DIBUJO DE ESCENAS 3D CON OpenGL

DIBUJO DE ESCENAS 3D CON OpenGL

7.1 INTRODUCCIÓN

En los temas anteriores, se han introducido los conceptos para el dibujo en 2D, así como el desarrollo de la clase simple *Canvas*. Esta clase proveía funciones que establecían la ventana y el puerto de vista y funciones para dibujar líneas como `moveTo()` y `lineTo()`. Además se introdujo el concepto de CT, así como de funciones para realizar transformaciones geométricas 2D (rotación, escala, traslación.) Éstas transformaciones son un caso particular de las transformaciones 3D en las que se ignoran la tercera dimensión.

En este capítulo veremos como se usan las transformaciones 3D en OpenGL. Haciendo énfasis en como se trasforman los objetos y posicionan para obtener la escena 3D deseada. Todo el trabajo se hace con matrices y OpenGL soporta las funciones necesarias para construir las matrices requeridas. Además la pila de matrices que mantiene OpenGL hace que sea fácil su uso para realizar transformaciones en los objetos y poder volver a las posiciones previas, para poder transformar otro objeto. Es muy fácil construir un programa en 3D con OpenGL para dibujar escenas utilizando transformaciones 3D. Experimentando con el programa se mejora notablemente la habilidad para la visualización de las sucesivas transformaciones 3D. OpenGL hace relativamente fácil la iniciación de la "cámara" para realizar una "foto" de una escena para un "punto de vista" específico. La cámara se crea con una matriz. En este capítulo nos vamos a centrar en como se establece la cámara y lo enfocaremos a la transformación de los objetos necesarias. Se pueden hacer escenas 3D impresionantes con muy pocas funciones OpenGL.

7.2 PROCESOS DE VISUALIZACIÓN Y TUBERÍAS GRÁFICAS

Todos los dibujos 2D realizados hasta ahora eran un caso especial de vistas 3D basadas en una simple "proyección paralela". Nosotros hemos usado la cámara mostrada en la figura 1. El "ojo" indica la posición del punto de vista de la escena y mira a través del eje *z* a la ventana que es rectangular y que se encuentra en el plano *xy*. El **volumen de la vista** de la cámara es un paralepípedo rectangular. Las dimensiones de la ventana delimitan el tamaño de éste junto con dos planos llamados cerca y lejos ("near" y "far"). Los puntos que se encuentran dentro del volumen delimitado se proyectarán en la ventana sobre líneas paralelas al eje *z*. Esto es equivalente a ignorar el componente *z*, así un punto 3D (x_1, y_1, z_1) se proyecta en (x_1, y_1, 0). Los puntos que se encuentran fuera son cortados. Otra transformación independiente de **puerto de vista** se encarga de proyectar el punto a la ventana en la posición indicada.

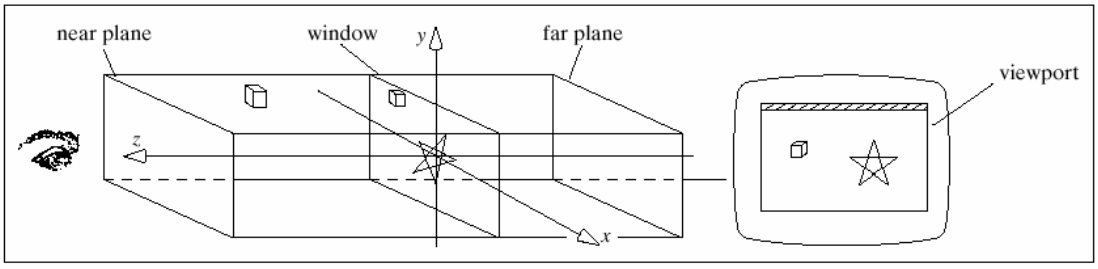

Figura 1. Vista usada en OpenGL para dibujos 2D

Ahora nos encontramos en gráficos 3D con objetos posicionados en la escena en 3D. Para nuestro ejemplo seguiremos usando la proyección paralela, usadas en ingeniería y arquitectura, para obtener una proyección más realista se utilizará la perspectiva cónica. Por lo tanto usaremos la misma cámara que la mostrada en la figura 1, pero con posicionamiento y orientación genérica con objeto de producir mejores vistas de la escena.

En la figura 2 se muestra la posición genérica de una cámara en una escena, todo lo que se encuentre dentro de la cámara será mostrado.

