rectangle centré à l'endroit sélectionné via la souris, et s'étendant sur toute la profondeur du volume projeté à l'écran sur le rayon de projection correspondant. Les dimensions de la base correspondront typiquement à une zone de quelques pixels autour du point sélectionné.



OpenGL possède une instruction GLU qui construit une matrice de projection permettant de restreindre le volume projeté à l'écran au volume d'intérêt décrit ci-dessus. Il s'agit de `gluPickMatrix(x, y, largeur, hauteur, viewport)`, où `x` et `y` définissent le centre de la base en coordonnées d'écran (par rapport à une origine au coin inférieur gauche de la fenêtre d'écran), `largeur` et `hauteur` définissent la largeur et la hauteur de la base en pixels, et `viewport` est un tableau contenant les caractéristiques de la viewport courante.

Lorsque `gluPickMatrix()` précède la définition du volume à projeter en mode de sélection, OpenGL considère le volume d'intérêt décrit ci-dessus pour tester si chaque primitive graphique d'un objet dont il rencontre la définition fait une touche.

Le code prendra typiquement la structure suivante :

```
...
void picking(int button, int state, int x, int y)
{
   ...
   Glint viewport[4];
   ...
   if(button != ... || state != ...) return;
   glGetIntergerv(GL_VIEWPORT, viewport); // si une seule viewport
```

```
    ...
    (void) glRenderMode(GL_SELECT);
    ...
    glMatrixMode(GL_PROJECTION);
    glPushMatrix(); // sauvegarde matrice de projection
    glLoadIdentity();
    // sélection dans une zone carrée de côté 5 pixels autour du click
    gluPickMatrix((GLdouble)x,(Gldouble)(h_global-y),5.0,5.0,viewport);
    // définition du volume à projeter, généralement identique à celui
    // utilisé en mode de rendu
    gluOrtho2D(...);// par exemple
    glPopMatrix(); // recuperation matrice de projection

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix(); // sauvegarde matrice de modelview
    glLoadIdentity();
    // gluLookAt(...) approprié
    // transformations géométriques et définitions d'objets +
    // manipulations de la pile des noms
    glPopMatrix(); // recuperation matrice de modelview

    glFlush();
    nbTouches = glRenderMode(GL_RENDER);
    ...
    glutPostRedisplay();//si la sélection entraîne des modifications
}
...
int main(int argc, char** argv)
{
    ...
    glutMouseFunc(picking);
    ...
}
```

Lorsque plusieurs objets sont sélectionnés simultanément parce qu'ils sont disposés à des profondeurs différentes à l'endroit du click de la souris, on pourra utiliser l'information relative à la profondeur dans le buffer de sélection pour décider quel objet l'utilisateur a réellement eu l'intention de sélectionner.

10.5.Un exemple illustratif

Reprenons l'exemple du carrousel proposé au chapitre 7. Cette fois-ci les nacelles sont toutes identiques, seules leurs couleurs diffèrent : les 3 nacelles d'une même traverse ont la même couleur, mais cette couleur est différente sur chacune des 3 traverses. Lorsque l'utilisateur sélectionne une nacelle à l'aide de la souris, les 3 nacelles de la traverse concernée changent de couleur.

```
...
int couleur[3]={0,0,0};// facteurs pour couleurs des nacelles
Glfloat hauteur[3]={2.0,3.0,4.0};
...
void dessineCarrousel(float hauteur[], Glenum mode)
// mode indique si l'on est en mode de rendu ou de sélection
{
   int i;
   glPushMatrix();
   glTranslated(0.0,-10.0,0.0);
   dessineMat();
   glPopMatrix();
   for(i=0;i<3;i++){
      glPushMatrix();
      if(mode == GL_SELECT) glLoadNames(i);// un nom identifiant la traverse
      glRotated(120.0*i,0.0,1.0,0.0);
      dessineTraversePlusNacelles(hauteur, i)
      glPopMatrix();
   }
...
void dessineTraversePlusNacelles(float hauteur[], int traverseID)
{
   int i;
   glPushMatrix();
   dessineTraverse();

   for(i=0;i<3;i++){
      glTranslated(5.0,0.0,0.0);
      glPushMatrix();
      glTranslated(0.0,-hauteur[i],0.0);
      glPushMatrix();
      glScaled(1.0,hauteur[i],1.0);
```

```
      dessineRaccordVertical();
      glPopMatrix();
      // Définit la couleur de la nacelle
      if(traverseID == 0) // nuance de rouge
        glColor3f((GLfloat)(couleur[traverseID]+1)/3.0,0.0,0.0);
      else if(traverseID == 1) // nuance de vert
        glColor3f(0.0,(GLfloat)(couleur[traverseID]+1)/3.0,0.0);
      else // nuance de bleu
        glColor3f(0.0,0.0,(GLfloat)(couleur[traverseID]+1)/3.0);
      dessineNacelle();
      glPopMatrix();
    }

    glPopMatrix();
}
...
void changeCouleur(GLuint bufferSelection[])
// Change la couleurs des nacelles
{
  GLuint i=bufferSelection[3];// Max 1 touche et 1 nom par touche
  // Change l'intensité de la couleur pour les nacelles concernées
  couleur[i]=(couleur[i]+1)%3;
}
...
#define TAILLE 64 // Dimension du buffer de sélection
...
void picking(int button, int state, int x, int y)
{
  ...
  Glint nbTouches, viewport[4];
  GLuint bufferSelection[TAILLE];
  ...
  // Sélection par enfoncement du bouton de gauche de la souris
  if(button != GLUT_LEFT_BUTTON || state != GLUT_DOWN) return;
  glGetIntergerv(GL_VIEWPORT, viewport); // si une seule viewport
  ...
  glSelectBuffer(TAILLE, bufferSelection);
  (void) glRenderMode(GL_SELECT);
  glInitNames();
```

```
    glPushName(0);
    ...
    glMatrixMode(GL_PROJECTION);
    glPushMatrix(); // sauvegarde matrice de projection
    glLoadIdentity();
    gluPickMatrix((GLdouble)x,(Gldouble)(h_global-y),5.0,5.0,viewport);
    gluOrtho2D(...); // définition du volume à projeter,
                     // identique à celui présent dans "viewport"
    glPopMatrix(); // recuperation matrice de projection

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix(); // sauvegarde matrice de modelview
    glLoadIdentity();
    gluLookAt(...); // identique à celui présent dans "viewport"
    ...
    // définition du modèle en mode de sélection
    dessineCarrousel(hauteur, GL_SELECT);
    glPopMatrix(); // recuperation matrice de modelview

    glFlush();
    nbTouches = glRenderMode(GL_RENDER);
    if(nbTouches==1) {
      changeCouleur(bufferSelection);
      glutPostRedisplay();// rend le changement de couleur effectif
    }
}
...
void display(void)
{
    ...
    // définition du modèle en mode de rendu
    viewport();
    dessineCarrousel(hauteur, GL_RENDER);
    ...
}
...
```

```
void viewport()
{
   ...
   glMatrixMode(GL_POJECTION);
   glLoadIdentity();
   gluOrtho2D(...); // identique à celui présent dans "picking"
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(...); // identique à celui présent dans "picking"
}
...
int main(int argc, char** argv)
{
   ...
   glutMouseFunc(picking);
   ...
}
```

Chaque fois que l'utilisateur sélectionnera une nacelle, les 3 nacelles de la traverse concernée changeront de couleur comme suit : il s'agira d'une des 3 couleurs fondamentales dont l'intensité passera de 1/3 à 2/3, de 2/3 à 1 ou de 1 à 1/3 suivant l'intensité courante.