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

Objetivos del LAB:

- Generar el proyecto CMake según las instrucciones del readme adjunto
- Compilarlo usando el compilador elegido Eg. VisualStudio, GCC etc.
- Familiarizarse con las librerías auxiliares **GLFW** (creación de ventanas y manejo de eventos) y **GLM** (para crear/operar con matrices).
- Identificar la funcionalidad de las distintas partes del código: creación de una ventana, inicialización de la escena, refresco de pantalla, etc.
- Entender la descripción de una escena, disposición ejes, etc. en OpenGL, así como la influencia de los parámetros de la matriz de perspectiva.
- Combinar el uso de variables "uniform" con las rutinas de manejo de eventos para mover objetos por la escena o cambiar observador.
- Usar una descripción de índices para describir y dibujar un objeto.
- Identificar y modificar los programas o *shaders* de la GPU.

Ejer 1: Descargar de Moodle los contenidos del enlace "Proyecto base cauce gráfico programable (CMake)". Descomprimir el fichero GpO_Project.zip. En el directorio raíz encontraréis un proyecto C configurado mediante cmake, seguid las instrucciones del readme adjunto para su compilación. Tras compilarlo, ejecutarlo y veréis dos ventanas: la consola por donde salen los volcados (printf y similares) que se hacen desde el programa corriendo en la CPU y una ventana donde aparecen los gráficos que genera la GPU con los comandos OpenGL. En esta ventana debéis ver un triángulo de colores moviéndose arriba y abajo por la ventana.

Para hacer las modificaciones pedidas, trabajad en el fichero GpO_01_entrega.cpp para que podáis mantener una copia del fichero original.

1. Dentro del fichero fuente, buscad la función `render_scene`, encargada de refrescar la pantalla (en vuestro entorno, probablemente podéis pinchar en el desplegable del fichero GpO_01.cpp en el explorador del proyecto para ver un listado de sus funciones). Buscar en **`render_scene`** la llamada a la función **`perspective`**() de GLM que crea la matriz de proyección usada. La relación de aspecto y el campo de visión vertical se definen previamente en dos variables globales (aspect=4/3 y fov=40º) justo antes de render_scene(). Correr el programa y cambiar el tamaño de la ventana (pinchando en su borde superior y arrastrando con el ratón) para que ocupe todo el alto de la pantalla. El triángulo aparecerá ahora deformado al haber cambiado la relación de aspecto (ancho/alto) de la ventana sin cambiar la relación de aspecto (4/3) definida en la matriz de perspectiva P.

   Usar la función ResizeCallback (que está asignada para que se ejecute cuando el usuario cambia el tamaño de la ventana) para recalcular la

nueva relación de aspecto de la ventana (usando las variables ANCHO y ALTO). Actualizar con ese valor la variable global **aspect** usada al crear la matriz de proyección P. Para que la división se realice correctamente, haced previamente un casting a float de las variables ANCHO y ALTO (definidas como enteros). Si lo hacéis correctamente veréis como el triángulo no se deforma al cambiar la relación de aspecto de la ventana. Adjuntar el código añadido dentro de **`ResizeCallback`** (1 línea).
aspect = (float)ANCHO / (float)ALTO;

2. El movimiento del triángulo es tal que hace que se salga un poco del campo visual por arriba y por abajo. Al hacer la ventana más alta esto no se arregla: el triángulo se ve más grande pero se sigue saliendo. La razón es que la matriz P se crea siempre con el mismo campo visual (definido por la variable **fov**). Si queremos que al aumentar el alto de la ventana se vean zonas que antes no veíamos (en vez de verlo todo más grande), se necesita ampliar el campo visual vertical. Para ello podemos recalcular la variable **fov** (campo visual vertical en grados) cada vez que se cambien las dimensiones de la ventana, haciendo **fov** proporcional al ALTO de la ventana. Recordad que la ventana original tenía 600 píxeles de alto y mostraba un campo visual de 40º. Adjuntar código añadido.
fov = (40.0f / 600.0f) * ALTO;

