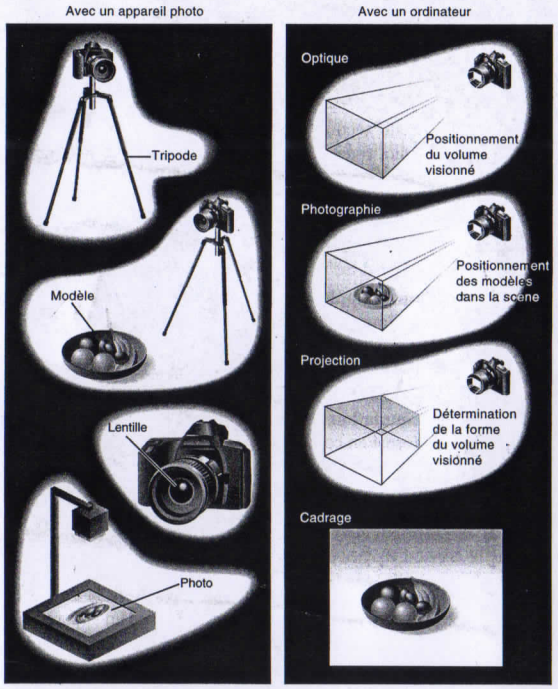
Analogie avec l'appareil photo

Le processus de transformation employé pour produire la scène souhaitée est comparable à l'action qui consiste à prendre une photographie avec un appareil photo. Comme le montre la Figure 3.1, les étapes à suivre avec un appareil ou avec un ordinateur peuvent se décomposer comme suit :

1. Positionnez votre tripode et tournez l'objectif vers la scène (transformation de visualisation).
2. Composez la scène à photographier de la manière souhaitée (transformation de modélisation).
3. Choisissez l'optique de l'appareil photo ou ajustez le zoom (transformation de projection).
4. Déterminez la dimension finale de la photographie, par exemple, agrandissez-la (transformation de cadrage).

Après avoir accompli ces étapes, vous pouvez appuyer sur le déclencheur ou dessiner la scène.

Figure 3.1
L'analogie avec l'appareil photo.

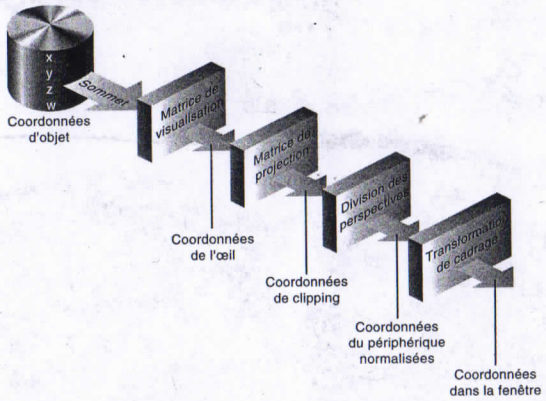


Remarquez que ces étapes correspondent à l'ordre dans lequel vous spécifiez les transformations souhaitées dans votre programme, ce qui n'est pas nécessairement l'ordre selon lequel les opérations mathématiques correspondantes seront effectuées sur un sommet donné. Les transformations de visualisation doivent précéder celles de la modélisation dans le code, mais vous pouvez spécifier les transformations de projection et de cadrage à n'importe quel moment précédant le dessin. La Figure 3.2 vous donne l'ordre dans lequel ces opérations sont réalisées sur votre ordinateur.

Pour spécifier la visualisation, la modélisation et les transformations de projection, construisez une matrice M, qui est alors multipliée par les coordonnées de chaque sommet v de la scène pour effectuer la transformation

$$v' = Mv$$

Figure 3.2
Etapes de la transformation des sommets.



Souvenez-vous que les sommets possèdent toujours quatre coordonnées (x, y, z, w), bien que w soit souvent égal à 1 et que pour des données bidimensionnelles, z soit égal à 0. Notez que les transformations de visualisation et de modélisation sont appliquées automatiquement aux vecteurs des surfaces normales en plus des sommets. On n'utilise les vecteurs normaux que pour les *coordonnées de l'œil*. La relation entre le vecteur normal et les données de sommets est ainsi préservée.

Les transformations de visualisation et de modélisation que vous spécifiez sont combinées afin de former la matrice de visualisation, qui s'applique aux *coordonnées d'objet* entrantes pour produire les coordonnées de l'œil. Ensuite, si vous avez spécifié des plans de clipping supplémentaires afin de supprimer certains objets de la scène ou de générer des vues de la scène en écorché, ils interviennent à ce moment-là.

