Apellidos: Lagoa Rúa	Nombre: Brais
Apellidos: Maruri Pérez	Nombre: Juan

Objetivos de este laboratorio:

1. Crear objetos 3D sencillos a partir de sus vértices y usando índices para describir los triángulos de sus caras. Ver como el orden en el listado de vértices define la orientación (cara de delante y detrás) de un polígono.
2. Activar en OpenGL el descarte de polígonos **(GL_CULL_FACE)** y/o la ocultación de fragmentos **(GL_DEPTH_TEST)**. Entender las diferencias entre ambas opciones.
3. Crear una sencilla interfaz de usuario para mover la posición del observador en la escena de forma interactiva.
4. Comenzar con el uso de texturas en OpenGL: entender el concepto de coordenadas de textura y como usarlas dentro de los "shaders" para recubrir un modelo3D.

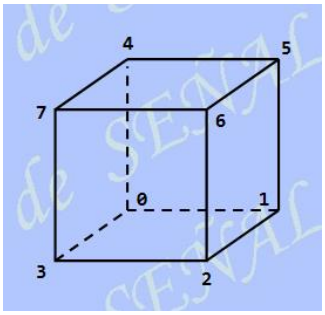
Esta práctica estará separada en dos bloques. El bloque q estaŕa centrado en las técncias de optimización vistas en clase, correspondientes a los puntos 1 al 3 del apartado anterior. El segundo bloque es una primer acercamiento a las coordenadas de textura y su mapeo dentro de los shaders. La entrega de esta práctica deberá cubrir ambos bloques.

Para el primer bloque de esta práctica, seaconseja que para trabajar guardéis vuestro código como GpO_02_entrega_ejer1.cpp, GpO_02_entrega_ejer2.cpp. En cada ejercicio se indica el código inicial a usar, que puede ser el programa suministrado "GpO_02.cpp" o el resultado de alguno de los ejercicios anteriores. Se seguirá la misma filosofía para el segundo bloque de esta práctica, donde los ficheros a utilizar se denomina GpO_03.cpp.

Junto con el fichero de respuestas adjuntar los ficheros fuente GpO_02_entrega_ejer[1|2|3|4].cpp correspondientes a cada uno de los ejercicios planteados en el primer bloque y GpO_03_entrega_ejer[1|2] para el segundo bloque.

Bloque 1: Optimización del renderizado

Ejercicio 1 (partir de GpO_02): En la función **`crear_cubo()`** se especifican las posiciones de los 8 vértices de un cubo (junto con sus colores), que tiene lado L=2 y está centrado en el origen. Adicionalmente se define una lista de 36 (12x3) índices. Cada una de esas 12 tripletas define un triángulo del modelo (12 triángulos especifican las 6 caras de un cubo). En este caso el modelo usa 36=12x3 índices pero sólo 8 vértices.



Como es habitual la función **crear_cubo()** se llama una vez en **init_scene()** para crear el objeto y mandar sus datos a la GPU. Luego, en cada refresco de pantalla **render_scene()**, usamos la función **dibujar_indexado()** para pintar el cubo. En esta función lo único que hacemos es activar el VAO (**glBindVertexArray**), dar la orden de dibujo (**glDrawElements**) y desactivar el VAO.

En la función **render_scene()** el cubo se gira 30º en el eje Z usando una matriz de rotación creada con `rotate()`, si notáis que va muy rápido en vuestros equipos, podéis rebajar este ángulo. El observador está mirando desde arriba (pos_obs=0,6,4) hacia el centro del cubo (target = 0, 0, 0). Compilar y ejecutar el programa. El cubo debería visualizarse correctamente.

Ahora vamos a hacer girar el cubo alrededor del eje Z con una velocidad de 30º por segundo. En los argumentos de la función **rotate()** cambiad el ángulo de giro fijo (30º) por **30.0f*t.** `Igualmente, si veís gira demasiado rápido, reducid el ángulo de giro.` El valor de t es un contador de tiempo en segundos obtenido con glfwGetTime(). De esta forma el ángulo de rotación se incrementa 30º cada segundo y por lo tanto el cubo se verá girando a esa velocidad. Mantened el mismo eje de rotación (eje Z=0,0,1).

Desde la posición del observador el cubo debe ahora verse girar en sentido contrario a las agujas del reloj. ¿Se visualiza correctamente el resultado? No, hay algún problema porque hace un artefacto extraño al girar con una de sus caras. En la inicialización de la escena (**init_scene**), habilitar la eliminación de los triángulos que no miren al observador con la orden **glEnable(GL_CULL_FACE)**. ¿Se ahora ve girar correctamente al cubo? ¿Cuál era el problema antes? Sí, se ve correctamente. El problema es que sin culling ni z-buffer, OpenGL dibuja todas las caras del cubo en el orden en que se especifican en los vértices, sin importar si estan en la parte frontal o trasera. Y si una cara trasera se dibuja después de una frontal, se superpondrá incorrectamente, porque debería estar oculta.

