TP4 : Synthèse d'images - OpenGL

TP à faire sous PC-Linux par binôme.
Compte-rendu à envoyer à l'adresse `nicolas.szafran@univ-grenoble-alpes.fr` avant le 31 mars 2017 en indiquant comme sujet de l'e-mail : [L3-Info] - Image - TP4 - *noms du binôme*

Récupérez le fichier archive `tp4.tar` à l'URL suivante :
http://www-ljk.imag.fr/membres/Nicolas.Szafran/ENSEIGNEMENT/L3
puis décompactez l'archive avec la commande `tar -xvf tp4.tar` .

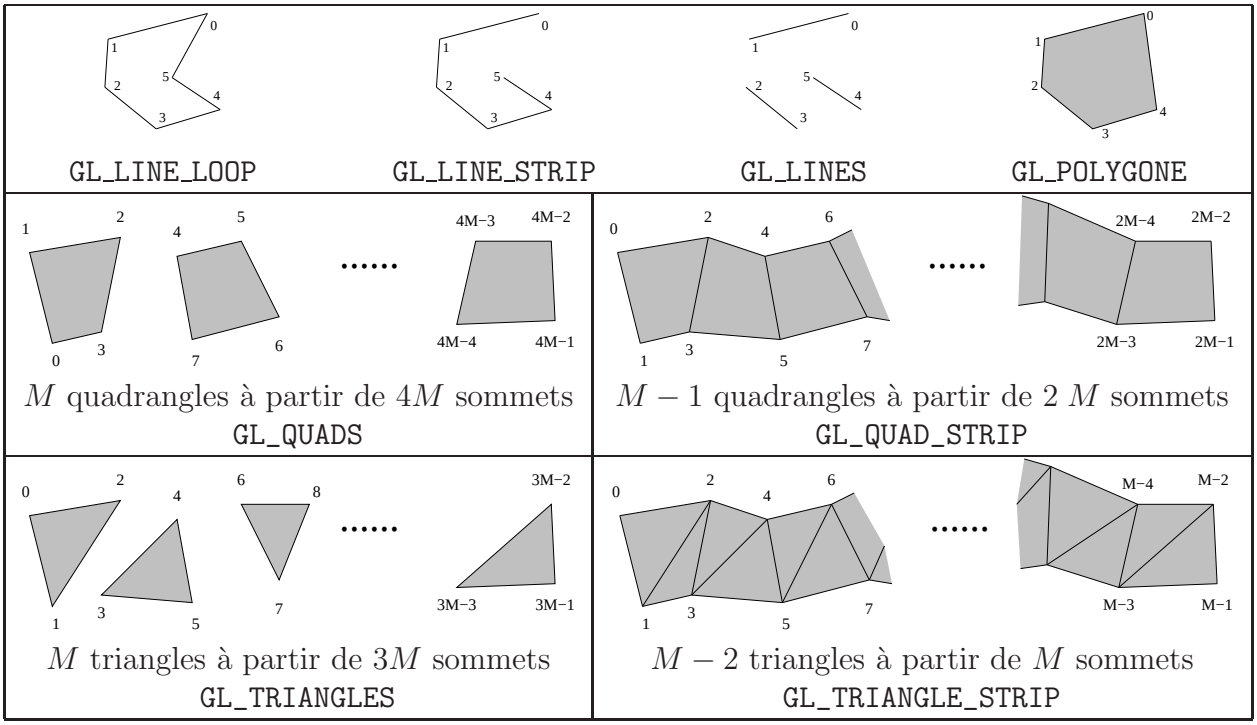
Pour le compte rendu de ce TP, vous ferez :
– une archive compactée avec un seul répertoire contenant l'ensemble de vos programmes sources, le fichier `Makefile`, **mais sans** les images au format PPM fournies dans l'archive `tp4.tar`,
– un rapport au format PDF en indiquant les exercices que vous avez traités, un commentaire sur chacun, et/ou les difficultés rencontrées pour tel ou tel exercice ou partie d'exercice.

---

1 - Dessin dans le plan

Dans cette partie, les différentes scènes seront définies dans le plan $\{z = 0\}$.

1.1 - Les modes de tracé



Dans cette partie, vous allez tester les différents modes graphiques en dessinant dans le plan $z = 0$.

• Le programme simple_rectangle.cpp est le programme donné en exemple en cours.
Dans la fonction `dessin`, l'instruction

```
glRectd(-25.0, -25.0, 25.0, 25.0);
```

définit un rectangle se trouvant dans le plan $z = 0$, ces 4 sommets sont $(-25, -25, 0)$, $(25, -25, 0)$, $(25, 25, 0)$ et $(-25, 25, 0)$.

• Pour tracer le bord du rectangle en rouge avec une épaisseur de 5 pixels, ajoutez les instructions suivantes après l'instruction `glRectf(-25.0, -25.0, 25.0, 25.0);`

```
glColor3d(1.0,0.0,0.0);
glLineWidth(5);
glBegin(GL_LINE_LOOP);
        glVertex3d(-25.0,-25.0,0.0);
        glVertex3d( 25.0,-25.0,0.0);
        glVertex3d( 25.0, 25.0,0.0);
        glVertex3d(-25.0, 25.0,0.0);
glEnd();
```

L'instruction `glLineWidth(e)` permet de définir l'épaisseur du tracé avec l'épaisseur e.
Les instructions `glBegin(GL_LINE_LOOP)` et `glEnd()` permettent de définir un contour polygonal fermé dont les différents sommets sont définis à l'aide des instructions `glVertex3d`.

