4. VIENDO LA ESCENA DESDE OTRO PUNTO DE VISTA: CÁMARAS

En este capítulo se profundiza en los conceptos de transformaciones de vista y de proyección y se explica con más detalle las funciones de que dispone OpenGL para manejar dichas transformaciones.

Para poder ver la escena desde otros puntos de vista es necesario mover la cámara, esto es, su ***punto de vista*** y/o su ***punto de atención***. Esto puede hacerse de forma directa dándole las coordenadas del nuevo punto, sin embargo es más útil dotar a la cámara de diferentes tipos de movimientos controlados mediante el ratón o el teclado, de forma que se pueda mover por la escena en la forma deseada. Se implementan en este capítulo, los dos tipos de movimiento de cámara más clásicos: el modo Examinar y el modo Caminar.

4.1. TRANSFORMACIONES DE VISTA

Una transformación de vista cambia la posición y orientación del punto de vista. Hay que recordar que para conseguir una cierta composición de una escena, se puede hacer bien moviendo la cámara o bien moviendo los objetos en dirección contraria. Por otro lado, hay que tener presente que los comandos para transformaciones de vista deben ser ejecutados antes de realizar cualquier transformación de modelado, para que las transformaciones de modelado tengan efecto sobre los objetos.

Por lo tanto, se puede realizar una transformación de vista de cualquiera de las siguientes maneras:

- Usando uno o más comandos de transformaciones de modelado (esto es, **glTranslate*()** y **glRotate*()**).
- Usando la rutina **gluLookAt()** de la Librería de Utilidades para definir una línea de visión. Esta rutina encapsula una serie de comandos de translación y rotación.
- Crear tu propia rutina de utilidades que encapsule rotaciones y translaciones.

A menudo los programadores construyen la escena alrededor del origen de coordenadas u otra localización conveniente y entonces deciden observarla desde un punto arbitrario para obtener una buena vista de ella. Como su propio nombre indica, la función **gluLookAt()** está diseñada con ese propósito. Concretamente, la declaración de la función es de la siguiente forma:

```
void gluLookAt( GLdouble eyex, GLdouble eyey, GLdouble eyez,
                GLdouble centerx, GLdouble centery, GLdouble centerz,
                GLdouble upx, GLdouble upy, GLdouble upz );
```

El punto de vista deseado está especificado por *eyex*, *eyey*, y *eyez*. Los argumentos *centerx*, *centery*, y *centerz* especifican cualquier punto a lo largo de la línea de visión, pero típicamente se trata de algún punto en el centro de la escena a la cual se está mirando. Los argumentos *upx*, *upy*, y *upz* indican qué dirección es hacia arriba (esto es, la dirección desde abajo hacia arriba del volumen de visión).

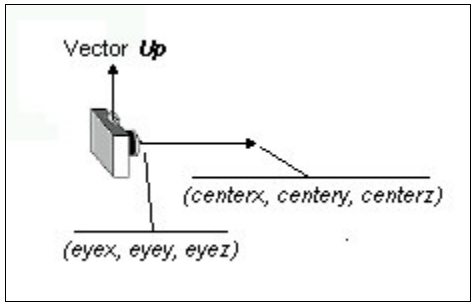


Figura 3.1. Parámetros de la función **gluLookAt()**.

Nótese que **gluLookAt()** es parte de la "Utility Library" y no de la librería básica de OpenGL. Esto no es porque no sea útil sino porque encapsula varios comandos básicos de OpenGL (específicamente, **glTranslate*()** y **glRotate*()**)

4.2. TRANSFORMACIONES DE PROYECCIÓN

El propósito de la transformación de proyección es definir un ***volumen de visión***, el cual se usa de dos formas. El volumen de visión determina cómo es proyectado un objeto en la pantalla (esto es, usando una proyección en perspectiva u ortogonal), y define qué objetos o porciones de objetos son colocados en la imagen final.