Manteniendo la habilitación de GL_CULL_FACE, cambiar ahora en el listado de índices la última tripleta (2,7,3) por (2,3,7). Volver a ejecutar. Explicar lo que está pasando. Ahora uno de los dos triángulos de una cara cuando pasa por delante no se ve y cuando está detrás se superpone a los que están delante. Cambiar ahora la tripleta a 3,2,7. ¿Hay cambios? Sí, se ve correctamente de nuevo. ¿Por qué? El problema es que OpenGL determina si un triángulo es frontal o trasero basándose en el orden de los vértices. Antihoraria es cara frontal y se dibuja. Horario es cara trasera y se elimina si culling está activado. Por esto cuando el triángulo se definió como (2,3,7), horario, se invirtió cuando el triángulo estaba oculto y visible. En la última opción (3,2,7) los vertices vuelven a estar en sentido antihorario y con el culling activa el cubo se ve correctamente.

Se trata ahora de añadir un segundo cubo en la escena con las siguientes características (aplicadas en este orden):

a) Escalado respecto al original con factores (sx,sy,sz)=(0.6,0.6,0.6).

b) Girando sobre sí mismo alrededor del eje X (1,0,0) a 50º por segundo.

c) Desplazándose en una trayectoria circular en el plano XY de radio 2.5 (similar a como lo hicimos con el triángulo en el LAB anterior).

Usar las funciones scale, rotate y translate de la librería glm para obtener las matrices S, R y T correspondientes a estos movimientos y combinarlas en el orden adecuado para crear la matriz M=T·R·S adecuada. Adjuntar el código añadido en render_scene().

```
//Segundo cubo
float angle = t * 50.0f; // 50 grados por segundo
float radius = 2.5f;

//Escalado
S = scale(mat4(1.0f), vec3(0.6f, 0.6f, 0.6f));

//Rotacion
R = rotate(radians(angle), vec3(1.0f, 0.0f, 0.0f));

//TRalacion
T = translate(mat4(1.0f), vec3(radius * cos(t), radius * sin(t), 0.0f));
M = T * R * S;
transfer_mat4("MVP", Proy * View * M);
dibujar_indexado(obj2);
```

¿Qué problema se observa en la visualización?

El cubo, que parece un satélite orbitando el cubo 1, se superpone cuando pasa por detras suyo, debería perderse de vista.

La opción **`glEnable(GL_CULL_FACE)`** eliminaba las caras que no miraban hacia al observador, pero no soluciona el problema de la ocultación cuando hay más de un objeto (o si el objeto es no convexo). En estos casos hay que activar el uso del z-buffer en el programa (ver las notas de las transparencias). Indicar las modificaciones efectuadas en el código y captura donde se aprecie que la visualización es ahora correcta.

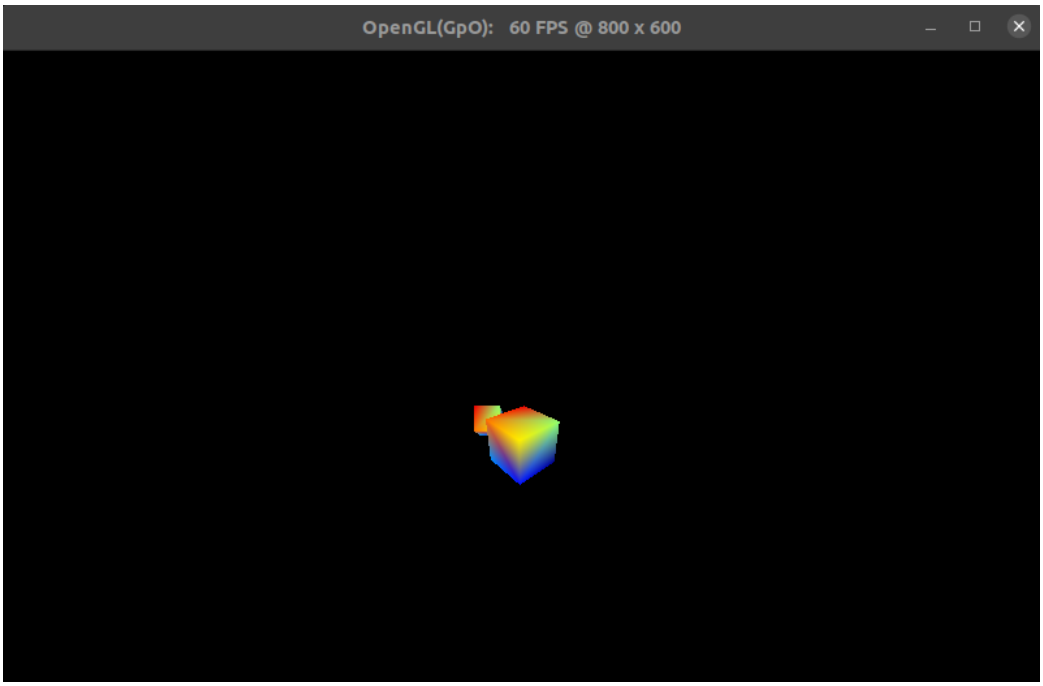
Se han añadido en init_scene():

glEnable(GL_DEPTH_TEST); // Habilita el test de profundidad (Z-Buffer)
glDepthFunc(GL_LESS); // Un fragmento se dibuja si su Z es MENOR que el almacenado

Y también se ha modificado una linea de render_scene:

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

Antes solo se borraba el buffer del color. Ahora gracias al or binario '|', se combinan los dos flags para que OpenGL borre el Z-buffer, que guarda la distancia de cada pixel observado, además del buffer del color.