Ensuite, OpenGL applique la matrice de projection pour produire les *coordonnées de l'œil*. Cette transformation définit un volume visionné et les objets situés à l'extérieur de ce volume sont découpés de manière à disparaître de la scène finale. A l'issue de cette étape, la *division des perspectives* s'accomplit par la division des valeurs des coordonnées par w, afin de produire des *coordonnées de périphérique normalisées*.

▶ *Voir Annexe F pour plus d'informations sur la signification de la coordonnée* w et la façon dont elle affecte les transformations matricielles.

Enfin, les coordonnées transformées sont converties en *coordonnées dans la fenêtre* par application de la transformation de cadrage. Vous pouvez manipuler les dimensions du cadrage pour agrandir l'image finale, l'étroitiser ou l'élargir.

(2)

Il ne serait pas faux de partir du principe que les coordonnées x et y suffisent pour déterminer les pixels à afficher à l'écran. Toutefois, toutes les transformations s'appliquent également à la coordonnée z. De cette manière, à la fin du processus de transformation, les valeurs z reflètent la profondeur d'un sommet donné, mesurée en distance par rapport à l'écran. L'une des utilisations de cette valeur de profondeur est l'élimination des tracés inutiles. Par exemple, supposons que deux sommets ont des coordonnées x et y identiques, mais des valeurs z différentes. OpenGL peut exploiter ces informations pour déterminer quelles sont les surfaces recouvertes par d'autres et peut ainsi éviter de tracer les surfaces cachées.

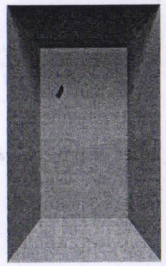
▶ *Voir Chapitres 5 et 10 pour plus d'informations sur la technique de suppression des surfaces cachées.*

Comme vous l'avez certainement deviné, vous devez maintenant découvrir quelques notions mathématiques relatives aux matrices de manière à tirer le meilleur parti de ce chapitre. C'est le moment de vous rafraîchir la mémoire en matière d'algèbre linéaire.

Exemple simple : dessiner un cube

L'Exemple 3.1 dessine un cube qui subit une transformation de modélisation (voir Figure 3.3). La transformation de visualisation, **gluLookAt()**, positionne l'objectif et le fait pointer vers le cube. Des transformations de projection et de cadrage interviennent également. Le reste de la section vous décrit en détail l'exemple et explique brièvement les commandes de transformation employées. Les prochaines sections contiennent une description complète et détaillée de toutes les commandes OpenGL de transformation.

Figure 3.3
Cube transformé.