• L'instruction `glBegin(GL_POLYGON)` permet de remplir un polygone *convexe* en spécifiant ces différents sommets ordonnés.
Compilez et exécutez le programme simple_cercle.cpp : un cercle rempli en vert avec un bord blanc est affiché.
Dans la fonction `dessin`, le disque vert est discrétisé à l'aide d'un polygone régulier convexe (`GL_POLYGON`) avec $N = 100$ sommets et le contour circulaire est discrétisé à l'aide d'une ligne polygonale fermée (`GL_LINE_LOOP`) avec les mêmes sommets.

• Remplacez l'instruction `glBegin(GL_LINE_LOOP)` par l'instruction `glBegin(GL_LINE_STRIP)` :
le contour n'est pas fermé (les premier et dernier sommets ne sont pas reliés).

• Remplacez l'instruction
`glBegin(GL_LINE_STRIP)`
par l'instruction
`glBegin(GL_LINES)` :
seul un segment sur deux est tracé.
• Pour dessiner des polygones non convexes, il faut les décomposer en triangles ou quadrangles convexes.
Compilez et exécutez le programme anneau_circulaire.cpp : dans la fonction `dessin`, l'instruction `glBegin(GL_QUAD_STRIP)` permet de définir une bande de quadrangles.
• Remplacez l'instruction `glBegin(GL_QUAD_STRIP)` par l'instruction `glBegin(GL_TRIANGLE_STRIP)` et l'instruction `glBegin(GL_LINE_LOOP)` par l'instruction `glBegin(GL_LINE_STRIP)` : le polygone est décomposé comme une bande de triangles.

L'instruction `glBegin(GL_TRIANGLE_STRIP)` permet de définir une bande de triangles.
• On peut utiliser l'instruction `glBegin(GL_QUADS)` (resp. `glBegin(GL_TRIANGLES)`) pour tracer une liste quelconque de quadrangles (resp. triangles).

1.2 - Correspondance entre espace réel et image bitmap

Reprenez le programme simple_cercle.cpp.

L'instruction `glViewport (0, 0, (GLsizei) w, (GLsizei) h)` définit la taille de la fenêtre et donc la taille de l'image bitmap finale (`w` × `h`). Au démarrage du programme, la dimension initiale est donnée par l'instruction `glutInitWindowSize(w,h)`.

Les coordonnées (réels de type `GLdouble`) des différents tracés sont définies dans l'espace réel.

Par défaut la projection se fait suivant l'axe Oz et la correspondance entre le repère de l'espace réel et le repère de l'image se fait par l'intermédiaire de l'instruction
`glOrtho(xmin, xmax, ymin, ymax, zmin, zmax)` suivant le tableau suivant :

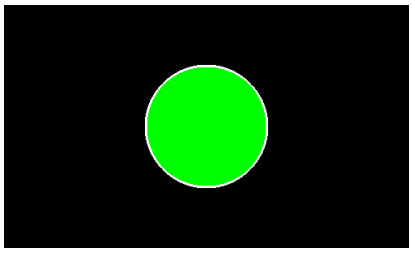
Coordonnée réelle	Coordonnée image	Coordonnée réelle	Coordonnée image
xmin	0	xmax	w
ymin	0	ymax	h

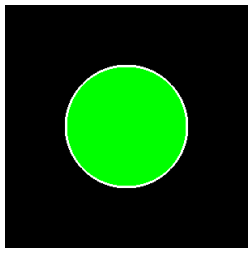
Dans le programme, le cercle est dessiné dans le plan $z = 0$ donc il suffit de choisir `zmin` $< 0 <$ `zmax`.

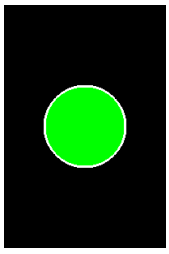
Exercice 1 :

Exécutez le programme `simple_cercle` et modifiez à la souris les dimensions de la fenêtre : le cercle devient un ovale si la fenêtre n'est plus carrée car le rapport $\rho = \frac{\texttt{(double)w}}{\texttt{(double)h}}$ des dimensions de la fenêtre n'est plus égal au rapport $\frac{\texttt{xmax} - \texttt{xmin}}{\texttt{ymax} - \texttt{ymin}} = \frac{50-(-50)}{50-(-50)} = 1$.
Donc dans la procédure `redimensionnement`, en fonction du rapport ρ, il faut adapter les valeurs des bornes `xmin`, `xmax`, `ymin` et `ymax`.

Modifiez le programme en adaptant les paramètres de l'instruction `glOrtho` (en fonction de `w` et `h`) de manière à ce que quelque soient les dimensions de la fenêtre, le cercle ne soit pas déformé en ovale, qu'il soit centré dans la fenêtre, et que la plus petite dimension de la fenêtre corresponde à deux fois le diamètre du cercle.