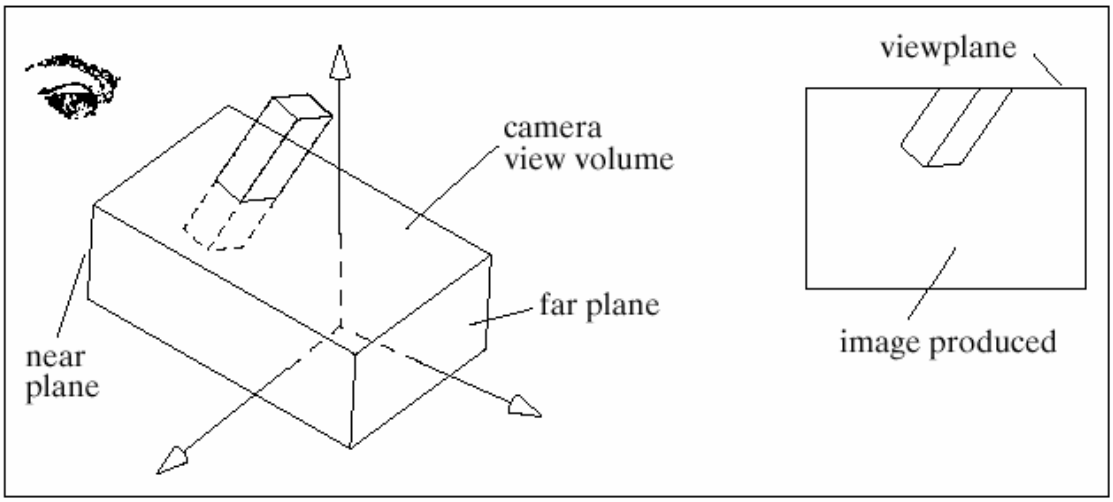

Figura 2. Posicionamiento genérico de una cámara que produce proyección paralela

Ya vimos anteriormente que OpenGL tiene tres funciones para realizar transformaciones, `glScaled(..)`, `glRotated(..)`, y `glTranslated(..)` para aplicar y poder darle la forma a los objetos. El prisma de la figura 2, es un cubo que ha sido rotado y alargado. OpenGL también tiene funciones que definen el volumen y posicionamiento de la escena.

El proceso gráfico para implementar que realiza OpenGL lo hace principalmente con transformaciones matriciales, nos vamos a centrar en lo que cada matriz de transformación realiza en la tubería. Ahora solamente nos vamos a centrar en comprender la idea básica de cómo cada matriz opera. En la figura 3 se muestra la tubería ligeramente simplificada.

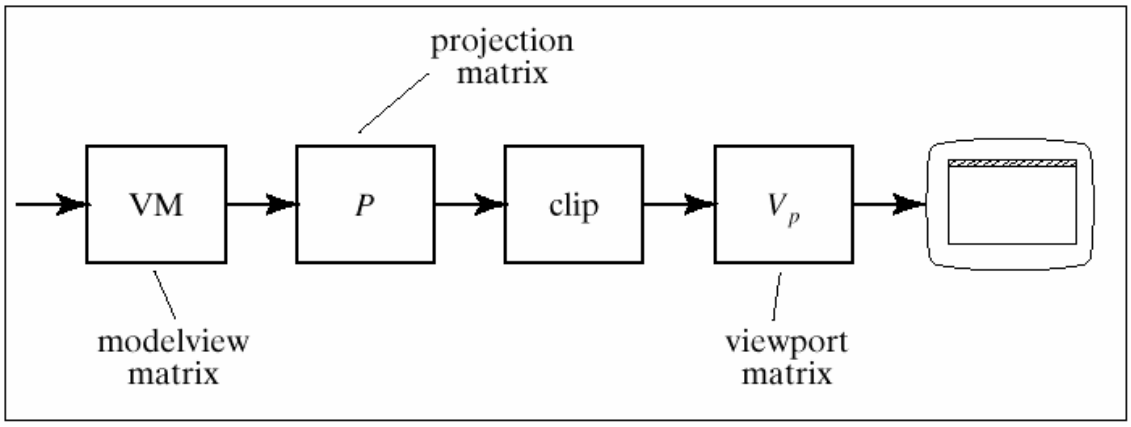

Figura 3. Tubería en OpenGL simplificada

Cada vértice de un objeto se pasa a través de esta tubería con una llamada tal como `glVertex3d(x, y, z)`. El vértice se multiplica por las distintas matrices, se recorta si es necesario, y si pasa el corte, se representa en el puerto de vista indicado. Cada vértice encuentra cada una de las tres matrices:

- La matriz del modelo, **modelview**.
- La matriz de proyección, **projection**.
- La matriz de puerto de vista, **viewport**.

La matriz modelview básicamente provee que es lo que estamos llamando al CT. Esta combina dos efectos: la secuencia de la trasformación del modelo aplicado a los objetos y la transformación de orientación y posición de la cámara en el espacio. Aunque la matriz modelview es una matriz simple en la tubería actual, es fácil pensar que es el producto de dos matrices, la matriz del modelo M y la matriz de vista V. Primero se aplica la matriz del modelo y después la matriz de vista.

La figura 4 nos muestra que es lo que hace la matriz M y la matriz V, de la situación introducida en la figura 2, en la que una cámara se posiciona en una escena consistente en un prisma.

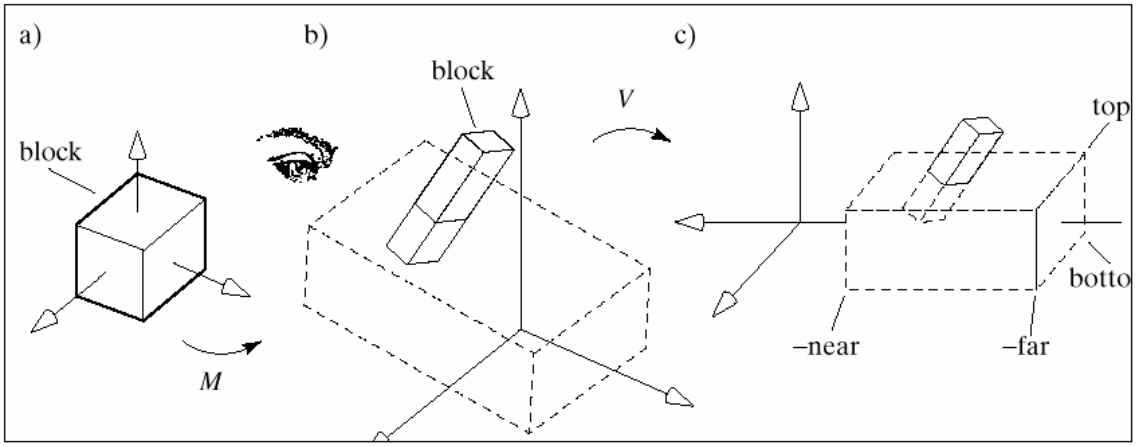

Figura 4. Efecto de la matriz modelview en la tubería gráfica. (a) antes de la transformación. (b) Después de la transformación del modelo. (c) Después de la transformación modelview.