Exemple 3.1 : Cube transformé : cube.c

```
#include <GL/glut.h>
#include <stdlib.h>

void init(void)
{
   glClearColor(0.0, 0.0, 0.0, 0.0);
   glShadeModel(GL_FLAT);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(1.0, 1.0, 1.0);
   glLoadIdentity();              /* vide la matrice */
         /* transformation de visualisation  */
   gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glScalef(1.0, 2.0, 1.0);       /* transformation de modélisation */
   glutWireCube(1.0);
   glFlush();
}

void reshape(int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glFrustum(-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
   glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize(500, 500);
   glutInitWindowPosition(100, 100);
   glutCreateWindow(argv[0]);
   init();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

Manipulation de la pile des matrices

Les matrices de modélisation-visualisation et de projection que vous avez créées, chargées et multipliées ne sont que les parties visibles de leurs icebergs respectifs. En réalité, chacune de ces matrices est le premier élément d'une pile de matrices (voir Figure 3.20).

Figure 3.20

Piles des matrices de modélisation-visualisation et de projection.

Une pile de matrices est un outil intéressant de construction de modèles hiérarchiques, au sein desquels des objets compliqués sont construits à partir d'objets plus simples. Par exemple, supposons que vous vouliez dessiner une automobile dotée de quatre roues, chacune étant reliée au châssis par cinq boulons. Une routine suffit pour les roues et une autre pour les boulons, car le rendu de ces deux objets est identique. Ces routines dessinent une roue ou un boulon dans une position et une orientation données, disons centrés à l'origine avec leur axe coïncidant avec l'axe des *z*. Lorsque vous dessinez la voiture avec les roues et les boulons, vous appelez la routine "dessiner roue" à quatre reprises avec des transformations différentes chaque fois afin de positionner les quatre roues de la manière appropriée. Pour chaque roue, le dessin du boulon revient cinq fois, chaque fois translaté de la manière souhaitée par rapport à sa roue.

Imaginons maintenant qu'il suffise de dessiner une carrosserie de voiture et quatre roues. Voici la syntaxe en langage simple de l'opération :

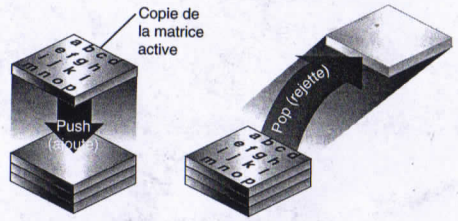
Dessine la carrosserie. Rappelle-toi ta position et déplace-toi vers la roue avant droite. Dessine la roue et rejette la dernière translation pour te replacer à l'origine de la carrosserie. Rappelle-toi ta position et translate-toi vers la roue avant gauche...

Ensuite, dessine la roue, rappelle-toi ta position et translate-toi successivement vers les positions des boulons, dessine-les, en rejetant les transformations antérieures après que chaque boulon a été dessiné.

Etant donné que les transformations sont stockées sous forme de matrices, la pile de matrices constitue un mécanisme idéal pour se charger de ces opérations successives : mémorisation, translation et rejet. Toutes les opérations matricielles que nous avons décrites jusqu'à présent (**glLoadMatrix()**, **glMultMatrix()** et **glLoadIdentity()**, ainsi que les commandes spécifiques) s'adressent à la matrice active, autrement dit celle qui se trouve en haut de la pile. Pour prendre le contrôle de l'ordre de la pile, utilisez la commande **glPushMatrix()**. Celle-ci copie la matrice active et ajoute la copie en haut de la pile. Quant à la commande **glPopMatrix()**, elle rejette la matrice située en haut de la pile, comme le montre la Figure 3.21. La matrice active est toujours la matrice du haut de la pile. En effet, **glPushMatrix()** signifie "rappelle-toi où tu es" et **glPopMatrix()** veut dire "retourne où tu étais".

Figure 3.21

Ajouter une matrice à la pile ou la rejeter.



void **glPushMatrix**(void);

Fait descendre d'un niveau toutes les matrices de la pile. La pile active est déterminée par **glMatrixMode()**. La matrice du haut de la pile est copiée de manière que son contenu soit dupliqué dans la première et la seconde matrice de la pile. Si trop de matrices sont ajoutées, cela génère une erreur.

void **glPopMatrix**(void);

Rejette la matrice de la pile, ce qui en détruit le contenu. La seconde matrice de la pile accède à la première position. La pile active est spécifiée par **glMatrixMode()**. Si la pile ne contient qu'une seule matrice, l'invocation de **glPopMatrix()** produit une erreur.

L'Exemple 3.4 dessine une automobile en supposant la présence des routines chargées de dessiner la carrosserie, une roue et un boulon.

Exemple 3.4 : Gérer la pile de matrices

```
dessine_roue_et_boulon()
{
   long i;