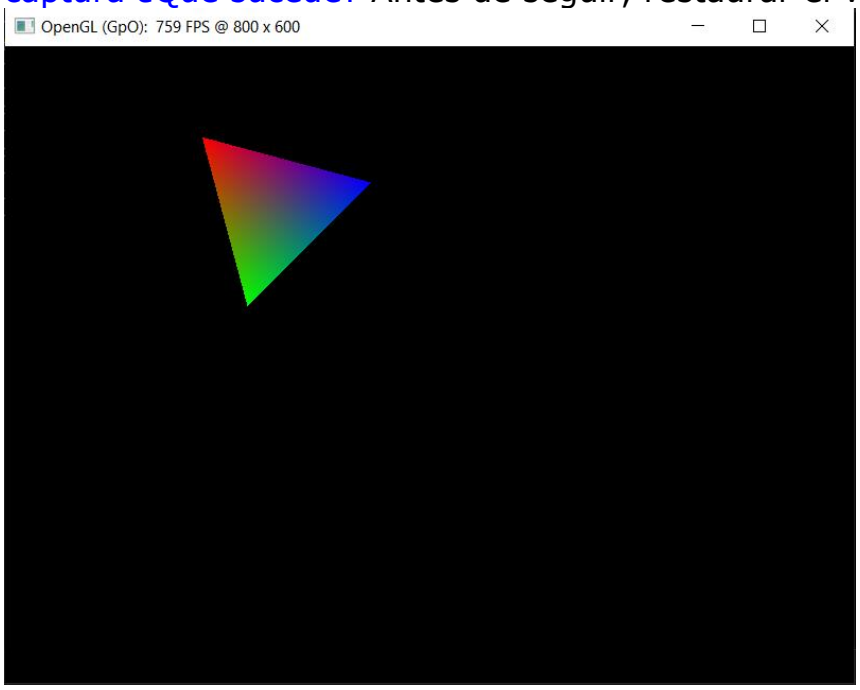
3. La función show_info() muestra (en el título de la ventana) la frecuencia con la que se ejecuta la función de refresco de pantalla (render_scene). Lo normal es un valor del orden de 60 fps ya que por defecto GLFW ajusta el refresco del framebuffer al de la pantalla (no tiene sentido actualizar el framebuffer 500 veces/sec si solo 60 de esos framebuffers llegan a transferirse a la pantalla (que se actualiza 60 veces/sec).

   En la función main() antes de entrar en el bucle principal, haced una llamada a **`glfwSwapInterval(0);`** Con esta opción GLFW ignora el valor de refresco de la pantalla y actualiza el framebuffer tan rápido como puede. ¿Cuántos FPS se indican ahora en la ventana? 1300 FPS Dependiendo del ordenador es posible que sigáis viendo el valor más bajo de antes porque en las opciones de la tarjeta gráfica podéis tener marcada la opción de que se mantenga siempre dicha sincronización. En mi ordenador puedo acceder a las opciones de la tarjeta pinchando (botón derecho) sobre la pantalla. Dentro de esas opciones buscad Vert Sync y seleccionar la opción de que sea controlada por la aplicación.

Ejer 2: Partir del código con las modificaciones añadidas del ejercicio anterior.

1) En render_scene(), buscar el código que crea la matriz del punto de vista (V) y la matriz de translación (T) del modelo. En un papel, haced un esquema indicando la posición de los ejes en la escena, la posición del observador, la dirección del vector up y el movimiento del objeto

sobre esos ejes. Cambiar en el código el vector up por (0,1,1). Adjuntad captura ¿Qué sucede? Antes de seguir, restaurar el vector up inicial.