La parte (a) muestra un cubo de unidad centrado en el origen. La transformación del modelo M se basa en escalado, rotación y traslación del cubo en un prisma como se muestra en la parte (b), en la que también se ve la posición relativa de la cámara y el volumen de ésta. La matriz V se usa ahora para rotar y trasladar el prisma a su nueva posición. La transformación específica en la que llevamos la cámara de su posición genérica de la escena a la posición del punto de vista situado en el origen y el volumen alineado con el eje z como se muestra en la parte (c). Los vértices del prisma se posicionan (esto es sus vértices tienen los valores apropiados), de tal forma que su proyección en un plano tenga los valores apropiados para mostrar la imagen final. Así que la matriz V realiza el cambio de coordenadas de los vértices de la escena en el sistema de coordenadas de la cámara (algunas veces se conocen como coordenadas de vista.) En el sistema de coordenadas de cámara, los bordes del volumen de vista son paralelos a los ejes *x*, *y* y *z*. El volumen de vista se extiende desde izquierda a derecha en *x*, de arriba a abajo en *y* y de cerca a lejos en *z*. La parte del objeto que queda fuera del volumen de vista no se muestra, como se ve en la figura 4.

La matriz projection escala y desplaza cada vértice de una forma particular, de tal forma que todos los vértices que caen dentro del volumen de vista caerán dentro de un *cubo estándar* que se extiende desde -1 a 1 en cada dimensión(conocido como coordenadas normalizadas.) La matriz de proyección ajusta el volumen de vista en un cubo centrado en el origen, que es particularmente eficiente para cortar los objetos. El escalado del prisma puede crear una distorsión, desde luego, pero esta distorsión será compensada por la transformación del puerto de vista. La matriz de proyección también invierte el sentido del eje z, así que incrementos del eje z ahora representan valores de profundidad desde el punto de vista de los ojos. En la figura 5 se muestra como un prisma se transforma en otro prisma diferente por esta transformación. (El volumen de vista de la cámara, nunca se representa como un objeto propiamente; este se define

solamente como la forma particular que la matriz de proyección tiene que convertir en el cubo estándar.)

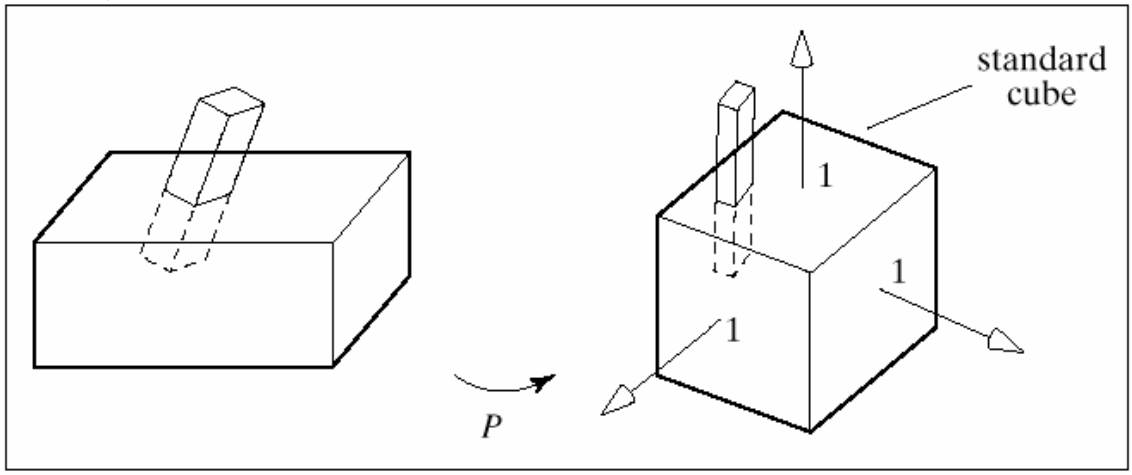

Figura 5. Efecto de la matriz "projection" para una proyección paralela

El proceso siguiente es realizar el recorte, el cual elimina la parte del prisma que queda fuera del cubo estándar. Y finalmente, la matriz viewport representa la parte interior del prisma en un puerto de vista 3D. Esta matriz representa el cubo estándar en un prisma de tal forma que *x* e *y* se extienden a través del puerto de vista en coordenadas de pantalla y la componente *z* se extiende de 0 a 1 y mantiene la medida de profundidad del punto (distancia entre el punto y el punto de vista de la cámara), como se muestra en la figura 6.

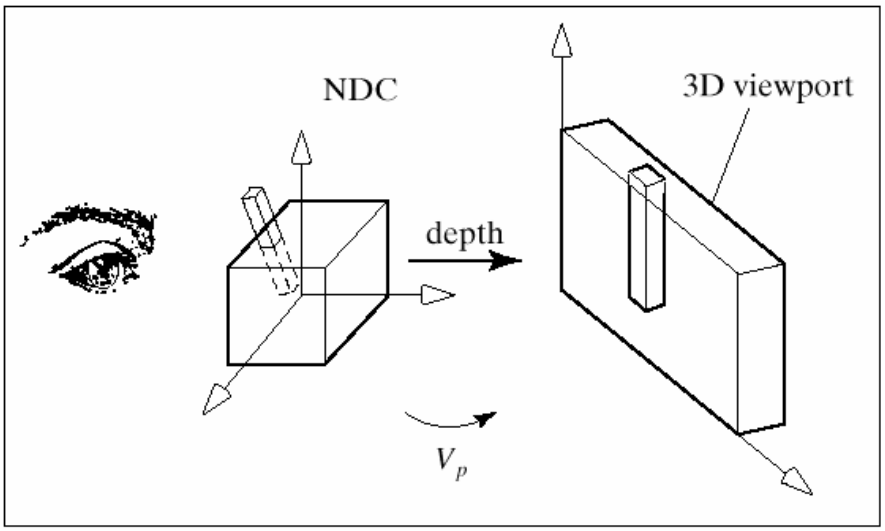

Figura 6. Efecto de la transformación "viewport"