Cas $w > h$


Cas $w = h$


Cas $w < h$

2 - Transformations géométriques

Dans cette partie, on va s'intéresser aux transformations géométriques concernant le tracé et voir la manipulation de la piles de matrices correspondante.

2.1 - Piles de matrices en OpenGL

Comme on l'a vu en cours, OpenGL utilise le principe des coordonnées homogènes et des transformations linéaires dans $\mathbb{R}^4$, les combinaisons de ces transformations se faisant par produit(s) de matrices 4×4.

Les différentes matrices sont stockées dans différentes piles.

Deux piles nous intéressent plus particulièrement :

— la pile de modélisation-visualisation (`GL_MODELVIEW`) :
cette pile de matrice permet de prendre en compte toutes les transformations de rotation (`glRotate*`),

de translation (`glTranslate*`) et d'homothétie (`glScale*`), la transformation due au positionnement (du repère) de la caméra par rapport à la scène (`gluLookat`) ainsi que les empilements et dépilements (`glPushMatrix()` et `glPopMatrix()`).
Pour qu'OpenGL utilise cette pile, il faut lui indiquer par l'instruction
`glMatrixMode(GL_MODELVIEW);`

— la pile de projection (`GL_PROJECTION`) : cette pile de matrice permet de prendre en compte toutes les transformations liées aux projections utilisées, projection perspective (`glFrustum` - `gluPerspective`) ou orthographique (`glOrtho`).
En général, pour la projection on n'utilise qu'une seule matrice.
Pour qu'OpenGL utilise cette pile, il faut lui indiquer par l'instruction
`glMatrixMode(GL_PROJECTION);`

Pour calculer la position de chacun des sommets dans l'image finale, OpenGL utilise les matrices courantes `GL_MODELVIEW` et `GL_PROJECTION` ainsi que les dimensions de la fenêtre définies par la fonction `glViewPort`.

En général, tout redimensionnement port graphique OpenGL, changement de projection ou changement de repère caméra doit réinitialiser ces deux matrices suivant le schéma suivant :

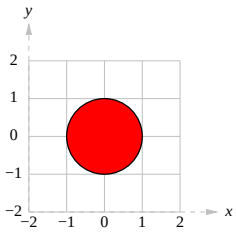
```
// fonction de redimensionnement de la fenetre graphique
void redimensionnement(int largeur_fenetre, int hauteur_fenetre)
{
        // redimensionnement du port graphique OpenGL
        glViewport (0, 0, largeur_fenetre, hauteur_fenetre);
        // initialiser les transformations géométriques dans le repère caméra
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        // choix de la perspective
        glOrtho(...); // ou gluPerspective(...)
        // positionnement de la caméra
        gluLookAt(...);
        ...
}

// fonction de dessin de la scène 3D
void dessin()
{
        // initialiser les transformations géométriques dans le repère scène
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        ...
}
```

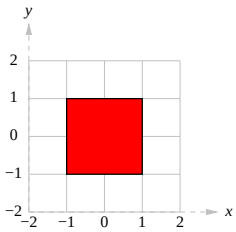
2.2 - La création de la scène à partir de primitives géométriques

Compilez et exécutez le programme pantins.cpp : ce programme crée une scène formée de différents pantins, chaque pantin étant créé à partir de deux formes géométriques de base, le cercle unité et le carré unité centrés à l'origine.

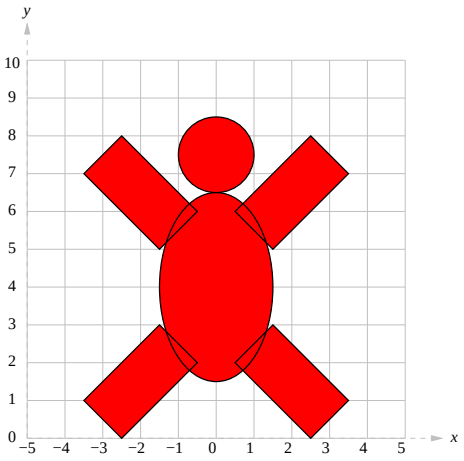
Regardez d'abord la fonction `pantin` pour comprendre comment la forme du pantin (*Figure 1* ci-dessous) est construite à partir des deux primitives de dessin de base, `cercle` et `carre`.



le cercle unité

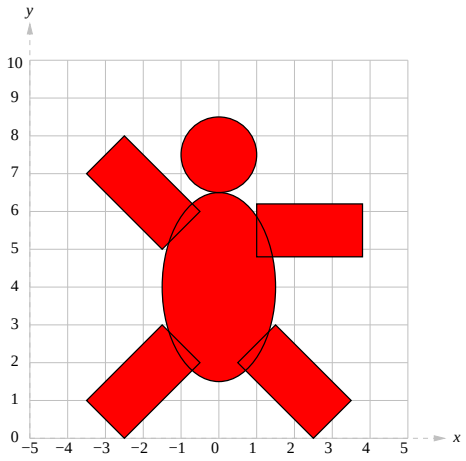


le carré unité



le pantin initial

Figure 1



le pantin modifié

Figure 2

Exercice 2 :

1. que se passe-t-il si on modifie le dessin du carré, par exemple en modifiant les coordonnées d'un sommet dans la fonction `carre` ?
2. modifiez le programme pour que le pantin bleu passe à la gauche du pantin rouge sans modifier la position des autres pantins.
3. modifiez le programme pour que le bras droit de chaque pantin soit horizontal (*Figure 2* ci-dessus).
4. modifiez le programme pour rajouter un pantin en l'air.
5. modifiez le programme pour ajouter un chapeau jaune à chaque pantin.