Ejercicio 2: partid de **GpO_02.cpp** habilitando la eliminación de caras que no miren al observador con la orden **`glEnable(GL_CULL_FACE)`**.

El objetivo de este ejercicio es hacer una sencilla interfaz de usuario para cambiar el punto de vista desde donde se observa un objeto 3D. Definiremos en el código tres variables globales (fuera del cuerpo de las funciones) de tipo float llamadas d, az y el (**d**istancia, **az**imuth y **el**evación). Estas variables determinarán la posición del observador:

$$pos\_obs = d \cdot vec3(\sin(az)\cdot\cos(el),\ \cos(az)\cdot\cos(el),\ \sin(el))$$

El parámetro d es la distancia del observador al origen, que coincide con el punto hacia donde estamos mirando (vector target). El azimuth (az) es la dirección (en el plano horizontal XY) desde donde se observa el objeto, con un valor en el intervalo $[0,2\pi]$ radianes que equivale a [0,360º]. La elevación (el) toma valores entre $-\pi/2$ y $\pi/2$ rads (± 90º) y nos indica si estamos viendo el objeto desde arriba (elevación positiva) o desde abajo (elevación negativa).

Al declarar estas variables inicializadlas con valores d=8.0f, az=0.0f y el=0.6f. En render_scene(), calculad la posición del observador pos_obs con la fórmula anterior antes de usarla para crear la matriz View. El vector target no cambia (esté donde esté el observador mantiene la vista en el origen).

En este ejercicio usaremos 3 acciones del ratón (o del touchpad): pulsación de algún botón, movimiento y "scrolling" con la rueda del ratón. Las funciones de la librería GLFW que nos permiten asociar una función de "callback" a estos eventos (para ejecutarse cuando sucedan) son:

- Botones: **glfwSetMouseButtonCallback(window,pulsar)**
- Movimiento: **glfwSetCursorPosCallback(window,mover)**
- Scrolling: **glfwSetScrollCallback(window,scroll)**

Las funciones asociadas que se indican las tenemos que escribir nosotros para conseguir nuestros objetivos y deben seguir los siguientes templates:

- **void pulsar(GLFWwindow* window, int Button, int Action, int Mode)**
  El parámetro **Button** nos da el botón pulsado (0=izquierdo, 1= derecho). **Action** nos indica si se ha pulsado (1) o liberado (0). **Mode** si se ha pulsado en combinación con alguna otra tecla (Shift, Ctrl, …).

- **void mover(GLFWwindow* window, double x, double y)**
  Los parámetros x, y nos dan la posición actual del cursor en la ventana.

- **void scroll(GLFWwindow* window, double dx, double dy)**
  Los parámetros dx,dy nos dan el "scrolling" en los ejes X e Y. En el caso de usar la rueda del ratón dx=0 y solo tenemos "scrolling" vertical.

En primer lugar usaremos el scrolling para cambiar la distancia d y ver el cubo más o menos grande al acercar o alejar el observador. Cambiaremos la distancia d (a saltos de ±0.25) según el signo del parámetro dy en la función 'Callback' de scrolling. Se trata de aumentar d girar la rueda hacia delante (como si lo empujáramos) y viceversa (como si lo acercáramos con la rueda del ratón). Para evitar problemas de visualización debido a los planos Znear y Zfar, poner unos topes para que la distancia d no se haga mayor de 22 ni menor de 4. Adjuntad el código de la función de scroll.

```
//Callback Scroll
void scroll(GLFWwindow* window, double dx, double dy){

d += (dy > 0) ? 0.25f : (dy < 0) ? -0.25f : 0.0f;

// Limitar d entre 4 y 22
d = glm::clamp(d, 4.0f, 22.0f);
```

```
fprintf(stdout, "Scroll: dy=%.1f, d=%.1f\n", dy, d);

}
```

Cuando uséis una de estas funciones por primera vez es aconsejable empezar haciendo un volcado de sus parámetros: `fprintf(stdout,"x %.1f y %.1f\n",dx, dy);` La idea es entender su comportamiento viendo los valores que toma dy al mover la rueda en un sentido u otro.

El siguiente paso es "rotar" nuestro cubo arrastrándolo con el ratón para verlo desde distintas posiciones. Para ello vamos a escribir un código de forma que cuando se mueva el ratón teniendo pulsado el botón izquierdo se modifiquen los valores de las variables de azimuth (az) y elevación (el). Esto cambia la posición del observador y por lo tanto el punto de vista desde el que vemos el objeto. Para que sea intuitivo, el azimuth se cambiará con el movimiento del ratón en la horizontal (eje x) y la elevación con su movimiento en vertical (y).

En este caso usamos el evento de pulsación (que detecta el inicio del proceso) y del movimiento del cursor (para saber cuánto hay que girar el cubo según el movimiento del ratón). Hay varias formas de implementar esta funcionalidad. En este guión os sugiero una posibilidad, pero podéis hacerlo de cualquier otra si os parece más sencillo. Lo primero es asociar la función "pulsar" con **`glfwSetMouseButtonCallback()`** para detectar la pulsación de un botón en el ratón. Dentro de esta función determinaremos si se ha pulsado o no el botón de la izquierda. La idea es activar el "sensor" de movimiento (asociando la función mover) al pulsar el botón izquierdo y desactivarlo al liberar el botón. El botón pulsado y su estado puede conocerse con los parámetros de la función. Para "desactivar" el movimiento (eliminar la asociación de la función de "Callback") llamad otra vez a **`glfwSetCursorPosCallback()`** pero usando ahora NULL como función asociada.

