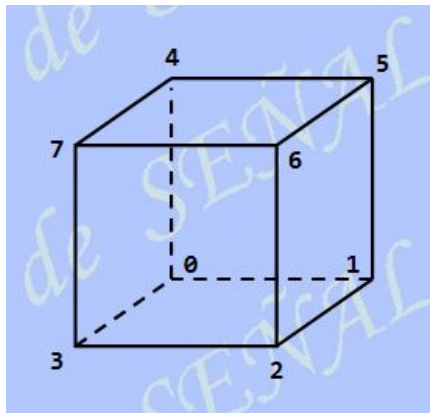
| | |
|---|---|
| Apellidos: García Ardiles | Nombre: Gabriel |
| Apellidos: Vicente de las Heras | Nombre: Sergio |


**Objetivos de este laboratorio:**

- Crear objetos 3D sencillos a partir de sus vértices, usando la descripción indexada de vértices en OpenGL.
- Aprender a usar descarte de polígonos ("culling") y z-buffer en OpenGL y entender las diferencias entre ambos.
- Entender la importancia del orden en el listado de vértices para definir la cara de delante y detrás de un polígono.
- Crear una interfaz para mover posición de la cámara interactivamente.

Partiremos del programa suministrado "GpO_02.cpp". Como siempre, para ir haciendo modificaciones, copiarlo con otro nombre (por ejemplo "ejer1.cpp") y cambiar el fichero fuente del proyecto ejemplo.sln al nuevo fichero.

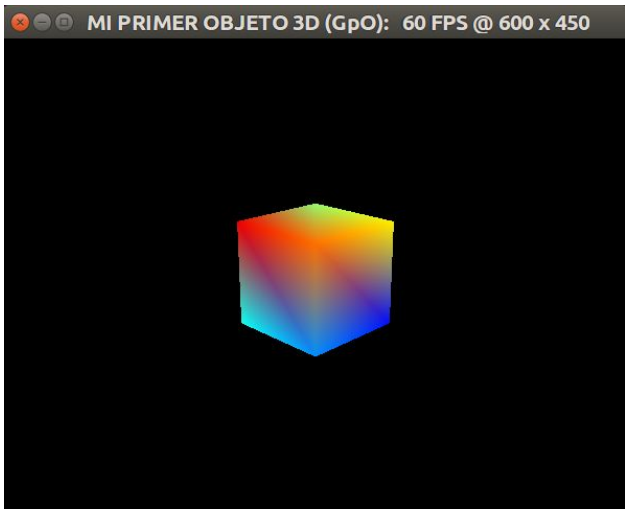
**Ejer 1:** El programa suministrado especifica en la función crear_cubo() los ocho vértices (posiciones y colores) de un cubo, junto con un listado de 36 (12x3) índices. Cada una de las 12 tripletas define un triángulo del modelo, ya que se necesitan 12 triángulos para especificar las 6 caras de un cubo.



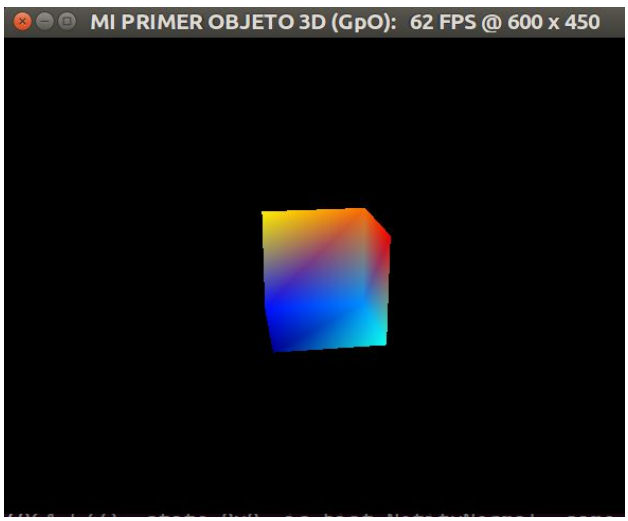

La función anterior se ejecuta una vez durante la inicialización de la escena para mandar el objeto a la GPU. Luego, en cada refresco de pantalla usamos dibujar_indexado(obj) para pintar el cubo. Esta función hace "activo" el objeto dado, da la orden de dibujar y luego lo "desactiva".