El proceso explicado anteriormente explica como la tubería gráfica trabaja en OpenGL, mostrando como las transformaciones son un factor importante en el proceso. Cada vértice de un objeto está sujeto a una secuencia de transformaciones que pasan de coordenadas universal a coordenadas de vista, en un sistema neutral específicamente diseñado para recortar, y finalmente, en el sistema de coordenadas apropiado para representar en la pantalla. Cada transformación es afectada por una multiplicación de matrices. Nota que hemos omitido una serie de factores importantes en la tubería en esta primera revisión. Cuando se usen proyecciones en perspectiva, necesitamos incluir "la perspectiva" que capitaliza en unas apropiadas coordenadas homogéneas. Otros detalles importantes no se han incluido en el último paso como son el color del pixel entre vértices y eliminación de caras ocultas.

7.3 ALGUNAS HERRAMIENTAS OpenGL PARA MODELAR Y VISUALIZAR

Vamos a examinar ahora que funciones tiene OpenGL para modelar y situar la cámara. Tres funciones son las que se usan para realizar las transformaciones de modelado. Las siguientes son las que se usan normalmente para modificar la matriz modelview, que se hace actual ejecutando: `glMatrixMode(GL_MODELVIEW)`:

- `glScaled(sx, sy, sz)`; Pos-multiplica la matriz actual por una matriz que hace una escala de *sx* a *x*, *sy* a *y*, *sz* a *z*. Pone el resultado en la matriz actual.
- `glTranslated(dx, dy, dz)`; Pos-multiplica la matriz actual por una matriz que hace traslación de *dx* a *x*, *dy* a *y*, *dz* a *z*. Pone el resultado en la matriz actual.
- `glRotated(angle, ux, uy, uz)`; Pos-multiplica la matriz actual por una matriz que hace la rotación del ángulo `angle` sobre el eje que pasa por el origen y el punto `(ux, uy, uz)`. Pone el resultado en la matriz actual. Se usa la matriz de la ecuación (5) del tema 6.

7.4 ASIGNACIÓN DE LA CÁMARA OpenGL

La función `glOrtho(left, right, bottom, top, near, far);` estable que paralepípedo se asigna al volumen de vista, de `left` a `right` en *x*, de `bottom` a `top` en *y*, y de `-near` a `-far` en *z*. Todos los parámetros de `glOrtho` son del tipo `GLdouble`. Puesto que esta definición opera en coordenadas de vista, la cámara está situada en el origen, mirando a la parte negativa del eje z. La función crea una matriz y pos-multiplica la matriz actual por ésta.

Nota que el signo menos antes de `near` y `far` es porque la cámara por defecto está en el origen mirando hacia el valor negativo de z, usando un valor de 2 para `near` significa que situamos el plano cercano en z = -2. Similarmente usar 20 para `far` sitúa el plano 20 unidades enfrente de los ojos.

El siguiente código asigna la matriz de proyección:

```
glMatrixMode(GL_PROJECTION); //hace la matriz projection actual
glLoadIdentity(); //martiz identidad
glOrtho(left,right,bottom,top,near,far); //multiplica esto por la nueva matriz
```

7.5 POSICIONANDO Y DIRECCIONAMIENTO DE LA CÁMARA

OpenGL ofrece una función que hace muy fácil el posicionado de la cámara:

```
gluLookAt(eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y, up.z);
```

Esta función crea la matriz de vista y pos-multiplica la matriz actual por esta matriz. La función toma como parámetros la posición de los ojos (`eye`), de la cámara y el punto de mira (`look`) y también indica la dirección apropiada (`up`). Puesto que el programador conoce cual es la parte interesante de la escena a mostrar elige el valor apropiado de `eye` y `look`. Y `up` nos indica cual es el valor positivo de la y, normalmente se suele elegir (0, 1, 0) que indica que el valor positivo es paralelo al eje *y*. Ver figura 7.

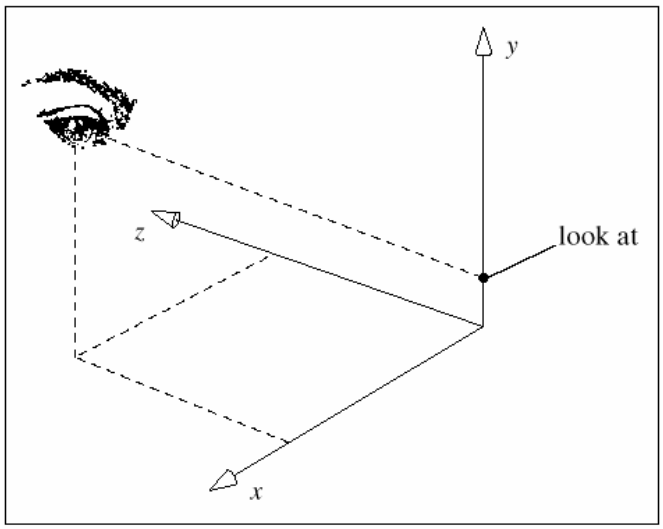

Figura 7.Definición de la cámara con `gluLookAt()`

Nosotros deseamos asignarle esta matriz V a la parte de la matriz VM. Por eso debe ser invocada antes de cualquier transformación del modelo, puesto que tal transformación será pos-multiplicada por la matriz modelview. Para usar la función `gluLookAt` empleamos la siguiente secuencia:

```
glMatrixMode(GL_PROJECTION); //hace la matriz projection actual
glLoadIdentity(); //martiz identidad
gluLookAt(eye.x,eye.y,eye.z,look.x,look.y,look.z,up.x,up.y,up.z);
```

7.5.1 EJEMPLO DE FUNCIONAMIENTO DE LA FUNCIÓN `gluLookAt()`

Sabemos que la función `gluLookAt()` construye la matriz que convierte las coordenadas universales en coordenadas de cámara ("eye".) En la figura 8 se muestra el posicionamiento del sistema de coordenadas de cámara respecto al universal, con su origen en el "ojo" y orientada de tal forma que sus tres vectores unitarios son **u**, **v** y **n**.