A continuación se expondrá cómo se define la matriz de proyección deseada, la cual es usada para transformar los vértices de la escena. Antes de ejecutar cualquier comando de transformación de proyección que se explican a continuación, hay que llamar a las siguientes funciones:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

para que los comandos afecten a la matriz de proyección y no a la de modelado y para que se eviten composiciones de transformaciones de proyecciones. Puesto que cada transformación de proyección describe completamente una transformación particular, normalmente no se quiere combinar una transformación de proyección con otra y es por eso que se carga la matriz identidad.

4.2.1. Proyección en perspectiva

La característica principal de la proyección en perspectiva es que cuanto más alejado está un objeto de la cámara, más pequeño aparece en la imagen final. Esto ocurre porque el volumen de visión para una proyección en perspectiva es un tronco de pirámide (una pirámide truncada a la cual se ha cortado el pico por un plano paralelo a la base). Los objetos que están dentro del volumen de visión son proyectados hacia la cúspide de la pirámide, donde estaría situada la cámara o punto de vista. Los objetos más cercanos al punto de vista aparecen más grandes porque ocupan proporcionalmente una cantidad mayor del volumen de visión que aquellos que están más alejados.

Para definir un volumen de visión de este tipo y por lo tanto crear una matriz de transformación de proyección en perspectiva se utiliza la rutina **gluPerspective()** de la "Utility Library" y que se especifica a continuación:

void **gluPerspective**(GLdouble *fovy*, GLdouble *aspect*, GLdouble *zNear*, GLdouble *zFar*);

La figura 3.2. muestra las características del volumen de visión especificado por la rutina **gluPerspective()** así como el significado de los parámetros que se deben pasar a esta.

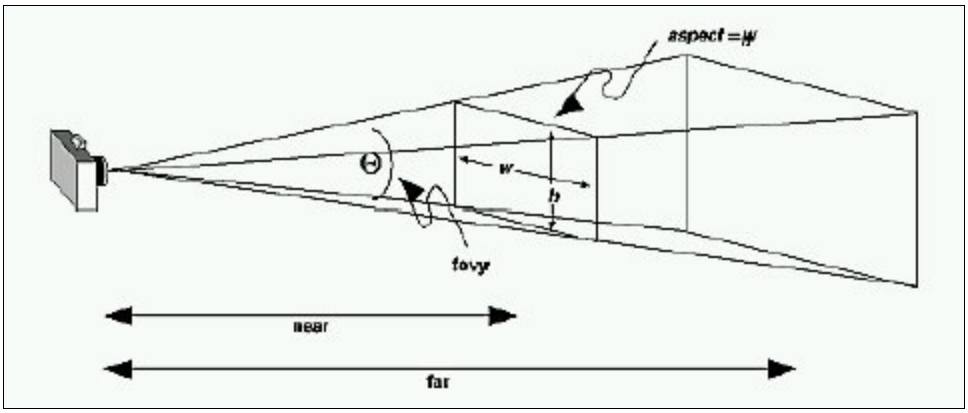


Figura 3.2. Volumen de visión en perspectiva especificado por **gluPerspective()**.

4.2.2. Proyección Ortogonal

Con una proyección ortogonal, el volumen de visión es un paralelepípedo, o más informalmente, una caja. A diferencia de la proyección en perspectiva, el tamaño del volumen de visión no cambia desde una cara a la otra, por lo que la distancia desde la cámara no afecta a lo grande que aparezca un objeto.

Para definir un volumen de visión de este tipo y por lo tanto crear una matriz de transformación de proyección ortogonal se utiliza la rutina **glOrtho()**, perteneciente a la librería básica de OpenGL, y que se especifica a continuación:

void **glOrtho**(GLdouble *left*, GLdouble *right*, GLdouble *bottom*, GLdouble *top*,
GLdouble *near*, GLdouble *far*);

La figura 3.3. muestra las características del volumen de visión especificado por la rutina **glOrtho()** así como el significado de los parámetros que se deben pasar a esta.