   dessine_roue();
   for(i=0;i<5;i++){
      glPushMatrix();
         glRotatef(72.0*i, 0.0, 0.0, 1.0);
         glTranslatef(3.0, 0.0, 0.0);
         dessine_boulon();
      glPopMatrix();
   }
}
dessine_carrosserie_et_roue_et_boulon()
{
   dessine_carrosserie();
   glPushMatrix();
    glTranslatef(40, 0, 30);  /*retourne à la position de la première roue*/
      dessine_roue_et_boulon();
   glPopMatrix();
   glPushMatrix();
    glTranslatef(40, 0, -30);   /* retourne à la position de la deuxième
    ➥roue */
      dessine_roue_et_boulon();
   glPopMatrix();
   ...                     /*continue avec les deux autres roues*/
}
```

Cet exemple de code a été mis au point en partant du principe que les axes des roues et des boulons coïncidaient avec l'axe des z ; autrement dit, les boulons sont espacés régulièrement, tous les 72 degrés, à 3 unités du centre de la roue et les roues avant sont situées à 40 unités vers l'avant et 30 unités vers la droite et la gauche par rapport à l'origine de la carrosserie.

Les piles sont plus efficaces que les matrices isolées et c'est encore plus vrai si elles sont implémentées dans le matériel. Lorsque vous ajoutez une matrice à la pile, vous n'avez pas besoin de recopier les données actives dans le processus principal et selon votre équipement, plusieurs éléments de la pile pourront être copiés simultanément. Pensez à conserver une matrice d'identité au bas de la pile de manière à ne pas avoir à rappeler systématiquement **glLoadIdentity()**.

Pile de la matrice de modélisation-visualisation

Comme vous avez pu le découvrir dans la section "Transformations de visualisation et de modélisation", la matrice correspondante contient le produit cumulé de la multiplication des matrices de visualisation et de modélisation. Chacune de ces transformations crée une nouvelle matrice qui multiplie la matrice active ; le résultat, qui devient la nouvelle matrice active, représente la transformation composite. La matrice de modélisation-visualisation contient au moins 32 matrices 4×4. Au départ, la matrice du haut est la matrice d'identité. Certaines implémentations OpenGL supportent plus de 32 matrices dans une pile. Pour trouver le nombre maximal de matrices autorisées, utilisez la commande de requête **glGetIntegerv**(GL_MAX_MODELVIEW_STACK_DEPTH, GLint *params).

Pile de la matrice de projection

La matrice de projection contient une matrice pour la transformation de projection, qui décrit le volume visionné. Généralement, on ne compose pas de matrices de projection, pour laquelle il faut émettre la commande **glLoadIdentity()** avant d'accomplir une transformation de projection. Pour cette raison également, la pile de la matrice de projection ne nécessite que deux niveaux. Certaines implémentations OpenGL supportent plus de 2 matrices 4×4. Pour déterminer la profondeur de la pile, invoquez **glGetIntegerv**(GL_MAX_PROJECTION_STACK_DEPTH, GLint *params).

Voici un cas dans lequel il vous faudrait une seconde matrice dans la pile : votre application a besoin d'afficher une fenêtre d'aide contenant du texte, en plus de la fenêtre destinée à afficher une scène en 3D. Le texte étant mieux rendu dans une projection en perspective cavalière, vous pouvez basculer temporairement d'une projection à l'autre, pour afficher l'aide, puis retourner à la projection précédente :

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();                  /*enregistre la projection active*/
   glLoadIdentity();
   glOrtho(...);                 /*se prépare à afficher l'aide*/
   afficher_aide();
glPopMatrix();
```

Remarquez que, par ailleurs, vous serez certainement obligé de modifier aussi la matrice de modélisation-visualisation.

Si vous êtes calé en mathématiques, vous pouvez créer des matrices de projection personnalisées qui réalisent des transformations projectives arbitraires. Par exemple, OpenGL et la bibliothèque d'outils n'incluent pas de mécanisme permettant les perspectives à deux points de fuite. Construisez une telle matrice et trouvez-lui des applications.

Exemples : Composition de plusieurs transformations :

Cette section illustre comment combiner plusieurs transformations afin d'obtenir un résultat particulier. Les deux exemples abordés sont celui du système solaire, dans lequel les objets doivent tourner sur leurs axes ainsi qu'en orbite les uns autour des autres, puis celui d'un bras robotisé, possédant plusieurs articulations qui transforment les systèmes de coordonnées tandis qu'elles se déplacent les unes par rapport aux autres.

Construire un système solaire

Le programme décrit dans cette section dessine un système solaire simple composé d'une planète et d'un soleil, exploitant chacun la même routine de dessin de sphère. Ce programme nécessite l'utilisation de **glRotate***() pour gérer la révolution de la planète autour du soleil et la rotation de la planète autour de son axe. Vous devez également faire appel à **glTranslate***() pour déplacer la planète vers son orbite en l'éloignant de l'origine du système solaire. Vous savez déjà que vous pouvez spécifier les dimensions souhaitées des deux sphères en apportant les arguments appropriés à la routine **glutWireSphere**().

Pour dessiner le système solaire, commencez par établir une transformation de projection et de visualisation. Dans cet exemple, on fait appel à **gluPerspective**() et à **gluLookAt**().

Il est facile de dessiner le soleil, car celui-ci se situe à l'origine du système de coordonnées global fixe, car c'est là que la routine de dessin de la sphère va le placer. Le dessin du soleil ne requiert donc pas de translation. Faites appel à **glRotate***() pour faire en sorte que le soleil tourne autour d'un axe arbitraire. Pour la planète qui tourne autour du soleil, comme dans la Figure 3.24, vous devez mettre en œuvre plusieurs transformations de modélisation. La planète doit effectuer une rotation autour de son axe une fois par jour. Et une fois par an, la planète aura effectué un tour complet autour du soleil.