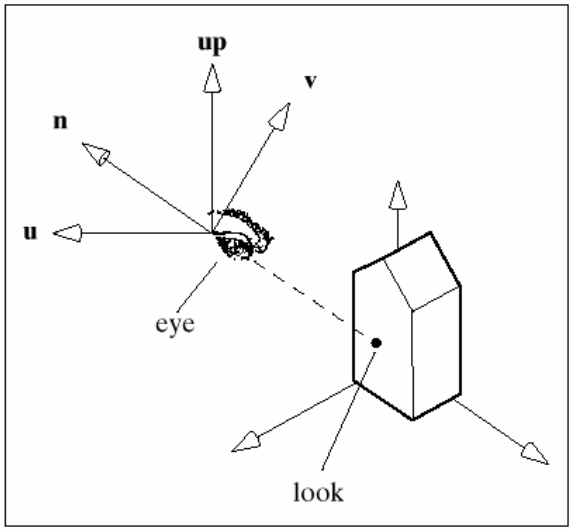

Figura 8. Conversión a coordenadas de cámara

El ojo está orientado en –**n**. `gluLookAt()` usa los parámetros *eye*, *look* y ***up*** para crear **u**, **v** y **n** de acuerdo a la siguiente relación:

n = *eye* – *look*
u = **up** x **n**
v = **n** x **u**

`gluLookAt` entonces normaliza los tres vectores de módulo unidad, construyendo la siguiente matriz

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

donde el punto d tiene como componentes (d_x, d_y, d_z) = (*-eye*.**u**, *-eye*.**v**, *-eye*.**n**)

a) **u**, **v** y **n** son perpendiculares mutuamente.
b) La matriz V convierte apropiadamente de coordenadas universal a "eye" en el origen (0, 0, 0), **u** a **i** = (1, 0, 0); **v** a **j** = (0, 1, 0) y **n** a **k** = (0, 0, 1).
c) Para el caso particular de *eye* = (4, 4, 4), *lookAt* = (0, 1, 0) y ***up*** = (0, 1,0) la matriz resultante es la siguiente

$$V = \begin{pmatrix} 0.70711 & 0 & -0.70711 & 0 \\ -0.3313 & 0.88345 & -0.3313 & -0.88345 \\ 0.6247 & 0.4685 & 0.6247 & -6.872 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

7.5.2 RECUPERANDO LOS VALORES DE UNA MARTIZ EN OpenGL

Es posible recuperar los valores de la matriz *modelview* para comprobar algunas de las trasformaciones realizadas. En OpenGL esto se hace definiendo una matriz `GLfloat mat[16]` y usando `glGetFloatv(GL_MODELVIEW_MATRIZ,mat)`que copia en la martiz `mat[16]` los 16 valores de la matriz modelview. `M[i][j]` se copia en los elementos de la matriz `mat[4j +i]`, para i, j = 0, 1, 2, 3

7.6 FORMAS PRIMITIVAS QUE TIENE OpenGL

Con objeto de poder experimentar con la cámara y puntos de vista, y antes de crear nuestros propios objetos, vamos a ver los que aporta por defecto OpenGL. GLUT tiene varios objetos definidos por defecto listos para visualizar, tales objetos incluyen esferas, toros, conos y sólidos platónicos (poliedros regulares), y la famosa *tasa de té.* Estas se pueden representar en forma alámbrica o modelo sólido con las caras sombreadas. La siguiente lista muestra las funciones usadas para dibujar estos objetos:

- **cube**: `glutWireCube(GLdouble size);` Cada lado tiene de longitud `size`.
- **sphere**: `glutWireSphere(GLdouble radius, GLint nSlices, GLint nStacks)`
- **torus**: `glutWireTorus(GLdouble inRad, GLdouble outRad, Glint nSlices, GLint nStacks)`
- **teapot:** `glutWireTeapot(Gldouble size)`

También existe `glutSolidCube(); glutSolidSphere()`, etc. La forma del toro se define por el radio interno y el externo. La esfera y el toro se aproximan a caras poligonales, y es posible definir los parámetros `nSlices` y `nStacks` para especificar cuantas caras se deben usar en la aproximación. `nSlices` es el número de subdivisiones alrededor del eje z, y `nStacks` es el número de "bandas" a lo largo del eje z, como si fueran disco.

Las funciones usadas para mostrar los cuatro sólidos platónicos (el quinto es el cubo ya mencionado) son las siguientes:

- **tetrahedron:** `glutWireTetrahedrom()`
- **octahedron:** `glutWireOctahedron ()`
- **dodecahedron:** `glutWireDodacahedron()`
- **icosahedron:** `glutWireIcosahedron()`

Todas los cuerpos geométricos se definen en el origen. También están disponibles los siguientes sólidos:

- **cono:** `glutWireCone(GLdouble baseRad, GLdouble height, GLint nSlice, GLint nStacks)`
- **tapered cylinder:** `gluCylinder(GLUquadricObj * qobj, GLdouble baseRad, GLdouble topRad, Gldouble height, GLint nSlice, GLint nStacks)`

El eje del cono y del cilindro cónico coincide con el eje *z*. Las bases están situadas en *z* = 0 y *z* = `height` a lo largo del eje *z*. El radio del cono y del cilindro para z = 0 se define por `baseRad`. El radio del cilindro cónico para *z* = `height` es `topRad`.