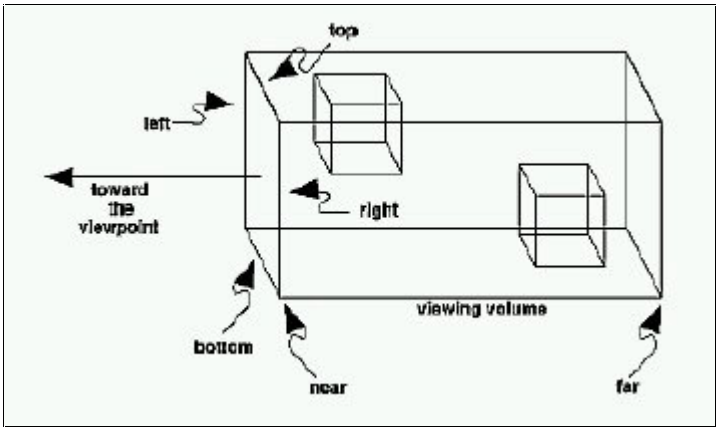


Figura 3.3. Volumen de visión Ortogonal especificado por **glOrtho()**.

4.3. DIFERENTES MODOS DE MOVER LA CÁMARA

Se va a dotar a la aplicación la posiblidad de realizar movimientos de cámara que permiten observar la escena desde diferentes puntos de vista. Se implementan los siguientes modos de movimiento de cámara:

1. **MODO EXAMINAR:** Consiste en mover la cámara por una esfera con centro en el Punto de Atención. Los movimientos de la cámara se realizarán mediante movimientos del ratón de la siguiente forma:
   - **Movimiento adelante-atrás**: El Punto de Vista de la cámara se mueve sobre un meridiano manteniendo el Punto de Atención fijo.
   - **Movimiento izquierda-derecha**: El Punto de Vista de la cámara se mueve sobre un paralelo manteniendo el Punto de Atención fijo.
   - **Movimiento de zoom pulsando el botón izquierdo del ratón**: Zoom - Alejar - Acercar o más exactamente de apertura o cierre del ángulo de la cámara.

2. **MODO CAMINAR:** Consiste en colocar la cámara a altura cero y avanzar en la dirección formada por el Punto de Vista y el Punto de Atención. Los movimientos de la cámara se realizarán mediante movimientos del ratón de la siguiente forma:
   - **Movimiento adelante-atrás pulsando el botón izquierdo del ratón**: El Punto de Vista y el de Atención de la cámara se mueven hacia delante o hacia atrás siguiendo el vector formado por ambos.
   - **Movimiento izquierda-derecha pulsando el botón izquierdo del ratón**: El Punto de Atención de la cámara gira hacia la izquierda o derecha.

Para ello se necesita definir un interface de cámara que guarde los datos de la cámara y una serie de funciones auxiliares para cada movimiento de la cámara. En el proyecto se incluyen los siguientes archivos:

`camera.h` contiene las declaraciones de las variables y de las funciones de manejo de cámara.
`camera.c` contiene las definiciones de las funciones para el manejo de la cámara.

A su vez, se necesitan ciertas funciones para el manejo de vectores, las cuales estan definidas en los siguientes ficheros, que también se incluyen en el proyecto:

`vector_tools.h` contiene las declaraciones de las funciones de manejo de vectores.
`vector_tools.c` contiene las definiciones de las funciones de manejo de vectores.

En la cabecera del fichero `tecnunLogo.c` se incluyen las sentencias:

```
#include "camera.h"
#include "vector_tools.h"
```

Se debes declarar una variable global que contendrá el interface de cámara, es decir, los datos de la cámara con los cuales se alimenta a las funciones que realizan las transformaciones de vista y de proyección. Puesto que los movimientos de la cámara se realizan con el ratón, se necesita conocer las coordenadas en las que estaba el puntero justo

antes de un movimiento para poder evaluar cuánto se ha movido. Se Incluyen las siguientes líneas en el fichero `tecnunLogo.c` inmediatamente después de los ficheros de cabecera:

```
static camera *LOCAL_MyCamera;
static int old_x, old_y;
```