El proceso completo, desde el punto de vista de los eventos, quedaría:

- Al **pulsar el botón izquierdo** guardamos la posición inicial del ratón en la ventana con la función **`glfwGetCursorPos(window,&xp,&yp).`**Declarad (xp,yp) como variables de tipo double globales (declaradas fuera de las funciones). Luego activamos el "sensor" de movimiento con la función **`glfwSetCursorPosCallback()`** asociandole la función **`mover().`**

- Al **arrastrar el ratón** (sin soltar el botón) se ejecutará continuamente la función asociada **`mover()`** cuyos parámetros son la posición actual del ratón (x,y). Dentro de esta función calculamos los desplazamientos DX, DY con respecto a la última posición guardada (xp,yp) y actualizamos xp, yp con la posición más reciente. Luego usamos los desplazamientos DX, DY para incrementar los ángulos de azimuth (DX) y elevación (DY). El factor usado para modificar ambos ángulos dependerá también de la

distancia d. Para evitar movimientos exagerados usaremos F=(0.007·d). Dado un cierto movimiento del ratón (DX,DY) el correspondiente ángulo debe incrementarse por F·DX o F·DY respectivamente.

- Al **liberar el botón izquierdo** desenlazad la rutina **`mover()`** del evento de mover el cursor con **`glfwSetCursorPosCallback(window,NULL)`**.

Como paso previo, no cambiar los valores de **az** y **el** en la función mover(). Simplemente volcad por pantalla las posición x,y del ratón. Lo importante es conseguir que al pulsar el botón y mover el ratón empiece a salir información por pantalla y que se detenga al liberar el botón (al dejar de ejecutarse la función mover). Cuando esto funcione, calculad y volcad los desplazamientos (DX, DY) y comprobad que salen valores adecuados. Finalmente usar esos valores para modificar los valores de los ángulos de azimuth y elevación como hemos indicado. En ese momento el cubo debe seguir vuestros movimientos.

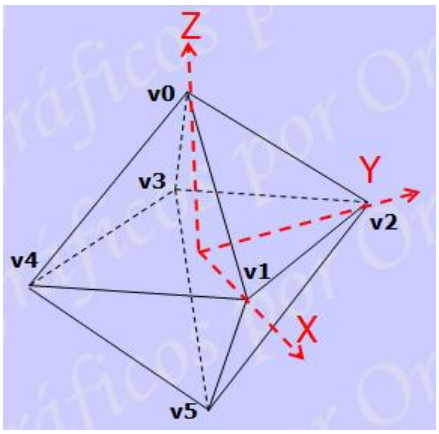
La interfaz debe ser intuitiva en el sentido de que si pinchamos en el vértice más "cercano" y movemos el ratón el vértice debe verse moverse siguiendo nuestros movimientos como si lo estuviéramos arrastrando. Adjuntar vuestro código para las funciones asociadas a la pulsación y el movimiento del ratón.

```
//Callback pulsar

void pulsar(GLFWwindow* window, int Button, int Action, int Mode) {
fprintf(stdout, "Botón: %d | Acción: %d | Mod: %d\n", Button, Action, Mode);
if (Button == GLFW_MOUSE_BUTTON_LEFT) {
mouse_pressed = (Action == GLFW_PRESS); // True al pulsar, False al soltar


fprintf(stdout, "Botón izquierdo: %s\n", mouse_pressed ? "PULSADO" :
"LIBERADO");

// Guardar posición inicial al pulsar
if (mouse_pressed) {
glfwGetCursorPos(window, &last_x, &last_y);

fprintf(stdout, "Posición inicial: (%.1f, %.1f)\n", last_x, last_y);
}
}
}


//Callback mover
void mover(GLFWwindow* window, double x, double y){
if (mouse_pressed) {
// Calcular diferencia respecto a la última posición
double dx = x - last_x;
```

```
double dy = y - last_y;

fprintf(stdout, "Desplazamiento: (%.1f, %.1f)\n", dx, dy);

// Sensibilidad (ajustable)
float F = 0.005f * d;

// Actualizar ángulos
az -= dx * -F; // Rotación horizontal
el -= dy * -F; // Rotación vertical

fprintf(stdout, "Angulos: az=%.2f rad, el=%.2f rad\n", az, el);

// Limitar elevación para evitar volteretas
el = glm::clamp(el, -1.5f, 1.5f); // +- 85º

// Guardar posición actual para el próximo frame
last_x = x;
last_y = y;
}
}
```

Ejercicio 3 (partir de GpO_02_entrega_ejer1.cpp): Copiar y pegar la rutina crear_cubo(), cambiando su nombre a crear_octaedro(). El nuevo código debe crear ahora un octaedro con su base en el plano XY y dos vértices adicionales en el eje Z a una altura de ±1 sobre el origen. Las coordenadas y colores a usar para sus 6 vértices serán:

```
                    0.0f,  0.0f, 1.0f,     1.00f, 1.00f, 0.00f,   // v0
                    1.0f,  0.0f, 0.0f,     0.00f, 0.00f, 1.00f,   // v1
                    0.0f,  1.0f, 0.0f,
    1.00f, 0.00f, 0.00f,   // v2
                   -1.0f,  0.0f, 0.0f,
    0.00f, 1.00f, 1.00f,   // v3
                    0.0f, -1.0f, 0.0f,
    0.00f, 1.00f, 0.00f,   // v4
                    0.0f,  0.0f,-1.0f,
         1.00f, 0.00f, 1.00f,   // v5
```



Las columnas corresponden a las coordenadas XYZ + colores RGB de los seis vértices (filas) del poliedro. El formato es idéntico al usado en crear_cubo() por lo que podremos cortar/pegar directamente para cambiar los datos de vertex_data en la nueva función.