2.3 - Prise en compte de la profondeur (axe Oz) dans le dessin

Les différents exemples précédents utilisent une vue d'objets dans le plan $z = 0$. Pour la visualisation de scène en 3D, il faut indiquer à OpenGL de prendre en compte la profondeur (valeur suivant l'axe Oz de la caméra).

Compilez et exécutez le programme `cubes3d.c` : ce programme affiche deux cubes, chaque face de chaque cube avec une couleur différente.

Dans ce programme, une gestion des touches du clavier est faite à l'aide de la *fonction de rappel* `clavier` (et l'utilisation de la fonction `glutKeyboardFunc` dans la fonction `main`) afin d'effectuer différentes actions :

- les touches `x` et `Maj-x` permettent la rotation autour de l'axe horizontal de l'écran,
- les touches `y` et `Maj-y` permettent la rotation autour de l'axe vertical de l'écran,
- les touches `z` et `Maj-z` permettent la rotation autour de l'axe perpendiculaire à l'écran,
- les touches `+` et `-` permettent un zoom avant ou arrière des cubes,
- les touches `q` et `Maj-q` et `Echap` permettent de quitter le programme.

Testez les différentes actions correspondant aux différentes touches.

Vous remarquez que lorsque les cubes tournent, certaines faces qui devraient être cachées ne le sont pas, et un cube apparait toujours devant l'autre. En effet, dans ce programme, les faces apparaissent dans l'ordre où elles sont dessinées, sans prise en compte de la position (suivant l'axe z perpendiculaire à la caméra) des faces des cubes.

Pour que les faces apparaissent correctement à l'observateur, il faut activer l'algorithme du *z-buffer*. Modifiez le programme ainsi :

– dans la fonction `main` remplacez l'instruction

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
```

par l'instruction

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

et ajouter l'instruction

```
glEnable(GL_DEPTH_TEST);
```

après les instructions de création de la fenêtre graphique (`glutCreateWindow`),

– dans la fonction `dessin` remplacez l'instruction

```
glClear(GL_COLOR_BUFFER_BIT);
```

par l'instruction

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

Recompilez le programme et testez la rotation : les cubes doivent être correctement représentés.
Cette méthode doit être utilisée pour la représentation de scènes en 3D.

En faisant tourner la scène, il peut apparaître des parasites lors de l'affichage de celle-ci. Cela est dû au fait que la fonction de dessin modifie directement la fenêtre graphique, d'où des parasites possibles notamment par l'instruction d'effacement de la fenêtre.

Pour remédier à cet inconvénient, il est possible de travailler en *double buffer graphique*, le premier sert à l'affichage et le second sert à la construction de la nouvelle image. Le principe est de créer la nouvelle image dans le second buffer puis de permuter le second buffer avec le premier lorsque l'image doit être affichée.

Modifiez le programme ainsi :

– dans la fonction `main` remplacez l'instruction

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
```

par l'instruction

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

– dans la fonction `dessin` remplacez l'instruction

```
glFlush();
```

par l'instruction

```
glutSwapBuffers();
```

Recompilez le programme et testez la rotation. Le cube est correctement représenté et sans parasites lors d'une animation.

Cette méthode de *double buffer graphique* doit être utilisée lorsque l'affichage est modifié par le programme, lors d'animation notamment.

2.4 - Le placement de la caméra et le choix de la projection

Dans cette partie, nous allons voir le placement de la caméra par rapport à une scène 3D, le choix de la perspective, et la limitation du volume représenté avec les deux plans de coupe.

Compilez et exécutez le programme `damier1.cpp` : ce programme représente un jeu de dames, avec un damier et des pièces disposées sur celui-ci.

Perspective orthographique

La scéne est représentée en projection perspective (utilisation de `glOrtho`).
La caméra est positionnée par l'instruction