Inicialmente se debe crear el interface para la cámara y asignárselo a la variable global que se acaba de declarar. Esto se realiza incluyendo la siguiente sentencia en la función main del programa:

```
LOCAL_MyCamera = CreatePositionCamera(0.0f, 1.0f, -3.0f);
```

Mediante esta sentencia se coloca la cámara en (0,1,-3) mirando hacia (0,0,0). Para poder manejar la cámara, bien en un modo o en otro, se utilizan las siguientes teclas:

F1	Desactivar cualquier modo de movimiento de cámara
F2	Modo Examinar
F3	Modo Caminar
HOME	La cámara vuelve a la situación inicial

Puesto que se trata de teclas especiales, se debe incluir en el main del programa la función callback **glutSpecialFunc(SpecialKey)**:

```
glutSpecialFunc(SpecialKey);
```

La función **SpecialKey()** que es pasada como parámetro será llamada cada vez que ocurra un evento de pulsado de una tecla especial entre las que se incluyen las teclas de funciones, las flechas de desplazamiento y las de Inicio, Fin, Avance página y Retroceso página. Se define a continuación la función **SpecialKey()**:

```
static void SpecialKey ( int key, int x, int y ){
  switch(key) {

    case GLUT_KEY_F1:
      glutPassiveMotionFunc(MouseMotion);
      LOCAL_MyCamera->camMovimiento = CAM_STOP;
      break;

    case GLUT_KEY_F2:
      glutPassiveMotionFunc(Examinar);
      LOCAL_MyCamera->camMovimiento = CAM_EXAMINAR;
      break;

    case GLUT_KEY_F3:
      glutPassiveMotionFunc(MouseMotion);
      LOCAL_MyCamera->camMovimiento = CAM_PASEAR;
      LOCAL_MyCamera->camAtY = 0;
      LOCAL_MyCamera->camViewY = 0;
      SetDependentParametersCamera( LOCAL_MyCamera );
      break;

    case GLUT_KEY_HOME: //Reset Camera
      LOCAL_MyCamera->camAtX =0;
      LOCAL_MyCamera->camAtY =0;
      LOCAL_MyCamera->camAtZ =0;
      LOCAL_MyCamera->camViewX = 0;
```

```
        LOCAL_MyCamera->camViewY = 1;
        LOCAL_MyCamera->camViewZ = -3;
        SetDependentParametersCamera( LOCAL_MyCamera );
        break;

      default:
        printf("key %d %c X %d Y %d\n", key, key, x, y );
    }
    glutPostRedisplay();
}
```

La función callback **glutPassiveMotionFunc()** recibe por ventana la función que se llamará cada vez que se produzca un evento de movimiento de ratón sin tener ningún botón de este pulsado. La función **MouseMotion(int x, int y)**, lo único que hace es guardar la posición a la que se mueve el ratón como la última posición de este:

```
void MouseMotion(int x, int y){
   old_y = y;
   old_x = x;
}
```

En la función main también hay que llamar a la función **glutPassiveMotionFunc()** para que cuando se mueva el ratón se guarde la posición a la que se traslade:

```
glutPassiveMotionFunc(MouseMotion);
```

La función **Examinar(int x, int y)** moverá la cámara en el modo Examinar según los movimientos del ratón:

```
void Examinar(int x, int y){
   float rot_x, rot_y;

   rot_y = (float)(old_y - y);
   rot_x = (float)(x - old_x);
   Rotar_Latitud( LOCAL_MyCamera, rot_y * DEGREE_TO_RAD );
   Rotar_Longitud( LOCAL_MyCamera, rot_x * DEGREE_TO_RAD );

   old_y = y;
   old_x = x;

   glutPostRedisplay();
}
```

Por otro lado, se debe indicar al programa qué función controlará los movimientos del ratón cuando se pulsa algún botón de este. Para ello se dispone de la función callback **glutMouseFunc()**, que se incluye en la función main del programa y que responde a los eventos de pulsado de botones del ratón:

```
glutMouseFunc(mouse);
```