Los 6 vértices (v0-v5) anteriores corresponden a los representados en la figura adjunta. A partir de ellos dar la lista de índices que especifiquen los 8 triángulos

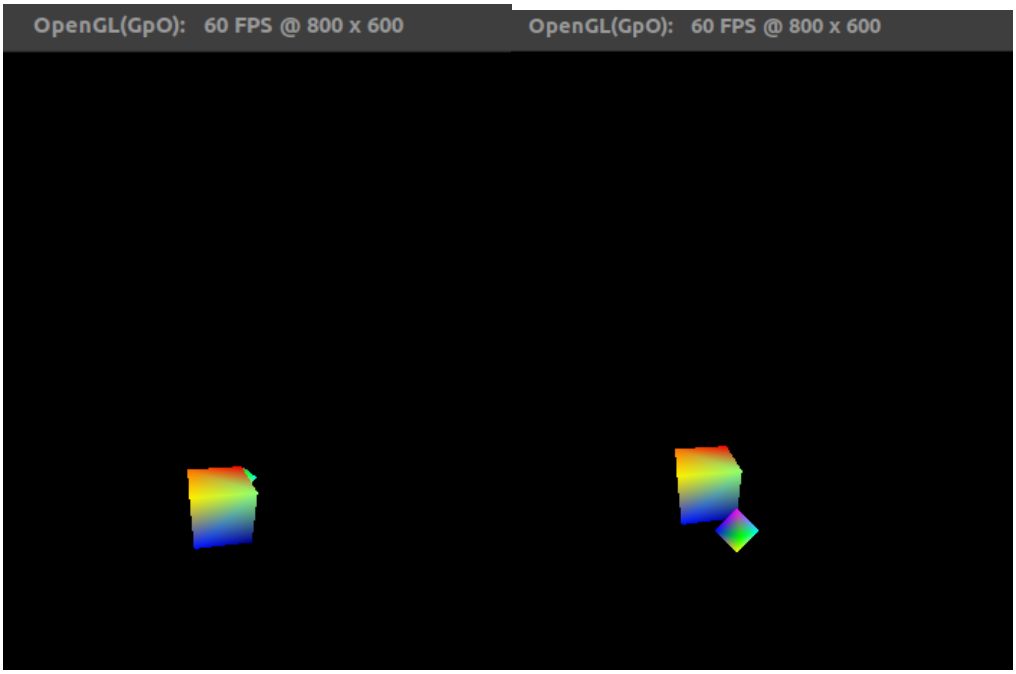
que forman el octaedro (1 por cara). Los índices se refieren a los vértices (del 0 al 5). Es importante fijaros en el orden de los vértices para que las caras "miren" hacia fuera. Recordad que con el criterio que usamos los vértices deben recorrerse en el sentido contrario a las agujas del reloj (vistos desde el exterior del poliedro).

Por ejemplo la primera cara podría especificarse con la tripleta de índices (0, 1, 2). Adjuntar el listado de índices en filas de 3 vértices cada una para cada uno de los triángulos. ¿Cuántos vértices e índices tenemos en este caso? Usadlos para modificar los campos `obj.Nv` y `obj.Ni` del nuevo objeto.

```
GLbyte indices[] =
{
0, 1, 2,
0, 3, 4,

0, 4, 1,
0, 2, 3,

5, 2, 1,
5, 4, 3,
5, 1, 4,
5, 3, 2
};

obj.Nv = 6;
obj.Ni = 8 * 3;
```

Una vez completada la función crear_octaedro(), modificar el código para que sea el octaedro el que aparezca dando vueltas alrededor del cubo central. Indicad los cambios (mínimos) efectuados el código y las funciones donde se añaden. Adjuntar una captura de pantalla durante la ejecución del programa.

Se ha añadido el codigo de crear_cubo(), en init_scene se ha activado el GL_DEPTH_TEST y GL_LESS y en render_scene se han pintado el cuadrado y el octaedro como se pedía.



Finalmente queremos que ambos objetos se intercambien al pulsar la tecla F1. Para ello usar la función (ya definida) **`KeyCallBack()`** que se ejecuta cada vez que se detecta la pulsación de una tecla. En la librería GLFW el código para la tecla F1 es GLFW_KEY_F1. Para evitar que el cambio se cancele al liberar la tecla, usad el parámetro **action** para detectar si la tecla se ha pulsado o se ha liberado y haced que el cambio tenga lugar solamente al pulsar la tecla. Adjuntad las modificaciones del código indicando en qué función se hacen.