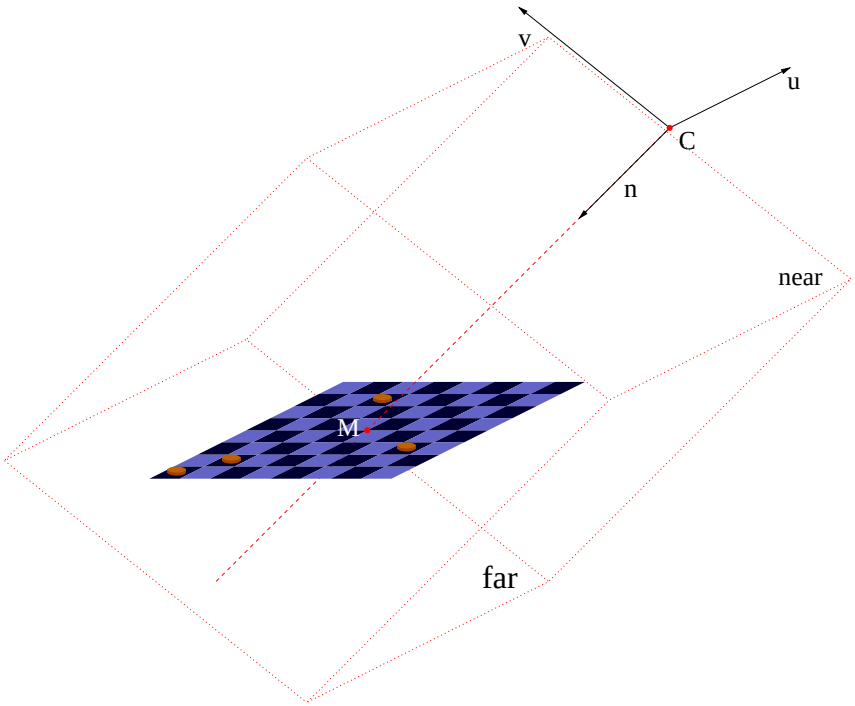
```
gluLookAt(Cx,Cy,Cz, Mx,My,Mz, vx,vy,vz);
```

avec $C = (\texttt{Cx}, \texttt{Cy}, \texttt{Cz})$ la position de la caméra,
$M = (\texttt{Mx}, \texttt{My}, \texttt{Mz})$ un point de visée définissant la normale du plan de projection $\overrightarrow{n} = \overrightarrow{CM}/\|\overrightarrow{CM}\|$ (dont l'origine est C)
$v = (\texttt{vx}, \texttt{vy}, \texttt{vz})$ le vecteur vertical de vue définissant l'axe vertical montant par rapport au plan de projection.

Le volume de coupe est défini par l'instruction

```
glOrtho(xmin, xmax, ymin, ymax, near, far);
```

avec $[\texttt{xmin}, \texttt{xmax}] \times [\texttt{ymin}, \texttt{ymax}]$ la fenêtre dans le plan de projection
et `near` et `far` ($0 < \texttt{near} < \texttt{far}$) les positions des deux plans de coupe par rapport à C.



Effectuez les modifications suivantes pour l'instruction `gluLookAt` :
— testez différentes positions pour le point C,
— revenir à la position initiale pour C, puis testez différentes positions pour le point M,
— revenir à la position initiale pour M, puis testez différentes directions pour le vecteur v,
— revenir à la direction initiale pour v,
puis effectuez les modifications suivantes pour l'instruction `glOrtho` :
— testez différentes fenêtres $[\texttt{xmin}, \texttt{xmax}] \times [\texttt{ymin}, \texttt{ymax}]$,
— revenir aux dimensions de la fenêtre initiale, puis testez différentes valeurs pour `near` et `far`.

Perspective perspective

Remplacez l'instruction :

```
glOrtho(-10.0*ratioFenetre, 10.0*ratioFenetre, -10.0, 10.0, 0.1, 10000.0);
```

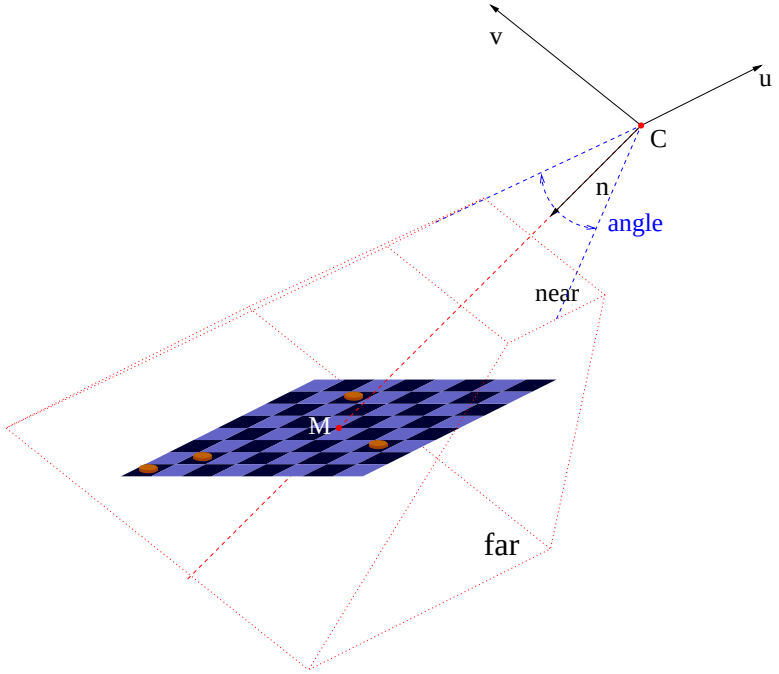
par l'instruction

```
gluPerspective(45.0, ratioFenetre, 0.1, 10000.0);
```

L'instruction

```
gluPerspective(angle, ratio, near, far)
```

permet de définir une représentation en perspective, le volume de coupe correspondant un tronc de cône à base rectangulaire
avec `angle` représentant l'angle de vue suivant la direction y (correspondant à la hauteur de l'image),
`ratio` est le rapport entre la largeur et la hauteur du rectangle de base du cône,
et `near` et `far` ($0 <$ `near` $<$ `far`) les positions des deux plans de coupe par rapport à C.