La función **mouse()** que es pasada como parámetro será llamada cada vez que ocurra un evento de pulsado de un botón del ratón. En ella se define qué operaciones se realizan cuando se pulse un botón y se encuentra en un modo de movimiento de cámara o en otro:

```
void mouse(int button, int state, int x, int y){
   old_x = x;
   old_y = y;

   switch(button){
```

```
   case GLUT_LEFT_BUTTON:
     switch(LOCAL_MyCamera->camMovimiento){

     case CAM_EXAMINAR:
       if (state == GLUT_DOWN) glutMotionFunc(Zoom);
         if (state == GLUT_UP){
           glutPassiveMotionFunc(Examinar);
           glutMotionFunc(NULL);
         }
       break;

     case CAM_PASEAR:
       if (state == GLUT_DOWN) glutMotionFunc(Andar);
         if (state == GLUT_UP) glutMotionFunc(NULL);
       break;
     }
     break;

   case GLUT_RIGHT_BUTTON:
     if (state == GLUT_DOWN) ;
     break;

   default:
     break;

   }

   glutPostRedisplay();
}
```

Se observa que se realizan llamadas a la función callback **glutMotionFunc()** que responde a los movimientos del ratón cuando se tiene pulsado algún botón de este. Dependiendo del tipo de movimiento de cámara que se esté llevando a cabo, se le pasa por ventana una función que realice la operación definida:

```
void Zoom(int x, int y){
   float zoom;

   zoom = (float) ((y - old_y) * DEGREE_TO_RAD);
   old_y = y;

   switch(LOCAL_MyCamera->camMovimiento){

   case CAM_EXAMINAR:
     if (LOCAL_MyCamera->camAperture + zoom > (5 * DEGREE_TO_RAD) &&
         LOCAL_MyCamera->camAperture + zoom < 175 * DEGREE_TO_RAD)
       LOCAL_MyCamera->camAperture=LOCAL_MyCamera->camAperture + zoom;
     break;

   }

   glutPostRedisplay();
}


void Andar(int x, int y){

   float rotacion_x, avance_y;
```

```
  avance_y = (float)(y - old_y) / 10;
  rotacion_x = (float)(old_x - x) * DEGREE_TO_RAD / 5;
  YawCamera( LOCAL_MyCamera, rotacion_x );
  AvanceFreeCamera( LOCAL_MyCamera, avance_y);

  old_y = y;
  old_x = x;

  glutPostRedisplay();
}
```

Para que todos los movimientos de cámara que se realizan se vean reflejados se debe incluir en la función **display()** una llamada a la función que se encarga de actualizar los valores de la cámara a aquellos que tiene guardados el interface de cámara y que son los que se han modificado al realizar movimientos de cámara. Para ello se debe incluir en la función **display()** la siguiente sentencia:

```
SetGLCamera( LOCAL_MyCamera );
```

Por último, la función **reshape()** necesita un cambio para que al cambiar el aspecto de la ventana se mantengan las proporciones de la escena y esta no se deforme. Quedará de la siguiente forma:

```
void reshape(int width, int height) {
  glViewport(0, 0, width, height);

  SetGLAspectRatioCamera( LOCAL_MyCamera );
}
```

4.4. TRABAJOS PROPUESTOS

Dotar al programa de una tecla que permita cambiar el modo de proyección entre ORTOGONAL y PERSPECTIVA.

Programar otros modos de movimiento de cámara como son el MODO PAN o el MODO TRÍPODE.