El **cilindro tapado** es una familia de formas, distinguidas por el valor de `topRad`. Cuando `topRad` es 1, no es cónico, y estamos en el caso del típico cilindro de base circular. Cuando `topRad` es 0, el cilindro es el cono.

Nota que el dibujo del cilindro en OpenGL requiere algún trabajo extra, porque es un caso especial de una superficie cuádrica. Para dibujar esto primero hay que definir un objeto nuevo cuadrático, segundo hay que asignarle un estilo de línea para dibujar (GLU_LINE, el modelo alámbrico, GLU_FILL, para el sólido), y tercero dibujar el objeto. El código se muestra a continuación.

```
GLUquadricObj * qobj = gluNewQuadric(); // crea un objeto cuadrático
gluQuadricDrawStyle(qobj, GLU_LINE); //estilo alámbrico
gluCylinder(qobj, baseRad, topRad, height, nSlices, nStacks); Dibuja el
cilindro
```

7.7 EJEMPLO DE UNA ESCENA COMPUESTA POR PRIMITIVAS OpenGL

En la figura 9 se muestra una escena con varios objetos situados en los vértices de un cubo de dimensión 1, cada objeto tiene su propio origen en el origen del sistema de coordenadas, todos están representados en modelo alámbrico.

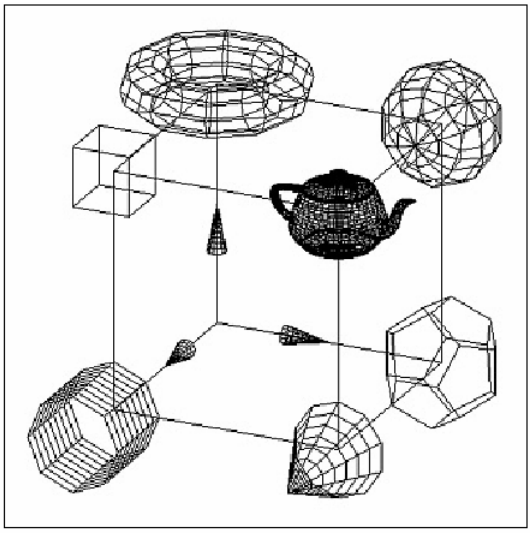

Figura 9. Primitivas dibujadas en estructura alámbrica

El volumen de la cámara se extiende desde –2 a 2 en *y*, con una relación de aspecto de 640/480. El plano cerca tiene el valor N = 0.1 y el lejos por F = 100. Esto se consigue usando

```
glOrtho(-2.0* aspect, 2.0* aspect, -2.0, 2.0, 0.1, 100);
```

La cámara se posiciona en eye = (2, 2, 2), lookAt = (0, 0, 0) y up = (0, 1,0) usando

```
gluLookat(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

A continuación se muestra el programa completo. En `main()` se inicializa para 640x480 pixel, el puerto de vista y el color de fondo se especifica en `displayWire()` cuando la función display se llame. En `displayWire()` la forma y disposición de la cámara se estable primero. Después cada objeto se dibuja por turnos. Cada objeto necesita su propia matiz de modelo con objeto de rotarla y posicionarla en el lugar correspondiente. Antes de cada transformación se establece un `glPushMatrix()` que se usa para recordar la posición actual, y después de que los objetos se han dibujado se vuelve a la actual transformación se recupera con `glPopMatrix()`. Así en el código cada objeto está comprendido entre el par de `glPushMatrix()`y `glPopMatrix()`.

También se ve en la figura que cada eje coordenado se muestra con un cono para orientar el dibujo. Para dibujar el eje *x*, el eje *z* se gira 90º sobre el eje *y*, al no poner este código entre `glPushMatrix()`y `glPopMatrix()` la próxima rotación produce el eje *y*.

Observa que como estamos usando proyección paralela, las cara paralelas se proyectan como paralelas.

```
//EJEMPLO DE UNA ESCENA COMPUESTA POR PRIMITIVAS OpenGL
#include <windows.h>  //suitable when using Windows 95/98/NT
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>
//<<<<<<<<<<<<<<<<<< axis >>>>>>>>>>>>>>
void axis(double length)
{ // draw a z-axis, with cone at end
       glPushMatrix();
       glBegin(GL_LINES);
          glVertex3d(0, 0, 0); glVertex3d(0,0,length); // along the z-axis
       glEnd();
       glTranslated(0, 0,length -0.2);
       glutWireCone(0.04, 0.2, 12, 9);
       glPopMatrix();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<< displayWire >>>>>>>>>>>>>>>>>>>>>>
void displayWire(void)
{
       glMatrixMode(GL_PROJECTION); // set the view volume shape
       glLoadIdentity();
       glOrtho(-2.0*64/48.0, 2.0*64/48.0, -2.0, 2.0, 0.1, 100);
       glMatrixMode(GL_MODELVIEW); // position and aim the camera
       glLoadIdentity();
       gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

       glClear(GL_COLOR_BUFFER_BIT); // clear the screen
        glColor3d(0,0,0); // draw black lines
        axis(0.5);                              // z-axis
       glPushMatrix();
       glRotated(90, 0,1.0, 0);
       axis(0.5);                             // y-axis
       glRotated(-90.0, 1, 0, 0);
       axis(0.5);                             // z-axis
       glPopMatrix();

       glPushMatrix();
       glTranslated(0.5, 0.5, 0.5); // big cube at (0.5, 0.5, 0.5)
       glutWireCube(1.0);
       glPopMatrix();

       glPushMatrix();
       glTranslated(1.0,1.0,0);  // sphere at (1,1,0)
       glutWireSphere(0.25, 10, 8);
       glPopMatrix();

       glPushMatrix();
       glTranslated(1.0,0,1.0);  // cone at (1,0,1)
       glutWireCone(0.2, 0.5, 10, 8);
       glPopMatrix();

       glPushMatrix();
       glTranslated(1,1,1);
       glutWireTeapot(0.2); // teapot at (1,1,1)
       glPopMatrix();

       glPushMatrix();
       glTranslated(0, 1.0 ,0); // torus at (0,1,0)
       glRotated(90.0, 1,0,0);
       glutWireTorus(0.1, 0.3, 10,10);
       glPopMatrix();

       glPushMatrix();
       glTranslated(1.0, 0 ,0); // dodecahedron at (1,0,0)
       glScaled(0.15, 0.15, 0.15);
       glutWireDodecahedron();
```