El cubo se mostrará inmóvil en el origen (estamos usando M=identidad en render_scene) visto desde la posición del observador en (X=4, Y=4, Z=2). Compilar y ejecutar el programa. ¿Se visualiza correctamente el cubo?

No se visualiza correctamente el cubo cuando empieza a girar debido a que las últimas caras pintadas son las que permanecen. Por eso cuando gira vemos las caras desde detrás, una especie de forma negativa de visualización.




Usando las técnicas que vimos en el laboratorio anterior, haced girar el cubo alrededor del eje Z con una velocidad de 30º por segundo. Basta usar para la matriz M (en vez de la identidad) la correspondiente matriz de rotación (creada con rotate). Desde la posición del observador el cubo debe verse girar de izquierda a derecha. ¿Se visualiza correctamente el resultado? ¿Os parece que cambia en algún momento el sentido de giro?

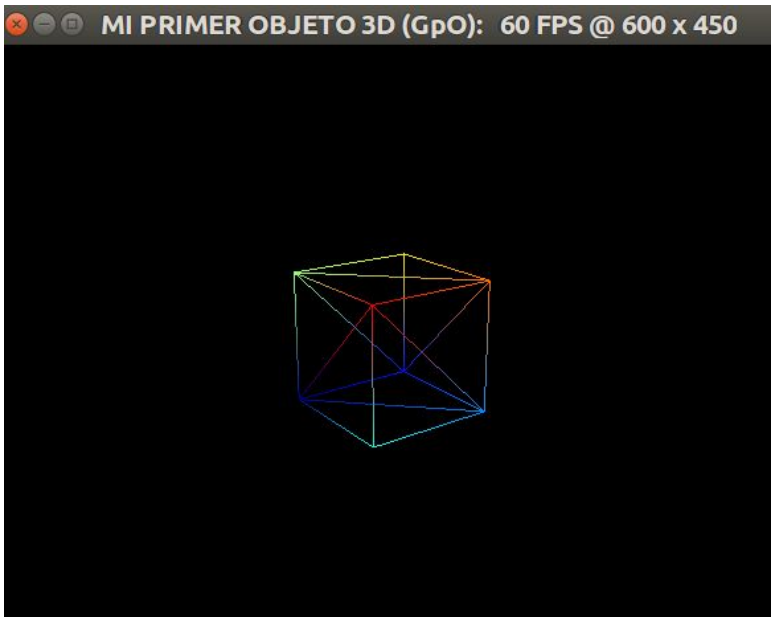



Parece que la cara de detras se visualiza tapando la cara de delante. Depende de la percepción del observador se puede ver en algunos puntos como parece que cambia de sentido.

Si cuando creamos un objeto vemos cosas raras al visualizarlo, para ver el origen del problema, a veces es bueno pintar sólo las líneas de cada polígono. En OpenGL, basta añadir la siguiente orden en la inicialización de la escena:

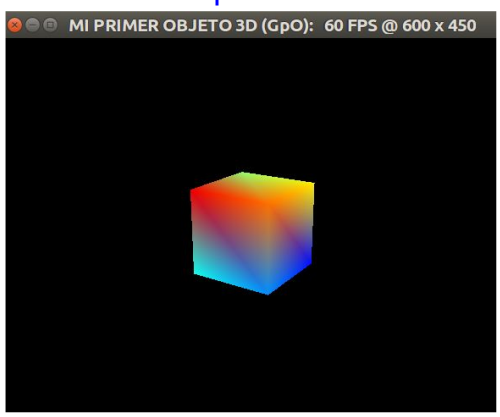
```
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
```

Esta orden afecta a las opciones del rasterizador de openGL y hace que solo se procesen los fragmentos que caen en los bordes del triángulo y no los del interior. Adjuntad captura de pantalla ¿Es correcto el modelo del cubo?