camera.h

```
#ifndef CAMERA_H
#define CAMERA_H

#define CAM_PARALLEL 1
#define CAM_CONIC 2

#define CAM_STOP 0
#define CAM_EXAMINAR 1
#define CAM_PASEAR 2


typedef struct _Camera
{
   // we consider a rigth handed reference system for the camera
   // V point where the camera is placed (world coordinates)
   // A point the camera is looking at   (world coordinates)
   // up vector : unit vector, perpendicular to AV (world componnents)
   // origin camera reference system : at V
   // Z camera : defined by vector from A to V (penetrates the viewer's eye)
   // Y camera : defined by up vector
   // X camera : looking from V towards A goes rigthwards
   float  camViewX;  // View point
   float  camViewY;
   float  camViewZ;
   float  camAtX;    // look At point
   float  camAtY;
   float  camAtZ;
   float  camUpX;    // Up vector
   float  camUpY;
   float  camUpZ;
   float  camAperture;  // field of view radians      // NOTE : OpenGL uses degrees

   // defined as they are used by OpenGL
   // always => positive ; Far > Near (distance from plane to camera origin)
   float  camNear;
   float  camFar;
   int       camProjection;   // PARALLEL or CONIC
   int       camMovimiento; // EXAMINAR, ANDAR, TRIPODE or PAN

   // ****** dependent values ******

   // window system dependent
   float  aspectRatio;

   // for ortho projection
   float  x1, x2, y1, y2, z1, z2;

   // camera i j k vectors in world coordinates
   float  camIX, camIY, camIZ;
   float  camJX, camJY, camJZ;
   float  camKX, camKY, camKZ;
} camera;

void DestroyCamera ( camera **theCamera );
camera *CreatePositionCamera( float positionX, float positionY, float positionZ );
void  SetCamera( camera *thisCamera, float viewX, float viewY, float viewZ,
                        float atX,   float atY,   float atZ,
               float upX,   float upY,   float upZ );

void  SetDependentParametersCamera( camera *thisCamera );
void  SetGLCamera( camera *thisCamera );
void  SetGLAspectRatioCamera( camera *thisCamera );

// Free camera advances "step" following vector VA, step admits negative values
void  AvanceFreeCamera( camera *thisCamera, float step );

// ROTATION
void  YawCamera( camera *thisCamera, float angle );  // local axis Y camera

void Rotar_Latitud( camera *thisCamera, float inc );
void Rotar_Longitud( camera *thisCamera, float inc );


#endif /* CAMERA_H */
```