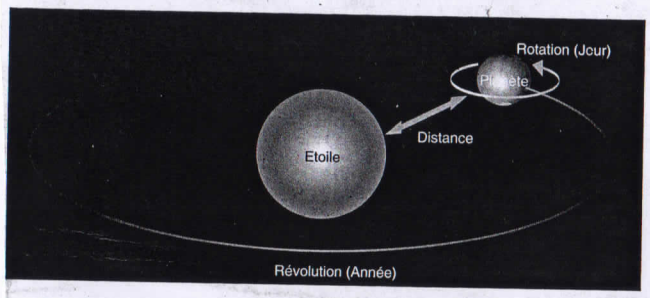


Figure 3.24
Système solaire à une planète.

Pour déterminer l'ordre des transformations de modélisation, visualisez ce qui se passe au niveau du système de coordonnées local. Une première commande **glRotate***() fait tourner le système de coordonnées local qui au départ, coïncide avec le système de coordonnées fixe. Ensuite, **glTranslate***() déplace le système de coordonnées local en une position située sur l'orbite de la planète. La distance de déplacement doit être égale au rayon de l'orbite. Par conséquent, la première commande **glRotate***() détermine en réalité où se situe l'orbite de la planète (autrement dit, à quel moment de l'année nous sommes).

Une seconde commande **glRotate***() applique une rotation au système de coordonnées local autour des axes locaux, déterminant ainsi l'heure du jour sur la planète. Une fois les commandes de transformation émises, vous pouvez dessiner la planète.

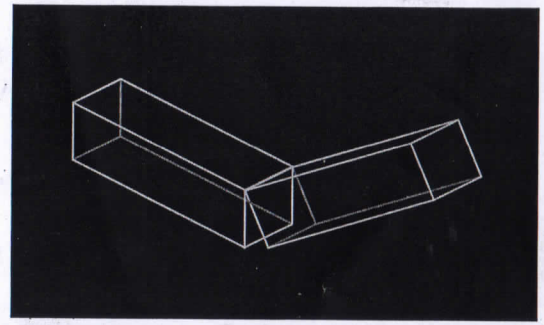
Essayez d'ajouter une lune en révolution autour de la planète, voire plusieurs lunes et des planètes supplémentaires. Indice : utilisez **glPushMatrix()** et **glPopMatrix()** pour enregistrer et restaurer la position et l'orientation du système de coordonnées aux instants souhaités. Pour créer plusieurs lunes autour de la planète, enregistrez le système de coordonnées avant de positionner chaque lune et restaurez le système de coordonnées entre chaque lune.

Essayez d'incliner l'axe de la planète.

Construire un bras articulé

Cette section aborde un programme de création d'un bras articulé de deux segments ou plus. Le bras doit être relié par des articulations à l'épaule, au coude ou autre. La Figure 3.25 illustre un tel bras à une articulation.

Figure 3.25
Bras articulé.



Vous pouvez partir d'un cube déformé pour chaque segment de bras, mais commencez par appeler les transformations de modélisation appropriées pour orienter chaque segment. Etant donné que le système de coordonnées local se trouve initialement au centre du cube, vous devez déplacer le système de coordonnées local vers une extrémité du cube. Dans le cas contraire, le cube tourne sur son centre et non pas par rapport à l'articulation.

Après avoir appelé **glTranslate***() pour établir le point d'articulation et **glRotate***() pour faire pivoter le cube, faites une translation pour retourner au centre du cube. Ensuite, le cube est déformé (aplati et élargi) avant d'être reformé. Les commandes **glPushMatrix**() et **glPopMatrix**() limitent l'effet de **glScale***(). Voici comment se présente le code pour ce premier segment de bras (le programme complet est repris dans l'Exemple 3.7) :

```
glTranslatef(-1.0, 0.0, 0.0);
glRotatef((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef(1.0, 0.0, 0.0);
glPushMatrix();
glScalef(2.0, 0.4, 1.0);
glutWireCube(1.0);
glPopMatrix();
```

Pour construire un second segment, vous devez déplacer le système de coordonnées local vers la prochaine articulation. Etant donné que celui-ci a été préalablement tourné, l'axe des *x* est déjà orienté parallèlement à la longueur du bras tourné. Par conséquent, la translation le long de l'axe des *x* déplace le système de coordonnées local vers la prochaine articulation.