Se crear una variable booleana global para saber si invertir o no, y se añade un case más al switch de KeyCallback que modifique dicho valor.

```
case GLFW_KEY_F1:
if (key == GLFW_KEY_F1 && action == GLFW_PRESS) { // Solo al pulsar (no al liberar)
invertirObjs = !invertirObjs; // Invertir el orden
}
```

Luego a la hora de dibujar en render_scene se añaden operadores ternarios en funcion de dicha variable booleana para ver si pintar el obj1(cuadrado) o obj2(octaedro)

```
(!invertirObjs) ? dibujar_indexado(obj2) : dibujar_indexado(obj1);


(!invertirObjs) ? dibujar_indexado(obj1) : dibujar_indexado(obj2);
```

Cada una de estas lineas despues de definir las matrices que definen los objetos en la escena.

Ejercicio 4 (partir del código de GpO_02_entrega_ejer2.cpp):

En el ejercicio 2 usábamos el botón izquierdo del ratón para cambiar el punto de vista. Se trata ahora de que al pinchar con el **botón derecho** del ratón en un punto de la ventana obtengamos sus coordenadas normalizadas. Luego podremos recuperar sus coordenadas cámara.

El nuevo código se añadirá a la función "pulsar" que ya está escrita con una condición para que solo se ejecute cuando el botón pulsado sea el derecho. El código que teníamos (en caso de pulsar el botón izquierdo) lo dejaremos sin tocar. Al igual que antes, usad la información del estado del botón para que esta información se muestre **SOLO al pulsar el botón**, no al liberarlo.

Lo primero es usar la función **`glfwGetCursorPos`** (usada en el ejercicio 2) para obtener la posición donde se ha pinchado con el ratón (xp,yp). Recordad que en las variables ALTO y ANCHO tenemos las dimensiones de la ventana. Antes de nada, haced yp=(ALTO-yp), ya que la función anterior devuelve la posición vertical del cursor con el origen arriba, mientras que openGL usa el criterio contrario.

Nos falta la coordenada z de profundidad. Al haberse hecho la proyección 2D podríamos pensar que esta coordenada se ha perdido. Sin embargo, sabemos que para pintar los objetos 3D, la coordenada z (de los fragmentos visibles) se actualiza en el z-buffer (al que podemos acceder y preguntar por sus valores). Definid una variable zp de tipo GLfloat y llamad a la función:

```
glReadPixels((GLint)xp,(GLint)yp,1,1,GL_DEPTH_COMPONENT,GL_FLOAT,&zp);
```

Esta función lee el contenido del z-buffer en la posición de la pantalla (xp,yp). El resultado (en zp) será un valor entre 0 y 1. Obviamente debemos habilitar previamente **el uso del z-buffer** como se hizo en el ejercicio 1. Si no es así obtendremos siempre el mismo valor de zp en cualquier punto (ya que el buffer no se actualiza). Añadid un fprintf() para ver las coordenadas xp,yp,zp usando:

```
fprintf(stdout, "xp= %.0f yp= %.0f zp= %.3f\n", xp, yp, zp);
```

A partir de (xp,yp,zp) podemos calcular las coordenadas normalizadas (xn, yn, zn de tipo float) con valores entre -1 y 1 como:

$$x_n = 2 \cdot \frac{x_p}{ANCHO} - 1 \qquad y_n = 2 \cdot \frac{y_p}{ALTO} - 1 \qquad z_n = 2 \cdot z_p - 1$$

Comprobad que $x_n \sim -1$ al pinchar en un punto por la izquierda de la ventana ($xp \sim 0$) y $x_n \sim 1$ cuando pinchamos por la derecha ($xp \sim ANCHO$). Para y_n sus valores deben ser ~1 al pinchar en la zona superior de la pantalla y ~-1 si lo hacéis en la parte inferior.

En cuanto a zn, cuanto mayor sea su valor más lejos estará el punto. Valores de zn=-1 corresponden a puntos en el plano cercano (Znear) y zn=1 a puntos sobre el plano lejano (Zfar). Si pincháis fuera del objeto el valor de zn es 1, que

es el valor con el que se inicializa el zbuffer (el más "lejano" posible). Luego, durante el proceso de dibujo, el valor del z-buffer se va actualizando si la coordenada z de los fragmentos que se van pintando es menor que la que está guardada en el buffer. Al final el buffer guarda la profundidad del punto más cercano al observador. Añadid un fprintf() para volcar las 3 coordenadas normalizadas xn,yn,zn con 3 decimales (%.3f).

```
xnorm = (2*xp/ANCHO)-1;
ynorm = (2*yp/ALTO)-1;
znorm = (2*zp) -1;
fprintf(stdout, "xnorm= %.3f ynorm= %.3f znorm= %.3f\n", xnorm, ynorm, znorm);
```

Conocidas la coordenada normalizada zn podemos hallar la correspondiente coordenada cámara z_{cam} como:

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

Usad para f y n los valores zfar y znear (variables globales definidas en el código). Conocida z_{cam} las relaciones:

$$\frac{x_{cam}}{z_{cam}} = -\frac{x_n}{P_{00}} \qquad \frac{y_{cam}}{z_{cam}} = -\frac{y_n}{P_{11}}$$

nos permiten hallar x_{cam} e y_{cam} . P_{00} y P_{11} son los dos primeros elementos de la diagonal de la matriz de proyección (Proy[0][0] y Proy[1][1] en nuestro programa). Añadid un nuevo fprintf para volcar las tres coordenadas cámara (x_{cam}, y_{cam}, z_{cam}) con 3 decimales.

```
float zcam = - ((-2*zfar*znormear)/((zfar+znormear)+(znorm*(-zfar+znormear))));

float xcam = -(xnorm*zcam/Proy[0][0]);
float ycam = -(ynorm*zcam/Proy[1][1]);

fprintf(stdout, "xcam= %.3f ycam= %.3f zcam= %.3f\n", xcam, ycam, zcam);
```

Para comprobad si estas coordenadas son correctas, acercar el cubo a la posición más cercana posible y orientarlo de forma que se vea una cara de frente (como un cuadrado). Comprobad que la coordenada z_{cam} en todos los puntos donde pinchéis de la cara mostrada es similar ya que todos los puntos están más o menos en un plano a la misma distancia z de la cámara. ¿Qué valor os da z_{cam}? Explicad por qué obtenemos justamente ese valor.

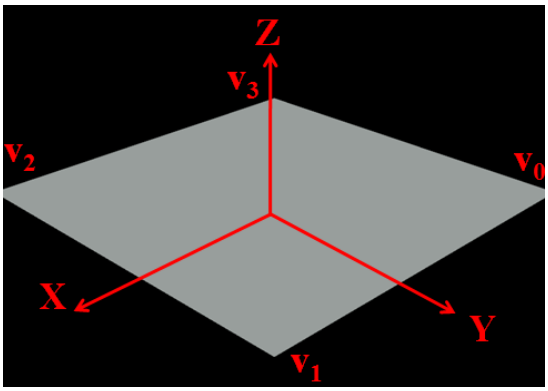
Si hacemos scroll al maximo y nos ponemos a la distancia mas cercana de d=4, como el cubo esta situado en d=1, al hacer click derecho en el raton la zcam nos da ~ 3. Que es la diferencia entre nuestra posicion y la del cubo.

Si (desde esa misma perspectiva) pinchamos en las esquinas del cubo las coordenadas x_{cam}, y_{cam} deben ser aproximadamente +/-1. Valores fuera del objeto deben dar una coordenada zcam = -25 (correspondiente al plano lejano).

Bloque 2: Comenzando con texturas.

Objetivos: En este bloque se comenzará con el uso de texturas en OpenGL. El iobjetivo es entender el concepto de coordenadas de textura y como usarlas dentro de los "shaders" para recubrir un modelo3D.
Usaremos algunas funciones auxiliares para leer imágenes o cargar los vértices y atributos de un modelo 3D desde un fichero (estas funciones están en el fichero auxiliar GpO_aux.cpp). Los datos de imágenes y modelos 3D los pondremos en el directorio ./data. En el código usamos p.e. **`cargar_textura("./data/foto0.jpg", GL_TEXTURE0);`** para cargar una textura, ya que el programa se ejecuta desde donde se crea el ejecutable. Si el lanzamiento del programa problemas por no encontrar las correspondientes texturas, revisad las rutas y también desde donde se lanza el binario, en Visual Estudio es muy común que se lance desde un directorio Debug o Release.

Ejercicio 1: Sobreescribid el código GpO_03.cpp y GpO_03_entrega.cpp con los nuevos ficheros suministrados. Sobreescribe también el directorio data con el nuevo contenido suministrado. En la función **`crea_cuadrado()`** especificamos los datos de un cuadrado (4 vert / 2 triang). En este caso, cada vértice, además de sus coordenadas 3D (X,Y,Z), tiene como atributo adicional dos coordenadas 2D de textura (u,v).



En **`init_scene()`** se carga la imagen contenida en el fichero "foto0.jpg" y se le asigna al primer slot o unit (GL_TEXTURE0) de las texturas activas en la GPU. En el procesador de vértices el atributo con las coordenadas textura (UV) se pasa sin modificar al procesador de fragmentos donde se usa para muestrear la textura y asignar el color del fragmento: **`col= texture(unit, UV).rgb;`**

Al ejecutarlo veréis que la foto no aparece, porque las coordenadas textura (u,v) que se dan como atributos de cada vértice en la función crear_cuadrado() se han inicializado todas al valor (0.5, 0.5). Darles los valores correctos tal como hemos visto para que foto aparezca sobre el cuadrado con su parte inferior en los vértices v2 y v1 y su parte superior en los vértices v3 y v0.

Una vez asignados valores adecuados a las coordenadas (u,v) la imagen de la fotografía aparecerá sobre el cuadrado. Haced girar el cuadrado (usando una matriz de rotación) alrededor del eje Z a una velocidad de 5º por segundo. Una vez que tenemos la textura "pegada" al cuadrado, ésta acompañará al objeto y no hay que preocuparse en hacer nada más, da igual lo complicado que sea nuestro modelo o el movimiento que le asignemos.