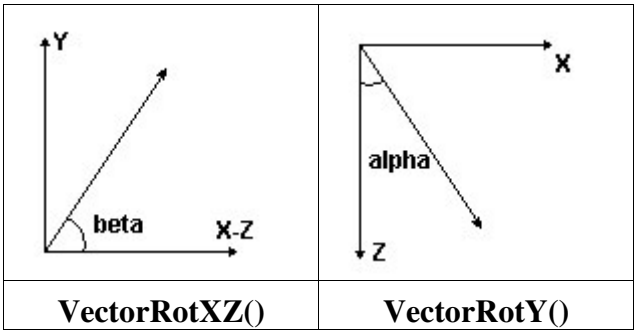
```
#include <GL/glut.h>
#include "camera.h"
#include "vector_tools.h"

void DestroyCamera ( camera **theCamera ){
   camera   *thisCamera = *theCamera;
   if( ! thisCamera ) return;

   free( thisCamera );

   *theCamera = NULL;
}

camera *CreatePositionCamera( float positionX, float positionY, float positionZ ){
   camera   *newCamera;
   int       ierr = 0;

   newCamera = (camera *) malloc( sizeof(camera) * 1 );

   newCamera->camViewX  = positionX;
   newCamera->camViewY  = positionY;
   newCamera->camViewZ  = positionZ;

   // looks towards
   newCamera->camAtX    = 0.0f;
   newCamera->camAtY    = 0.0f;
   newCamera->camAtZ    = 0.0f;

   newCamera->camUpX    = 0.0f;
   newCamera->camUpY    = 1.0f;
   newCamera->camUpZ    = 0.0f;

   newCamera->camAperture  = 60.0f * DEGREE_TO_RAD;
   newCamera->camNear      = 0.5f;
   newCamera->camFar       = 200.0f;

   newCamera->camProjection = CAM_CONIC;
   newCamera->aspectRatio   = 1.0f;

   SetDependentParametersCamera( newCamera );

   return newCamera;
}

void SetDependentParametersCamera (camera *thisCamera){
   // camera i j k vectors in world coordinates
   // camIX, camIY, camIZ;
   // camJX, camJY, camJZ;
   // camKX, camKY, camKZ;

   float  ix,    iy,    iz;
   float  jx,    jy,    jz;
   float  kx,    ky,    kz;
   float  atX,   atY,   atZ;
   float  upX,   upY,   upZ;
   float  viewX, viewY, viewZ;
   int       ierr = 0;

   viewX = thisCamera->camViewX;
   viewY = thisCamera->camViewY;
   viewZ = thisCamera->camViewZ;
   atX   = thisCamera->camAtX;
   atY   = thisCamera->camAtY;
   atZ   = thisCamera->camAtZ;
   upX   = 0.0f;
   upY   = 1.0f;
   upZ   = 0.0f;

   // i, j, k, up must be unit vectors
   // k = normalizar(  AV  )
   // i = normalizar( up ^ k )
   // j = k ^ i
   UnitVectorPP( &ierr, &kx, &ky, &kz, atX, atY, atZ, viewX, viewY, viewZ );
   UnitVectorVV( &ierr, &ix, &iy, &iz, upX, upY, upZ, kx, ky, kz );
   UnitVectorVV( &ierr, &jx, &jy, &jz, kx, ky, kz, ix, iy, iz );
```

```
   thisCamera->camKX = kx;
   thisCamera->camKY = ky;
   thisCamera->camKZ = kz;

   thisCamera->camIX = ix;
   thisCamera->camIY = iy;
   thisCamera->camIZ = iz;

   thisCamera->camJX = jx;
   thisCamera->camJY = jy;
   thisCamera->camJZ = jz;

   thisCamera->camUpX = jx;
   thisCamera->camUpY = jy;
   thisCamera->camUpZ = jz;
}

void  SetGLCamera( camera *thisCamera ){
   float  fovy;

   glMatrixMode( GL_PROJECTION );
      glLoadIdentity( );

   if( thisCamera->camProjection == CAM_CONIC ){
      fovy = thisCamera->camAperture*RAD_TO_DEGREE;
      gluPerspective(fovy, thisCamera->aspectRatio, thisCamera->camNear, thisCamera->camFar );
   }
   else { // CAM_PARALLEL
      glOrtho(thisCamera->x1, thisCamera->x2, thisCamera->y1, thisCamera->y2,
               thisCamera->z1,thisCamera->z2);
   }

   gluLookAt(thisCamera->camViewX, thisCamera->camViewY, thisCamera->camViewZ,
             thisCamera->camAtX, thisCamera->camAtY, thisCamera->camAtZ,
             thisCamera->camUpX, thisCamera->camUpY, thisCamera->camUpZ );
   glMatrixMode( GL_MODELVIEW ); //* GL_MODELVIEW *

}

void SetGLAspectRatioCamera( camera *thisCamera ){
   GLint     viewport[4];

   glGetIntegerv( GL_VIEWPORT, viewport );
   if( viewport[3] > 0 )
      thisCamera->aspectRatio = (float) viewport[2] / (float) viewport[3];// width/height
   else
      thisCamera->aspectRatio = 1.0f;

   SetDependentParametersCamera( thisCamera );
}

void  SetCamera( camera *thisCamera, float viewX, float viewY, float viewZ,
                                     float atX,   float atY,   float atZ,
                                     float upX,   float upY,   float upZ ){

   thisCamera->camViewX = viewX;
   thisCamera->camViewY = viewY;
   thisCamera->camViewZ = viewZ;
   thisCamera->camAtX = atX;
   thisCamera->camAtY = atY;
   thisCamera->camAtZ = atZ;
   thisCamera->camUpX = upX;
   thisCamera->camUpY = upY;
   thisCamera->camUpZ = upZ;

   SetDependentParametersCamera( thisCamera );
}

void AvanceFreeCamera(camera *thisCamera, float step) {
   float  vaX, vaY, vaZ;

   vaX= step * thisCamera->camKX;
   vaY= step * thisCamera->camKY;
   vaZ= step * thisCamera->camKZ;

   // Set V & A
```

```
   thisCamera->camViewX=thisCamera->camViewX+vaX;
   thisCamera->camViewY=thisCamera->camViewY+vaY;
   thisCamera->camViewZ=thisCamera->camViewZ+vaZ;
   thisCamera->camAtX = thisCamera->camAtX + vaX;
   thisCamera->camAtY = thisCamera->camAtY + vaY;
   thisCamera->camAtZ = thisCamera->camAtZ + vaZ;

   SetDependentParametersCamera( thisCamera );
}

void  YawCamera(camera *thisCamera, float angle){
   float vIn[3];

   vIn[0]=thisCamera->camAtX-thisCamera->camViewX;
   vIn[1]=thisCamera->camAtY-thisCamera->camViewY;
   vIn[2]=thisCamera->camAtZ-thisCamera->camViewZ;

   VectorRotY( vIn, angle );

   thisCamera->camAtX=thisCamera->camViewX+vIn[0];
   thisCamera->camAtY=thisCamera->camViewY+vIn[1];
   thisCamera->camAtZ=thisCamera->camViewZ+vIn[2];

   SetDependentParametersCamera( thisCamera );
}

void Rotar_Longitud(camera *thisCamera,float inc){

   float vIn[3];

   vIn[0]=thisCamera->camViewX-thisCamera->camAtX;
   vIn[1]=thisCamera->camViewY-thisCamera->camAtY;
   vIn[2]=thisCamera->camViewZ-thisCamera->camAtZ;

   VectorRotY( vIn, inc );

   thisCamera->camViewX=thisCamera->camAtX+vIn[0];
   thisCamera->camViewZ=thisCamera->camAtZ+vIn[2];

   SetDependentParametersCamera( thisCamera );
}

void Rotar_Latitud(camera *thisCamera,float inc){

   float vIn[3];

   vIn[0]=thisCamera->camViewX-thisCamera->camAtX;
   vIn[1]=thisCamera->camViewY-thisCamera->camAtY;
   vIn[2]=thisCamera->camViewZ-thisCamera->camAtZ;

   VectorRotXZ( vIn, inc, TRUE );

   thisCamera->camViewX=thisCamera->camAtX+vIn[0];
   thisCamera->camViewY=thisCamera->camAtY+vIn[1];
   thisCamera->camViewZ=thisCamera->camAtZ+vIn[2];

   SetDependentParametersCamera( thisCamera );
}
```