El triágnulo se mueve en diagonal

2) Queremos usar los cursores Up y Down para cambiar la posición del observador sobre el eje X: Up para acercar la cámara al origen y Down para alejarla. Usar un salto de 0.1 en coordenada X por cada pulsación de las teclas. ¿Qué variable del programa debéis cambiar? Recordad que con la librería GLM podemos incrementar un vector en su conjunto con **`V+=vec3(0.1,0.0,0.0);`** o modificar sus componentes: **`V.x+=0.1`**;

Este código se pondrá en la función **`KeyCallback()`**, que ya está definida en nuestro programa. Haciendo luego **`glfwSetKeyCallback(KeyCallback)`** dentro de **`asigna_funciones_callback()`** dicha función `KeyCallback()` se ejecutará cuando haya un evento de teclado. Dentro de **`KeyCallback()`** el parámetro key nos dice la tecla pulsada para actuar en consecuencia. En el caso de los cursores UP/DOWN sus valores son GLFW_KEY_DOWN y GLFW_KEY_UP. Después de modificar la coordenada X del observador volcad su valor con printf() para saber donde está y verificar que se ha cambiado correctamente. Adjuntar código modificado de KeyCallback().

```
static void KeyCallback(GLFWwindow* window, int key, int code, int action, int mode)
{
	fprintf(stdout, "Key %d Code %d Act %d Mode %d\n", key, code, action, mode);
	if (key == GLFW_KEY_ESCAPE) glfwSetWindowShouldClose(window, true);


	if (action == GLFW_PRESS || action == GLFW_REPEAT) {
```

```
            if (key == GLFW_KEY_UP) {
                pos_obs.x -= 0.1f; // Acercar la cámara al origen
            }
            else if (key == GLFW_KEY_DOWN) {
                pos_obs.x += 0.1f; // Alejar la cámara del origen
            }

            fprintf(stdout, "Posición del observador: x = %f, y = %f, z =
%f\n", pos_obs.x, pos_obs.y, pos_obs.z);
        }


}
```

3) Tras añadir la funcionalidad anterior, correr el programa y acercaros al origen con el cursor UP. Poneros a X=0.8: el triángulo debe verse muy grande llenando vuestro campo visual al pasar por delante. Acercaros a X=0.4 del origen. ¿Qué ocurre entonces? ¿Por qué? Se deja de ver, porque ya hemos pasado su posición.

4) Buscar el código que causa el movimiento de translación del triángulo en la función render_scene(). Modificarlo para que el triángulo, en vez de subir y bajar en el eje Z, describa una trayectoria circular de radio 3 en el plano XY, usando X=3*cos(t), Y=3*sin(t), Z=0 para las componentes X,Y,Z del vector de translación. Adjuntar la línea de código modificado. ¿Qué se observa ahora si colocáis al observador en X=18? Justificadlo.

```
T = glm::translate(glm::vec3(3.0f * cos(t), 3.0f * sin(t), 0.0f));
```

Al colocar el observador en X=18 se deja de ver el triángulo en parte de su movimiento debido a que se sale del plano lejano

Ejer 3: Partir del código con las modificaciones de los 2 ejercicios anteriores. Tenemos un solo objeto (un triángulo) en la escena describiendo un círculo en el plano XY. Se trata ahora de dibujar un segundo objeto. Para no tener que crear y transferir los datos de un segundo objeto a la GPU vamos a pintar un segundo triángulo. De esta forma los datos ya los tenemos en la GPU y solo tenemos que volver a pintarlos. Para dibujar un 2º triángulo basta repetir la orden de dibujar (`glDrawArrays`) en la función de rendering. El problema es que si damos las dos órdenes de dibujo seguidas el segundo triángulo se pintará exactamente encima del primero y solo se verá uno de ellos. Lo que haremos es cambiar la matriz M (que es la encargada de posicionar el objeto en la escena) entre las dos órdenes de dibujo.

- Para la matriz M del primer triángulo usaremos la matriz de translación que tenemos ahora, describiendo un círculo en el plano XY.
- Para la M del segundo triángulo usad la matriz que teníamos en el código original (la que movía el objeto de arriba/abajo en el eje Z).