En la inicialización de la escena cargar una 2ª textura (del fichero "foto1.jpg"), y asociadla al slot 1. Luego haced que el programa muestre una u otra textura al pulsar la tecla F1. El "fragment shader" usa la textura especificada en "unit", por lo que habrá que cambiar su valor que por defecto es 0 (=slot 0) a 1 (= usar slot1). Podemos transferir un valor de 0/1 a la variable "unit" (declarada como uniform en GLSL) usando **`transfer_int("unit",valor)`**

Esta función (definida en GpO_aux) es similar a otras funciones transfer_xxx que hemos usado para transferir datos (float, mat4, ...) entre la aplicación y la GPU. Adjuntar las modificaciones del código indicando dónde las habéis hecho. Adjuntad captura de pantalla con la segunda imagen.

Vamos ahora a cambiar el cuadrado por un objeto 3D más complicado (una esfera). El nuevo modelo tiene un número elevado de vértices (256 vértices y 450 triángulos) por lo que no es conveniente introducirlos manualmente en vertex_data[] e indices[] como hemos hecho hasta ahora. En lugar de eso los cargaremos de un fichero con: **`obj=cargar_modelo("./data/esfera_256.bix")`**

La función lee los datos de los vértices e índices del fichero "esfera_256.bix". Al igual que antes, cada vértice lleva 5 datos, agrupados en 2 atributos (la posición XYZ y las coordenadas UV de la textura). Sustituir la creación del cuadrado por la carga del modelo de la esfera al inicializar la escena. La foto debe aparecer ahora sobre la esfera. Adjuntad una captura de la imagen. A esta escala se debe apreciar las facetas de la supuesta "esfera".

Obviamente no tiene mucho sentido superponer este tipo de fotos sobre una esfera. Cambiadlas por las imágenes de tierra.jpg y luna.jpg. Añadid también las texturas de marte.jpg y jupiter.jpg en los "slots" 2 y 3. Haced ahora que cada vez que se pulse F1 mostremos la siguiente imagen (al terminar volver a empezar con la Tierra). Adjuntar código de la función de teclado y una captura mostrando a Júpiter.

Para dar más realismo a la visualización podemos considerar la escala relativa de los planetas según la siguiente tabla:

Cuerpo	Tierra	Luna	Marte	Júpiter
Escala	1.0	0.27	0.53	11.0

Podéis declarar una variable global esc[4] para guardar estas escalas y en la función de refresco aplicar adicionalmente una matriz de escalado S (creada con la función scale) a los datos del modelo usando la escala correspondiente. Adjuntar captura de pantalla para el caso de Marte ¿Funciona correctamente en todos los casos? ¿Por qué?

Adjuntar código fuente final de este ejercicio como GpO_03_entrega_ejer1.cpp

Ejer2: Partir del código del ejercicio anterior y en vez de la esfera, cargar el modelo 3D del fichero spider.bix, junto con su textura asociada (spider.jpg).

Deberías ver una figura dando vueltas como si estuviera en una peana. El problema es que las coordenadas Z a lo largo de la altura de la figura van desde Z=0 a Z~2. Como nuestro target es ahora el origen estamos mirando a los pies y no vemos la figura al completo. Levantad la vista cambiando la coordenada Z del target a Z~0.9 o 1. Ahora debéis ver la figura al completo. Adjuntar captura del resultado. ¿Observáis algún problema al visualizar?
¿A qué es debido? Añadid el código necesario para solucionarlo.

Para apreciar la complejidad del modelo 3D podemos modificar la función de eventos de teclado para que al pulsar la letra F1 (en vez de cambiar de textura como antes) alternemos entre una visión de polígonos o de líneas. Para ello usaremos la función:

```
glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);  // Rasteriza solo líneas
glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);  // Rasteriza triángulos
```

Adjuntar una captura con la opción GL_LINE (en la que el rasterizador solo genera los fragmentos situados en los bordes del triángulo, no los interiores).

Cargad un 2º modelo con su correspondiente textura (halo.bix y halo.jpg) y dibujarlo usando para M la misma matriz de rotación que la figura anterior. Adjuntar una captura con las dos figuras. El problema es que el nuevo modelo tiene unas coordenadas propias donde el eje longitudinal a lo largo del cuerpo coincide con el eje Y en vez del eje Z (en ausencia del giro la figura estaría a lo largo del eje Y). Para arreglarlo podemos (antes de aplicar la matriz de giro que estamos usando ahora) aplicar un giro único de 90º alrededor del eje X para enderezar la figura. Recordad que si vamos a aplicar la matriz M_1 antes de M_2, la matriz compuesta a usar será $M_2 \cdot M_1$ (la matriz más a la derecha es la que se aplica en primer lugar). Adjuntad la línea de código para crear la matriz M usada y captura de la imagen resultante.

Ahora las figuras deben verse girando de la misma forma, pero el problema es que están colocadas en el mismo sitio. Para arreglarlo vamos a desplazar la primera figura a la posición (0.5,-0.5,0.0) y la segunda a (-0.5,0.5,0.0).Cread las correspondientes matrices de translación con translate() y recordad que la translación debe ser la última operación a aplicar (luego su matriz debe ser la

más a la izquierda en el producto de matrices). Al ejecutar el código ambas figuras deben aparecer girando como antes, pero con la 1ª figura en la parte izquierda de la ventana y la 2ª en la parte derecha.

Finalmente haced que al pulsar F2 se intercambien ambas texturas de forma que cada modelo tenga asociada la textura del otro. Adjuntar una captura de pantalla con las texturas intercambiadas.

Adjuntar código fuente del ejercicio como GpO_03_entrega_ejer2.cpp.