vector_tools.h

```
#ifndef TOOLS_H
#define TOOLS_H

#include <math.h>

#ifndef TRUE
#define TRUE   1
#define FALSE  0
#endif

// *** Mathematics
#define VECTOR_EPSILON    0.00001f
#define DISTANCE_EPSILON  1e-08f
#define ANGLE_EPSILON     0.00872665f  // 0.5 degrees

#define MOD(A,B,C) (float) sqrt( A*A + B*B + C*C )

#define PI_VALUE           3.14159265359f
#define DEGREE_TO_RAD      0.0174533f  /* 2.0 * 3.1415927 / 360.0 */
#define RAD_TO_DEGREE      57.2958f    /* 360 / ( 2.0 * 3.1415927 ) */


void VectorNormalize( int *ierr, float *vX, float *vY, float *vz );
void UnitVectorPP( int *ierr, float *wX, float *wY, float *wZ,
                      float aX,  float aY,  float aZ,
                      float bX,  float bY,  float bz );
void UnitVectorVV( int *ierr, float *wX, float *wY, float *wZ,
                      float uX,  float uY,  float uZ,
                      float vX,  float vY,  float vz );
void VectorRotY( float *v, float inc );
                            // rotates vector : around Y axis like moving its end on its parallel
void VectorRotXZ( float *vIn, float inc, int flagStop );
                            // rotates vector : like moving its end on its meridiam

#endif /* TOOLS_H */
```



VectorRotXZ()	VectorRotY()