El modelo se dibuja correctamente en todas las caras al especificar la opción de `GL_FRONT_AND_BACK` a la función.

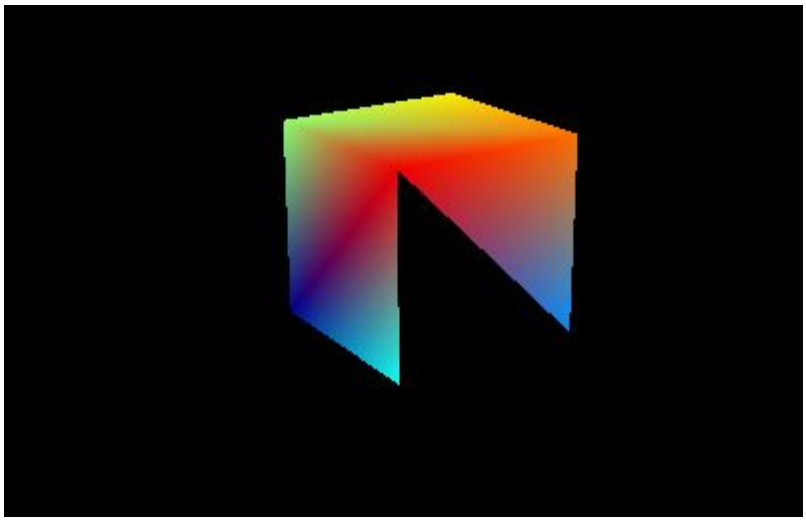
El problema parece estar en la visualización de las caras. Vamos a ver cómo podemos arreglarlo. **Quitar la orden anterior** (queremos ver las caras del objeto) y habilitar la eliminación de triángulos con glEnable(GL_CULL_FACE) en la inicialización de la escena. ¿Se ve correctamente? ¿Qué explicación le daríais al comportamiento inicial?




Al dar vueltas hay ciertos momentos en los que ciertas cara del cubo no se pintan. Esto se debe a que al inicio, al pintar el cubo la primera vez, ciertas caras tenian un ángulo > 90 con el observador y por tanto no se pintan. Al girar el cubo, esas caras no cambian su vector y siguen sin pintarse.

El culling lo conseguimos mediante: `glEnable(GL_CULL_FACE);`

Manteniendo la habilitación de GL_CULL_FACE cambiar ahora en el vector de índices la tripleta 2,7,3 por 3,7,2. Volver a ejecutar. Describir lo observado y explicar por qué está pasando.



Desaparece un triángulo. Al poner los vértices al revés el triángulo está mirando hacia dentro y desde fuera no se ve. Si estuviéramos dentro del cubo podríamos ver el triángulo.

**Restaurar la tripleta de vértices a su valor original para volver a visualizar de forma correcta el cubo (usaremos este código en ejercicios posteriores).**

**Ejer 2:** Partir del código del ejercicio anterior, deshaciendo el cambio de índices final, de forma que el cubo se vea girando correctamente. Se trata de añadir un 2º cubo en la escena con las siguientes características:

a) S: Escalado con factores (sx,sy,sz)=(0.8,0.4,0.2) respecto al original.

b) R: Girando sobre si mismo sobre el eje (1,0,1) al doble de la velocidad que el cubo original (60º por segundo).

c) T: Siguiendo trayectoria circular en el plano XY de radio 1.5 dando una vuelta ($2\cdot\pi$ rads) cada 6 segundos.