Effectuez les modifications suivantes pour l'instruction `gluLookAt` :
— testez différentes positions pour le point C,
— revenir à la position initiale pour C, puis testez différentes positions pour le point M,
— revenir à la position initiale pour M, puis testez différentes directions pour le vecteur v,
— revenir à la direction initiale pour v,
puis effectuez les modifications suivantes pour l'instruction `gluPerspective` :
— testez différentes valeurs pour `angle` (avec $0 <$ `angle` < 180),
— revenir à la valeur initiale pour `angle`, puis testez différentes valeurs pour `ratio`,
— revenir à la valeur initiale pour `ratio`, puis testez différentes valeurs pour `near` et `far`.

Exercice 3 :

Compilez et exécutez le programme `damier2.cpp` : ce programme est similaire au programme `damier1.cpp`, et vous allez le compléter afin d'implémenter différentes actions.

Dans ce programme, la position de la caméra est définie en coordonnées cylindriques :

$$\begin{cases} Cx &= distance\ \cos(azimut) \\ Cy &= distance\ \sin(azimut) \\ Cz &= hauteur \end{cases}$$

avec *distance* la distance de la caméra à l'axe Oz,
azimut la longitude (angle par rapport à l'axe Ox)
et *hauteur* la position de la caméra par rapport au plan $\{z = 0\}$.

A partir de la position (Cx, Cy, Cz), on peut construire un repère orthonormé (u, v, n) lié à la caméra de la manière suivante :
– le vecteur n de visée est choisie pour que la caméra C pointe toujours en direction du point-origine O

$$n = \frac{\overrightarrow{CO}}{\|\overrightarrow{CO}\|} = \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = \begin{pmatrix} -distance \times \cos(azimut)/R \\ -distance \times \sin(azimut)/R \\ -hauteur/R \end{pmatrix} \quad \text{avec } R = \sqrt{distance^2 + hauteur^2}$$

– on choisit ensuite un vecteur u horizontal et perpendiculaire à n :

$$u = \begin{pmatrix} u_x \\ u_y \\ u_z \end{pmatrix} = \begin{pmatrix} -\sin(azimut) \\ \cos(azimut) \\ 0 \end{pmatrix}$$

– puis on calcule le vecteur vertical de vue $v = n \wedge u$:

$$v = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} -hauteur \times \cos(azimut)/R \\ -hauteur \times \sin(azimut)/R \\ d/R \end{pmatrix}$$

Dans le programme `damier2.c`, la distance, l'azimut et la hauteur sont fixés, seul l'angle d'ouverture de la caméra peut être modifié à l'aide des touches clavier `+` et `-`.

1) Complétez le programme (procédure `clavier`) afin d'ajouter des actions clavier pour pouvoir modifier de manière interactive les valeurs de *distance*, *azimut* et *hauteur*, en faisant en sorte que la *distance* reste toujours supérieure ou égale à 1.
Choisir correctement les formules de modification de paramètres afin que cela soit ni trop lent ni trop rapide.

2) Pour une position C de la caméra, le vecteur vertical de vue v est fixé (suivant les trois paramètres *distance*, *azimut*, *hauteur*).

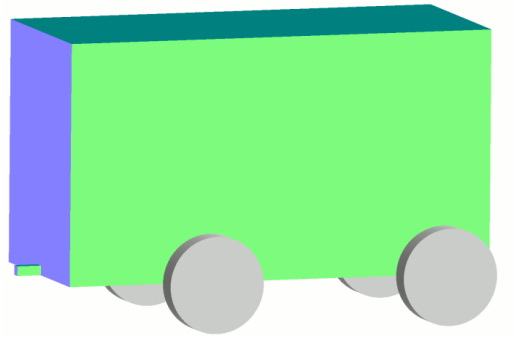
Complétez le programme pour pouvoir à une position fixée de la caméra, faire pivoter le vecteur de vertical de vue autour du vecteur de visée n (afin de pouvoir faire pivoter la caméra sur elle-même).
Pour cela, ajouter un paramètre angulaire β qui pourra varier par une action clavier, et remplacer le vecteur vertical de vue v par le vecteur $v_\beta = v\ \cos(\beta) + u\ \sin(\beta)$.

3 - Création d'une scène complexe

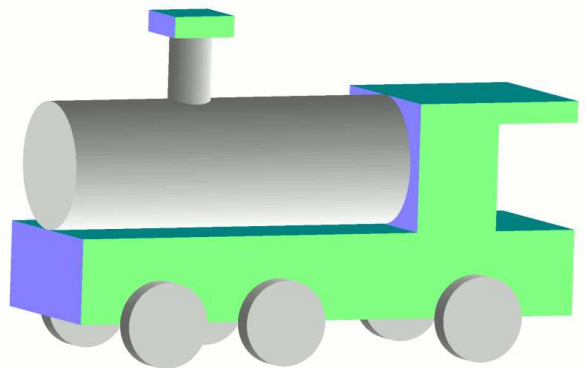
Exercice 4 :

Le programme `petit_train.cpp` correspond au programme `cubes3d.cpp` avec en plus la définition d'une autre primitive géométrique, un cylindre à base circulaire.