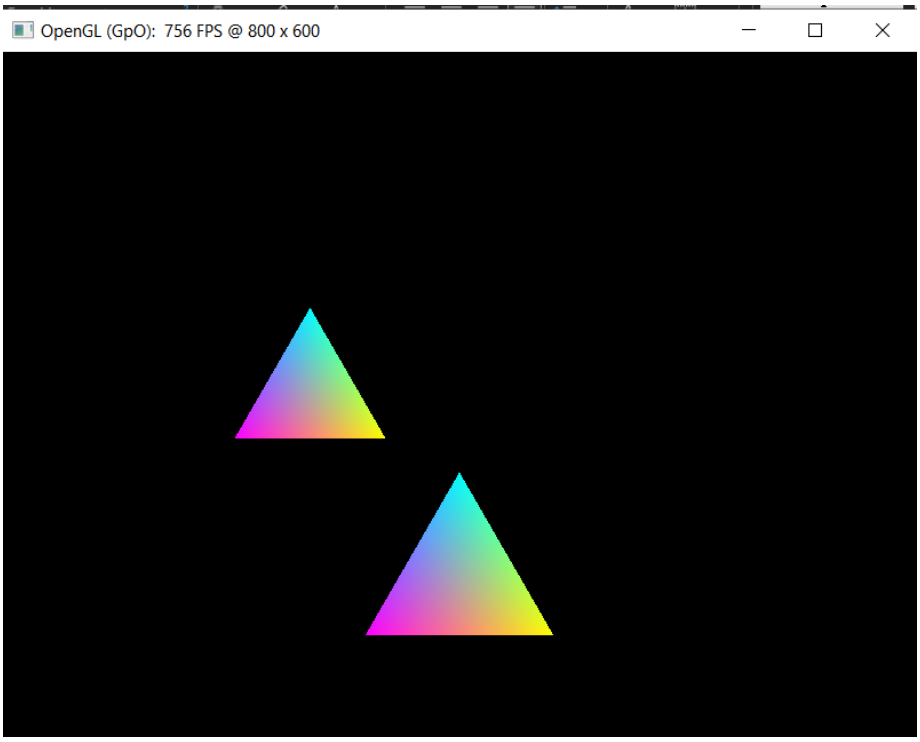
Adjuntar el código añadido a vuestra función de rendering (solo la parte de crear las matrices M y dibujar ambos objetos). Compilar y ver el resultado. ¿Apreciáis algo incorrecto en la evolución de la escena mostrada?

```
	mat4 P,V,M,T1,T2, R,S;

	P = perspective(glm::radians(fov), aspect, 0.5f, 20.0f);  //40º FOV, 4:3 , Znear=0.5, Zfar=20
	V = lookAt(pos_obs, target, up  );  // Pos camara, Lookat, head up

	//T = translate(0.0f, 0.0f, 3.0f*sin(t));
	T1 = glm::translate(glm::vec3(3.0f * cos(t), 3.0f * sin(t), 0.0f));
	M = T1;
	transfer_mat4("MVP",P*V*M);

	// ORDEN de dibujar
	glBindVertexArray(triangulo.VAO);		// Activamos VAO asociado al objeto
   glDrawArrays(GL_TRIANGLES, 0, triangulo.Nv);   // Orden de dibujar (Nv vertices)
	glBindVertexArray(0);			// Desconectamos VAO

	//////////////////////////////////////////////////////


// Segundo triángulo: movimiento original en Z
	T2 = glm::translate(glm::vec3(0.0, 0.0, 3.0f * sin(t)));
	M = T2;
	transfer_mat4("MVP", P * V * M);
	glBindVertexArray(triangulo.VAO);
	glDrawArrays(GL_TRIANGLES, 0, triangulo.Nv);
	glBindVertexArray(0);
```

Ejer 4: Buscad en el programa usado en el ejercicio anterior el código GLSL del procesador de vértices o "vertex shader". Lo encontraréis en una cadena de texto llamada **`vertex_prog`** definida al principio del programa. Eliminar el punto y coma al final de la línea `col=color;` lo que causa un error de sintaxis en el programa GLSL. Volver a compilar el programa. ¿Hay errores? ¿Por qué? Ejecutarlo. ¿Funciona? Volver a poner el ; y asignar ahora a col el valor de (1-color). Adjuntad captura.

En compilación no hay problema, porque el shader no se compila ahí, ya que se ejecuta en la GPU. Actualmente, los shaders se compilan en runtime, en una función denominada *init_scene*. Cuando se ejecuta el programa se ve el error devuelto por el compilador del shader.