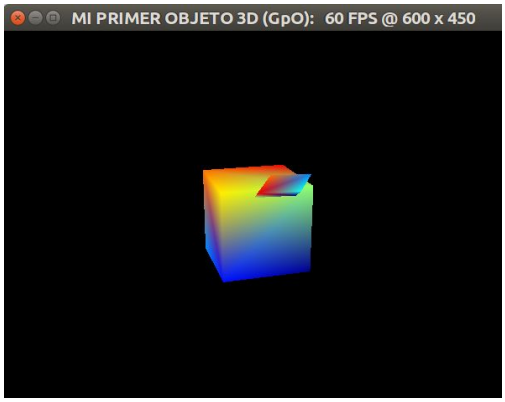
Adjuntar código añadido en render_scene().

```
// trayectoria de r=1.5 y 1 vuelta cada 6 segundos
T = translate( (float)(1.5 * sin((2*3.14*tt)/6.0)),
               (float)(1.5 * cos((2*3.14*tt)/6.0)),
               0.0f);
// Escalado
S = glm::scale(0.8f, 0.4f ,0.2f);

// Rotacion 60º/segundos en el vector (1,0,1)
R = glm::rotate(60.0f*tt,vec3(1,0,1));
M = T * S * R;

transfer_mat4("MVP",P*V*M);
 dibujar_indexado(obj2);
```

¿Se ven bien los dos objetos por separado? ¿Qué problema observáis? Adjuntar una captura de la ventana donde se observe el problema.
El segundo cubo debería aparecer por detrás del primero, pero aparece por delante al haber sido dibujado después.




Solucionarlo activando el uso del z-buffer en el programa (ver las notas de las transparencias). Volver a adjuntar una captura donde se vea que funciona.

• Al crear la ventana, indicar que vamos a usar un z-buffer

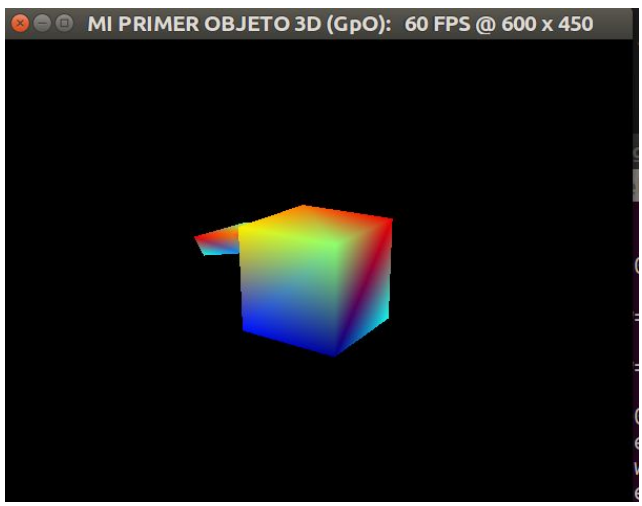
```
glutInitDisplayMode(GLUT_RGBA|GLUT_DOUBLE|GLUT_DEPTH);
```

• En la inicialización de la escena (init_scene) indicar a OpenGL que compruebe la profundidad del fragmento antes de rellenar el píxel:

```
glEnable(GL_DEPTH_TEST);
```

• Cada vez que vamos a recrear la escena hay que hacer un reset del z-buffer, al igual que hacíamos con el color de fondo:

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

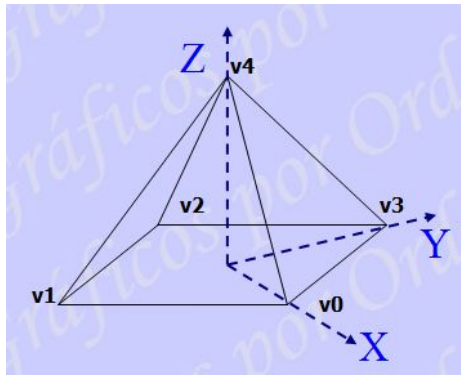


**Ejer 3:** Partiendo del código solución del primer ejercicio duplicar la rutina crear_cubo(), cambiándole su nombre a crear_piramide(). Se trata ahora de crea una pirámide con la base en el plano XY y su vértice en el eje Z a una altura de 1 sobre el origen. Las coordenadas y los colores a usar para sus 5 vértices son:

```
 1.0f,  0.0f, 0.0f,    0.00f, 0.00f, 1.00f,   // v0
 0.0f, -1.0f, 0.0f,    0.00f, 1.00f, 0.00f,   // v1
-1.0f,  0.0f, 0.0f,    0.00f, 1.00f, 1.00f,   // v2
 0.0f,  1.0f, 0.0f,    1.00f, 0.00f, 0.00f,   // v3
 0.0f,  0.0f, 1.0f,    1.00f, 1.00f, 0.00f,   // v4
```

Las filas son los 5 vértices y las columnas las coordenadas XYZ y colores RGB. El formato es idéntico al usado en crear_cubo() y puede usarse directamente (cortar/pegar) para cambiar los datos de vertex_data en la nueva función.

Los vértices anteriores (v0-v4) corresponden a los representados en la figura adjunta. A partir de ellos dar la lista de índices especificando los 6 triángulos que conforman la pirámide (2 para la base y 1 por cada cara restante). Los índices se refieren a los vértices desde el 0 hasta el 4. Hay que ser cuidadosos y respetar el orden de los vértices para aseguraros de que todos "miran" hacia fuera. Recordar que los ejes deben recorrerse en sentido contrario a las agujas del reloj si se miran desde la cara que deseamos que esté mirando al "frente".




Adjuntar vuestro listado de índices (en formato de 6 filas con los 3 vértices especificando cada triángulo en cada fila).

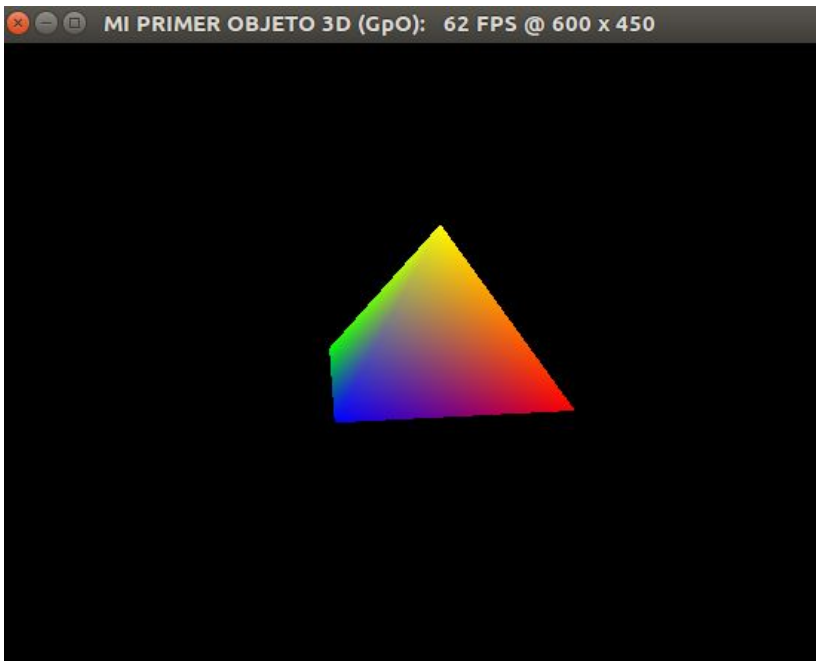
```
GLfloat vertex_data[] = {
    1.0f, 0.0f, 0.0f, 0.00f, 0.00f, 1.00f,  // v0
    0.0f, -1.0f, 0.0f, 0.00f, 1.00f, 0.00f, // v1
    -1.0f, 0.0f, 0.0f, 0.00f, 1.00f, 1.00f, // v2
    0.0f, 1.0f, 0.0f, 1.00f, 0.00f, 0.00f,  // v3
    0.0f, 0.0f, 1.0f, 1.00f, 1.00f, 0.00f,  // v4
};

GLbyte indices[] = {0, 4, 1,  // Cara 1
                    1, 4, 2,  // Cara 2
```

```
2, 4, 3,  // Cara 3
3, 4, 0,  // Cara 4
0, 1, 2,  // Base 1
0, 2, 3}; // Base 2
```