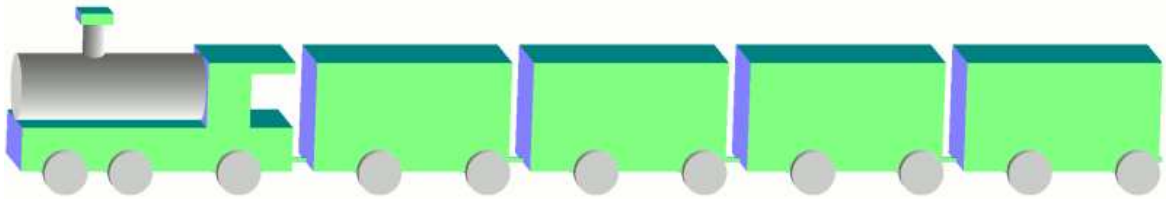
1. en utilisant les deux primitives géométriques `cube` et `cylindre`, construire un nouvel objet géométrique `wagon`.



2. en utilisant les deux primitives géométriques `cube` et `cylindre`, construire un nouvel objet géométrique `locomotive`.



3. en utilisant les deux objets `locomotive` et `wagon`, construire une scène représentant un train.



4 - Ombrage de faces

Compilez et exécutez le programme `ombrage_face.cpp` : ce programme calcule et affiche 2 sphères facettisées s'intersectant, en mode ombrage, chaque face est représentée avec une couleur unique.
Différentes actions sont implémentées à l'aide des touches clavier :

- niveau de facettisation : `1` (grossier), `2` (moyen),`3` (fin), et `4` (très fin),
- déplacement de la caméra en hauteur : `i` et `k`
- changement de la valeur de l'azimut de la caméra : `j` et `l`
- changement de l'angle d'ouverture de la caméra : `-` et `+`
- changement de la direction de la lumière : `g`, `h` (azimut) et `b`, `y` (élévation)
- affichage ou non des axes du repère *modelview* : `a`
- affichage ou non de la direction de la lumière : `z`
- sortie du programme : `q` ou `ESC`.

Testez le programme et ces différentes options.

4.1 - Ombrage de Gouraud

Dans ce programme, l'ombrage des faces est constant par face, c'est à dire que chaque face est dessinée avec une couleur unique dépendant de la normale N_F à la face F, cette normale N_F correspond au point central de la face F.

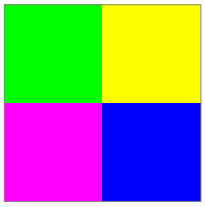
Exercice 5 :

Modifiez le programme afin d'utiliser la méthode de Gouraud (dégradé continue de couleur), c'est à dire pour chaque sommet S d'une face, calculer la normale puis la couleur correspondante.

5 - Texture

5.1 - Utilisation de texture

Compilez et exécutez le programme `texture1.cpp` : ce programme dessine un cube en plaquant sur chaque face l'image-texture :



vert	jaune
magenta	bleu

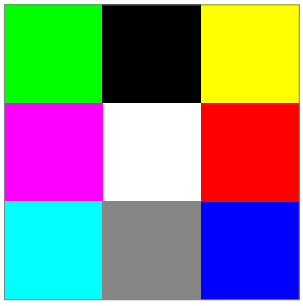
l'image-texture étant définie sous forme d'un tableau de `GLubyte` (octet), pour chaque pixel, les trois valeurs R,G,B (entre 0 et 255), ligne par ligne.
Dans le tableau, chaque ligne de l'image doit être codé avec un nombre d'octets multiple de 4. Pour cette image, la valeur de L étant 2, chaque ligne nécessite 6 octets, on complète alors avec deux 0 supplémentaires afin d'obtenir $8(= 2 \times 4)$ octets.

Cette texture est plaquée ensuite sur chacune des faces d'un cube.
Différentes actions sont implémentées à l'aide des touches clavier :

- tracé ou non des arêtes du cube : `a/A`
- rotation de la scène autour des axes : `x/X/y/Y/z/Z`
- zoom avant / arrière : `+/-`
- sortie du programme : `q` ou `ESC`.

Exercice 6 :

Modifiez le programme `texture1.cpp` afin de définir sous forme d'un tableau l'image-texture suivante :



vert	noir	jaune
magenta	blanc	rouge
cyan	gris	bleu

puis l'utiliser pour la plaquer sur chaque face du cube.