```
      glPopMatrix();
       glPushMatrix();
       glTranslated(0, 1.0 ,1.0); // small cube at (0,1,1)
       glutWireCube(0.25);
       glPopMatrix();

       glPushMatrix();
       glTranslated(0, 0 ,1.0); // cylinder at (0,0,1)
       GLUquadricObj * qobj;
       qobj = gluNewQuadric();
       gluQuadricDrawStyle(qobj,GLU_LINE);
       gluCylinder(qobj, 0.2, 0.2, 0.4, 8,8);
       glPopMatrix();
       glFlush();
}
//<<<<<<<<<<<<<<<<<<<< main >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char **argv)
{
       glutInit(&argc, argv);
       glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
       glutInitWindowSize(640,480);
       glutInitWindowPosition(100, 100);
       glutCreateWindow("Transformation testbed - wireframes");
       glutDisplayFunc(displayWire);
       glClearColor(1.0f, 1.0f, 1.0f,0.0f);  // background is white
       glViewport(0, 0, 640, 480);
       glutMainLoop();
}
```

7.8 EJEMPLO DE UNA ESCENA 3D CON RENDER Y SOMBREADO

En este ejemplo, se desarrolla una escena más compleja para ilustrar y completar el uso de las transformaciones del modelo. Veremos que es fácil hacer dibujos realistas de objetos sólidos incorporando sombra y eliminar las partes ocultas.

La cámara se incorpora en la posición indicada por `gluLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0, 1.0, 0.0);` Para mostrar la escena más cerca se modificará el valor de `double winHt = 1.0;`

La escena contiene tres objetos que están apoyados sobre una mesa en la esquina de una "habitación". Cada una de las tres paredes se crea aplastando un cubo en dirección del espesor y posicionandolo. La proyección no es muy realista puesto que se utiliza la proyección paralela, probar con gluPerspective(38,640/480,15,1) utilizando la siguiente secuencia:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(angulo,relacion_aspecto,cerca,lejos);
```

La figura está compuesta por tres brazos de esferas orientadas y posicionadas cada una en su sitio adecuado, más seis pequeñas esferas al final de estos brazos. La mesa consiste en un tablero y cuarto patas. Cada una de estas cinco piezas son cubos que se han escalado y posicionado correctamente. En la figura 10 se muestra el dimensionado paramétrico que caracteriza cada una de las partes. La función `tableLeg()` dibuja una pata y se llama cuatro veces en la función `table()` en diferentes posiciones. Siempre comprendidas las transformaciones entre `glPushMatrix()` y `PopMatrix()`. El código completo se muestra a continuación. Aquí se usa la versión sólida en vez de la estructura alámbrica, tal como `glutSolidSphere()`. En el código además se posiciona una luz y se le dan los parámetros necesarios para crearla, además de las propiedades del objeto como la luz que refleja.