Una vez lista la función añadid obj2=crear_piramide(); en la inicialización de la escena, para tener disponible el nuevo objeto en la tarjeta gráfica.

Para la visualización apropiada de la pirámide habilitamos culling: `glEnable(GL_CULL_FACE);`




En render_scene() dibujad ahora el objeto pirámide, aplicándole las siguientes transformaciones:

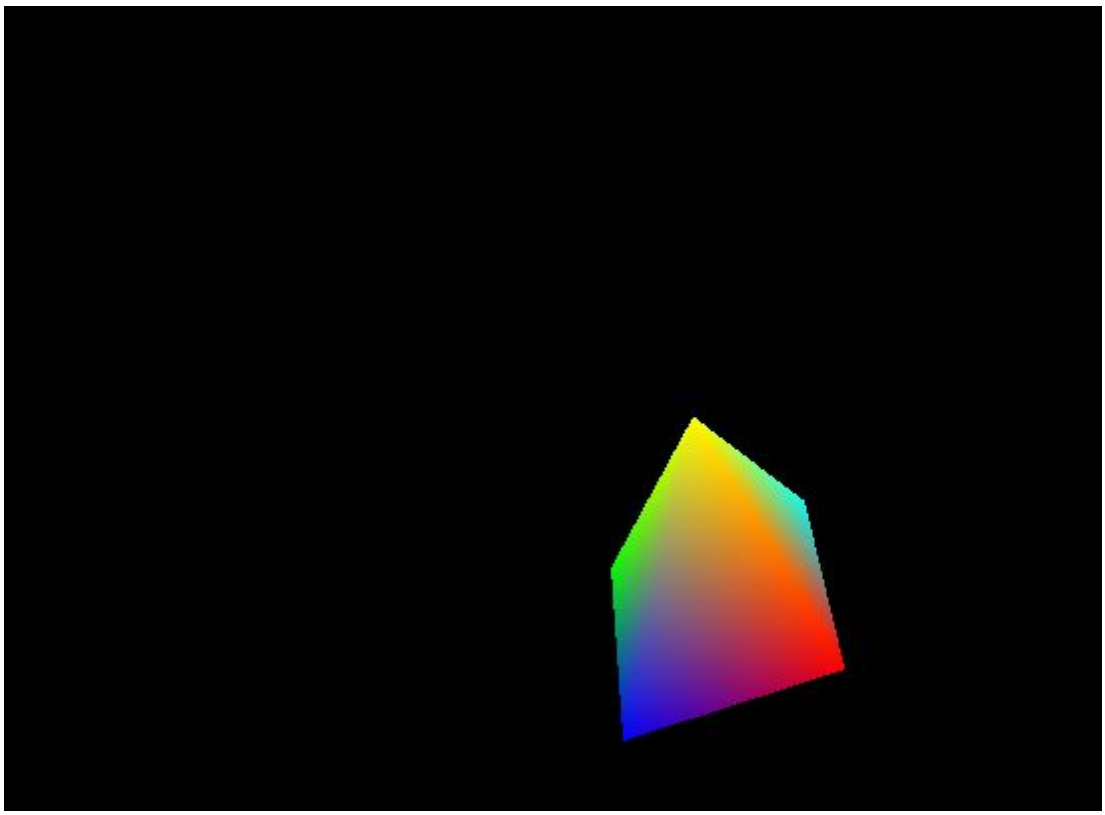
1) Un escalado S con factores sx=sy=0.5 y sz=0.8
2) Una rotación R sobre si mismo alrededor del eje Y (0,1,0) con una velocidad de 50 grados por segundo.
3) Una translación T describiendo un círculo en el plano XY de radio 2.