• L'image-texture peut aussi être chargée et créée à partir d'un fichier PPM binaire .
Par exemple, dans le fichier `texture1.cpp`, fonction `main`, remplacez la ligne

```
texture_cube = Texture(2,2,image_texture1);
```

par

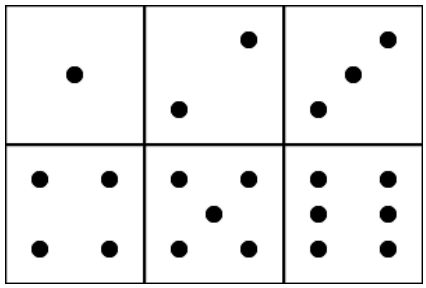
```
texture_cube = Texture("texture_tigres.ppm");
```

recompilez et exécutez le programme ainsi modifié.

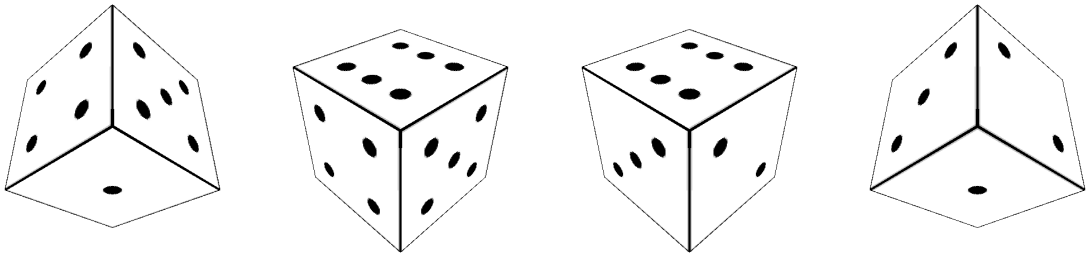
• Dans la procédure de dessin du cube, modifiez les coordonnées-texture associées aux quatre sommets de chaque face.

Exercice 7 :

1) Modifiez le programme `texture1.cpp` afin que le dessin du cube représente un dé à 6 faces. Pour cela utiliser l'image texture `texture_de.ppm` :



puis modifiez la procédure `cube` afin de plaquer sur chaque face une partie de cette image-texture afin d'obtenir le dé representé ci-dessous suivant quatre orientations différentes



2) Il est possible d'utiliser différentes images-textures dans un même programme.
Modifier le programme précédent afin d'obtenir le même objet dé mais en utilisant les 6 fichiers images suivants :

`texture_de_1.ppm`

`texture_de_2.ppm`

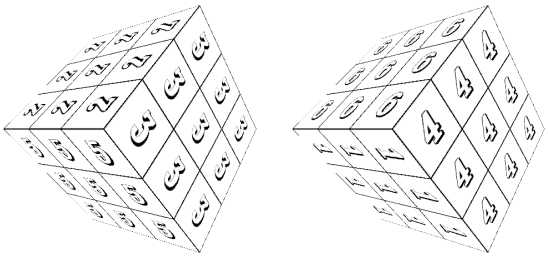
`texture_de_3.ppm`

`texture_de_4.ppm`

`texture_de_5.ppm`

`texture_de_6.ppm`

et de manière à ce que sur chaque face, la texture correspondante soit répétée $3 \times 3 = 9$ fois.



• On peut ensuite utiliser différentes textures avec différentes surfaces.

Compilez et exécutez le programme texture2.cpp : ce programme dessine un cylindre, la base inférieure en jaune, la base supérieure en magenta et la partie latérale en cyan.
Les commandes clavier sont les mêmes que le programme `texture1.cpp` (sauf pour le tracé des arêtes).

Exercice 8 :

Modifiez et complétez le programme afin de plaquer l'image-texture `BUS.ppm` sur les bases inférieure et supérieure, et l'image-texture `UGA.ppm` sur la partie latérale afin d'obtenir l'objet texturé ci-dessous.

Vue de dessous

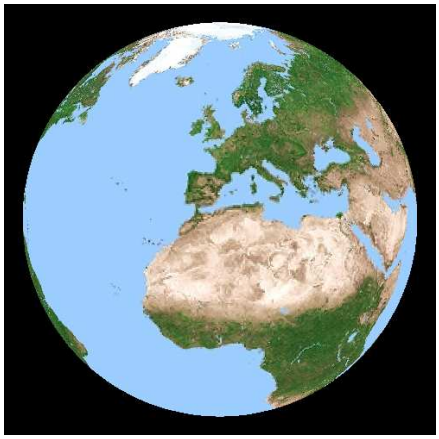
Vue de dessus

5.2 - Combinaison de texture et couleur

Texture(s) et couleur(s) peuvent être combinées pour simuler l'éclairage par une source lumineuse d'objets texturés.

Compilez et exécutez le programme texture_couleur.cpp : ce programme dessine une sphère texturée avec l'image-texture `UGA.ppm`.
Les commandes clavier sont les mêmes que le programme `texture2.cpp`.

Pour la texture, remplacez l'image `UGA.ppm` par l'image `texture_mappemonde.ppm` : vous devez obtenir l'image ci-dessous à gauche.



Couleur unie

Modèle d'illumination de Lambert

Exercice 9 :

Dans ce programme, l'ensemble de la sphère est représentée avec la couleur blanche.
En reprenant des parties du programme `ombrage_face.cpp`, modifiez/complétez le programme `texture_couleur.cpp` afin de pouvoir obtenir une représentation similaire à celle de la figure ci-dessus à droite.