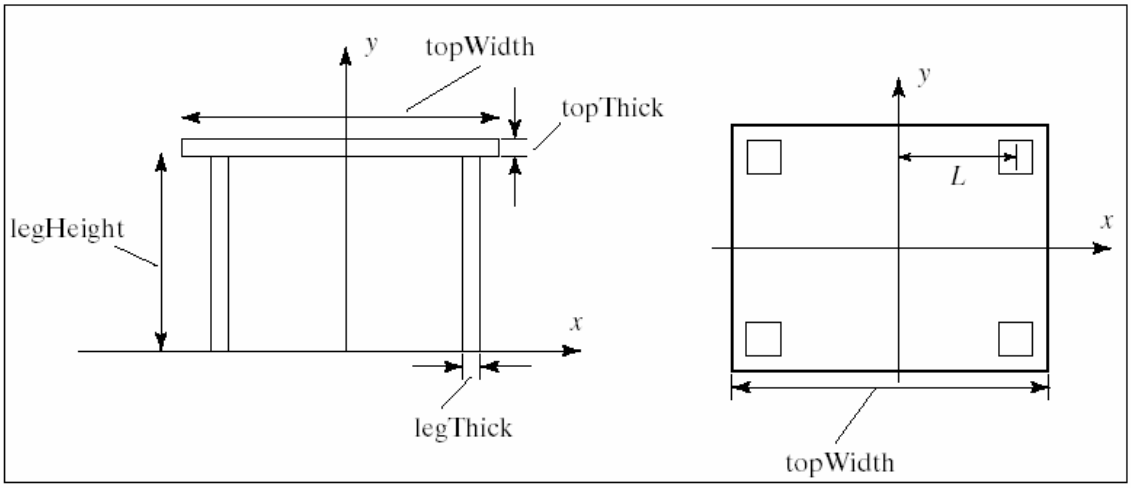

Figura 10. Diseño de la mesa

```
//PROGRAMA COMPLETO QUE DIBUJA UNA ESCENA REALISTA 3D
#include <windows.h>
#include <iostream.h>
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>
//<<<<<<<< PARED> >>>>>>>>
void wall(double thickness)
{
//Dibuja la pared con espesor y tapa = palno xz, esquina en el origen
	glPushMatrix();
	glTranslated(0.5,0.5 * thickness,0.5);
	glScaled(1.0, thickness, 1.0);
	glutSolidCube(1.0);
	glPopMatrix();
}
//blouqe superior de la mesa
void tableLeg(double thick, double len)
{
	glPushMatrix();
	glTranslated (0, len/2, 0);
	glScaled(thick, len, thick);
	glutSolidCube(1.0);
	glPopMatrix();
}
//<<<<<<<<<<Partes de la figura>>>>>>>>
void jackPart()
{
//dibuja un eje unidad de la figura - una esfera
	glPushMatrix();
	glScaled(0.2,0.2,1.0);
	glutSolidSphere(1,15,15);
	glPopMatrix();
	glPushMatrix();
	glTranslated(0,0,1.2); // bola al final
	glutSolidSphere(0.2,15 ,15);
	glTranslated(0,0, -2.4);
	glutSolidSphere(0.2,15 ,15); //bola al final del otro extremo
	glPopMatrix();
}
//<<<<<<<<figuras>>>>>>>>>>>>
void jack()
{
//dibuja una rama del esferoide
	glPushMatrix();
	jackPart();
	glRotated(90.0, 0, 1, 0);
	jackPart();
	glRotated(90.0, 1, 0, 0);
	jackPart();
	glPopMatrix();
}
//<<<<<<<<<<<<<mesa>>>>>>>>>>>>>>>
void table(double topWid, double topThick, double legThick, double legLen)
{
// dibuja la mesa la parte suporior y las cuarto patas
	glPushMatrix(); //dibuja el tope
	glTranslated(0, legLen,0);
	glScaled(topWid, topThick, topWid);
	glutSolidCube(1.0);
	glPopMatrix();
	double dist = 0.95 * topWid/2.0 - legThick / 2.0;
	glPushMatrix();
	glTranslated(dist, 0,  dist);
	tableLeg(legThick, legLen);
	glTranslated(0, 0, -2 * dist);
	tableLeg(legThick, legLen);
	glTranslated(-2 * dist, 0, 2 * dist);
	tableLeg(legThick, legLen);
	glTranslated(0, 0, -2 * dist);
	tableLeg(legThick, legLen);
	glPopMatrix();
}
```

```
//Dibuja los sólidos
void displaySolid(void)
{
//Asigna los apropiados materiales a las superficies
	GLfloat mat_ambient[]  = {0.7f, 0.7f, 0.7f, 1.0f}; //gris
	GLfloat mat_diffuse[]  = {0.6f, 0.6f, 0.6f, 1.0f};
	GLfloat mat_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
	GLfloat mat_shininess[]= {50.0f};
	glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
	glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
	glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
	glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
	// asigna la apropiada fuente de luz
	GLfloat lightIntensity[] = {0.7f, 0.7f, 0.7f, 1.0f};
	GLfloat light_position[] = {2.0f, 2.0f, 3.0f, 0.0f};
	glLightfv(GL_LIGHT0, GL_POSITION,light_position);
	glLightfv(GL_LIGHT0, GL_DIFFUSE,lightIntensity);
	//asigna la cámara
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	double winHt = 1.0;//altura mitad de la ventana. Probar con valores 0.5, 0.25, 0.125
	glOrtho(-winHt*64/48.0, winHt*64/48.0, -winHt, winHt, 0.1,100.0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();
//gluPerspective(38,640/480,15,1);
	gluLookAt(2.3, 2.3, 2, 0, 0.25, 0, 0.0, 1.0, 0.0);
	//comienza el dibujo
	glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT); // Limpia la pantalla
	glPushMatrix();
	glTranslated(0.4, 0.4, 0.6);
	glRotated(45, 0, 0, 1);
	glScaled(0.08, 0.08, 0.08);
	jack(); //Dibuja el objeto
	glPopMatrix();
	glPushMatrix();
	glTranslated(0.6, 0.38, 0.5);
	glRotated(30,0,1,0);
	glutSolidTeapot(0.08); //Dibuja la tetera
	glPopMatrix();
	glPushMatrix();
	glTranslated(0.25, 0.42, 0.35);
	glutSolidSphere(0.1, 15, 15);
	glPopMatrix();
	glPushMatrix();
	glTranslated(0.4, 0, 0.4);
	table(0.6,0.02,0.02,0.3); //dibuja la mesa
	glPopMatrix();
	wall(0.02); //pared en el plano xz
	glPushMatrix();
	glRotated(90,0.0,0.0,1.0);
	wall(0.02); //pared en el plano yz
	glPopMatrix();
	glPushMatrix();
	glRotated(-90.0, 1.0, 0.0,0.0);
	wall(0.02); //pared en el plano xy
	glPopMatrix();
	glFlush();

}
void main(int argc, char **argv)
{
	glutInit(&argc, argv);
//	glutInitDisplayMode(GLUT_SINGLE| GLUT_RGB| GLUT_DEPTH);
		glutInitDisplayMode(GLUT_SINGLE| GLUT_RGB| GLUT_DEPTH);
	glutInitWindowSize(640,480);
	glutInitWindowPosition(100,100);
	glutCreateWindow("Ejemplo de escena 3d");
	glutDisplayFunc(displaySolid);
	glEnable(GL_LIGHTING);
	glEnable(GL_LIGHT0);
	glShadeModel(GL_SMOOTH);
	glEnable(GL_DEPTH_TEST); //para eliminar las caras ocultas
	glEnable(GL_NORMALIZE); //normaliza el vector para ombrear apropiadamente
	glClearColor(0.1f, 0.1f, 0.1f, 0.f); //El color de fondo es gris
	glViewport(0, 0, 640, 480);
	glutMainLoop();
}
```