Adjuntar el código añadido en render_scene() para crear las matrices S,R,T y transferir la matriz producto al programa de la tarjeta gráfica.
Adjuntar captura de pantalla durante la ejecución del programa.

```
// Translación en el plano XY de radio 2
T = translate(2 * sin(tt), 2 * cos(tt), 0.0f);
// Escalado
```

```
S = glm::scale(0.5f, 0.5f ,0.8f);
// Rotacion 50°/segundos en el vector (0,1,0)
R = glm::rotate(50.0f*tt,vec3(0,1,0));
M = T * S * R;

transfer_mat4("MVP",P*V*M);
dibujar_indexado(obj2);
```



**Ejer 4:** En este ejercicio queremos recrear la funcionalidad que vimos en MATLAB para poder observar un objeto 3D desde diferentes perspectivas pinchando y arrastrando el ratón sobre la ventana.

Partir de nuevo del código solución del primer ejercicio y usarlo para dibujar el cubo original sin rotaciones, escalados o translaciones (volviendo a usar como matriz M la identidad). El cubo debe verse quieto y centrado en el origen.

Definir ahora dos variables globales (de tipo float) que definirán la posición del observador a través de dos ángulos: azimuth ( $\theta$ ) y elevación ( $\phi$ ):

$$\text{pos\_obs} = (D\cdot\cos(\theta)\cdot\cos(\phi),\ D\cdot\sin(\theta)\cdot\cos(\phi),\ D\cdot\sin(\phi))$$

El azimuth corresponde a la dirección desde donde se observa el objeto y se da en el intervalo $[0,\ 2\cdot\pi]$ (en radianes). Los valores posibles para la elevación están entre $-\pi/2$ y $\pi/2$ radianes (+/- 90º), indicándonos si vemos el objeto desde arriba (elevación positiva) o desde abajo (elevación negativa).

Inicializar ambas variables de azimuth y elevación a 0.6. D es la distancia del observador al origen y será una constante D=3.

En render_scene(), calcular la posición del observador a partir del azimuth y la elevación y construir la matriz V. Asumimos que el observador está siempre mirando al origen (target).

Se trata de escribir un código que al arrastrar el ratón teniendo pinchado uno de sus botones cambie los valores de azimuth y elevación, lo que cambiará la posición del observador y por lo tanto el punto de vista desde el que vemos el objeto. Para que sea más intuitivo el azimuth debe cambiar al mover el ratón en la horizontal (x) y la elevación al moverlo en la vertical (y).

En la función que detecta el movimiento del ratón mientras tenemos un botón pulsado (asignada con glutMotionFunc) debemos:

- Restar la posición actual x,y de la última guardada (x0,y0) para obtener los desplazamientos relativos dx, dy entre llamadas a la función.

- Usar la variaciones (dx,dy) para incrementar el azimuth y la elevación. Usar una constante de proporcionalidad del orden de 0.005. De esta forma por cada píxel que desplacemos el ratón (dx=1) en la horizontal tendremos un incremento del azimuth de 0.005 rads.

- Guardar los últimos valores x,y en x0,y0 para la próxima llamada.

El problema de este enfoque es que x0,y0 estarán indefinidos en la primera llamada a la función. Para solucionarlo los declararemos globales (fuera del cuerpo de las funciones) y los inicializaremos en la función de detectar el click (asociada con glutMouseFunc) que ser ejecutará siempre de forma previa.

El proceso, desde el punto de vista de los eventos, sería:

- Al **pulsar el botón** se guarda su posición inicial del ratón (x0,y0) sobre la ventana en una primera función asignada con glutMouseFunc().

- Al **arrastrar el ratón** (sin soltar el botón) se llama continuamente a la segunda función que calcula los incrementos dx,dy desde la última vez, y los usar para modificar las variables de azimuth y elevación.

Adjuntar código de las dos funciones asociadas a los eventos de ratón.

```
void mouse_click(int but,int state, int x, int y)
{
    mx0 = x;
    my0 = y;
}

void mouse_mov(int x, int y)
{
    float dx = (float) (x - mx0);
    float dy = (float) (y - my0);
    printf("********************++dx=%f, dy=%f\n", dx,dy);
    azimuth += 0.005f * dx;
    elevacion += 0.005f * dy;
    mx0 = x;
    my0 = y;

}
```

Añadir código (un par de líneas) para poder usar la rueda del ratón para acercar/alejar la posición del observador (modificando la variable D)
Añadiendo el evento a glutMouseWheelFunc

```
void keyboard(unsigned char, int, int);
void key_special(int, int, int);
void mouse_click(int,int, int, int);
void mouse_mov(int,int);
void mouseWheel(int, int, int, int);


void eventos_teclado_mouse()
{
    glutKeyboardFunc(keyboard);      // Caso de pulsar alguna tecla
//  glutSpecialFunc(key_special);  // Teclas de funci□n, cursores, etc

    glutMouseFunc(mouse_click);            // Eventos del rat□n
    //glutPassiveMotionFunc(mouse_mov);   // Mov del ra□n
    glutMotionFunc(mouse_mov);
    glutMouseWheelFunc(mouseWheel);
}
```

```
void mouseWheel(int button, int dir, int x, int y)
{
    if (dir > 0)
    {
        // Zoom in
        D += 0.005;
    }
    else
    {
        // Zoom out
        D -= 0.005;
    }

    return;
}
```

Adjuntar vuestro fichero fuente completo como lab4.cpp

**Ejer 5:** Partir del código del ejercicio 3 (la pirámide moviéndose por la pantalla) y modificarlo para que cuando se pulse la tecla 't' se cambie entre la visualización de la pirámide y el cubo original. Esto es, al pulsar 't' el cubo debe sustituir a la pirámide (con sus mismos movimientos). Al pulsar de nuevo 't' debe volver a aparecer la pirámide.

Indicad qué partes del código habéis modificado y añadir las modificaciones. (basta añadir 3 o 4 líneas en un par de sitios del programa).

**Ejer 6:** La relación entre la coordenada z con respecto al observador (que corresponde a la distancia al observador) y la correspondiente coordenada zn normalizada viene dada por la expresión:

$$z_n = \frac{1}{(f-n)}\left((f+n)+\frac{2f\cdot n}{z}\right)$$

Donde (z=-n) y (z=-f) son los planos cercanos y lejanos de la proyección. Recordad que el eje Z en coordenadas cámara apunta hacia atrás, por lo que las coordenadas z por delante de la cámara son negativas.

a) Dar la expresión de z en función del valor normalizado zn para valores del plano cercano n = 2 y plano lejano f = 100.

b) Cuando usamos un z-buffer para ocultación, en dicho buffer se van guardando los valores de zn de los fragmentos que vamos pintando. Considerar un zbuffer con una profundidad de 8 bits, lo que solo permite 256 niveles. Como hay que cubrir un intervalo de 2 (de -1 a 1) el salto en la coordenada zn debido a la discretización en 256 niveles será:

$$\delta z_n = \frac{2}{256} = \frac{1}{128}$$

Calcular el correspondiente salto en la coordenada z (no normalizada) que se provoca con un salto mínimo en zn (1/128), cerca de ambos extremos.

¿Cuál es la precisión (%) de la coordenada z cerca de z=-n? ¿Y de z=-f?

Haced vuestros cálculos en un papel y adjuntar una foto o scanner.