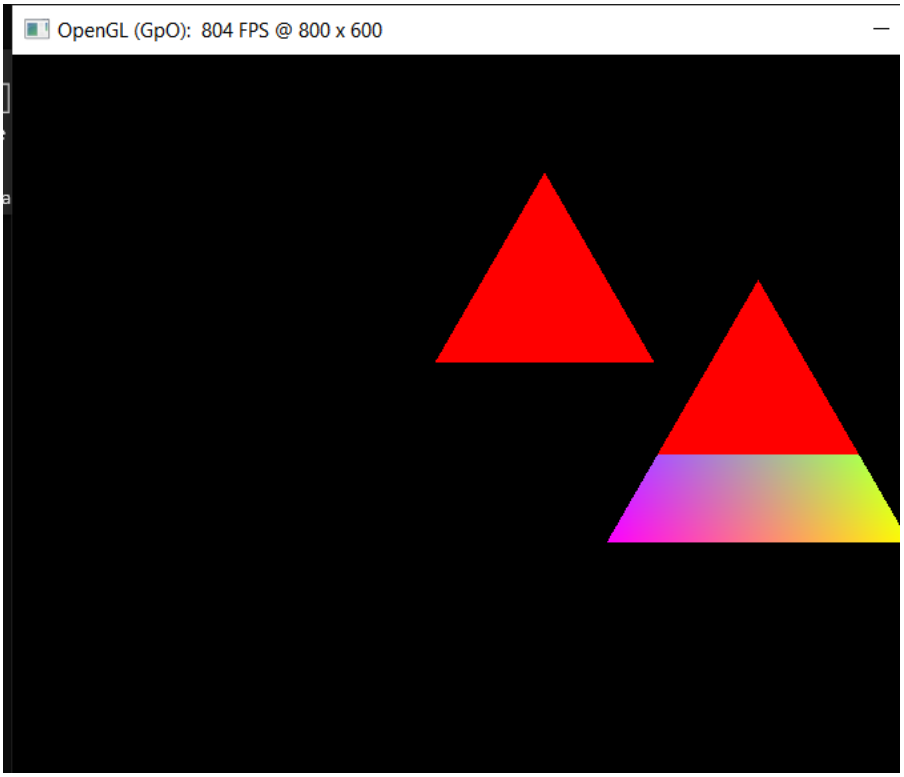
Al cambiar por col = (1-color) se ve que los colores de los triángulos han cambiado.



El código del procesador de fragmentos está justo después del código anterior dentro de una cadena llamada **fragment_prog**. Cuando se ejecuta el fragment shader en la GPU, la posición del fragmento en la ventana (en píxeles) es conocida y se puede acceder a ella a través de la variable gl_FragCoord.

Usar la 2ª componente de dicha variable (**gl_FragCoord.y**) para poner el color del fragmento de salida a rojo (1, 0, 0) si la posición Y del fragmento en la ventana es mayor que 300 (el alto original de la ventana eran 600 píxeles).
Adjuntar código del "fragment shader" modificado y una captura de la imagen resultante donde se observe el efecto del cambio de código. ¿Dónde está (arriba o abajo) el origen de la coordenada gl_FragCoord.y según OpenGL?

```
const char* fragment_prog = GLSL(
in vec3 col;
out vec3 outputColor;
void main()
 {
      if (gl_FragCoord.y > 300)
            outputColor = vec3(1.0, 0.0, 0.0); // Rojo
      else
            outputColor = col; // Color original
 }
);
```



El origen de la coordenada gl FragCoord.y según OpenGL está abajo.

Durante la ejecución del procesador de fragmentos cualquier fragmento puede descartarse (y no ser pintado) en el último momento usando la orden discard; Poner una condición para que se descarte el fragmento si su coordenada X (gl_FragCoord.x) es mayor que 400. Observar los resultados y adjuntad una captura de la pantalla.

A partir de la coordenada x 400 los fragmentos no se dibujan y entonces parte del primer triángulo no se pinta y durante parte del recorrido del segundo triángulo, éste no se pinta.

Ejer 5: Partir del **código original** (GpO_01.cpp), copiarlo a un nuevo fichero llamado GPO_01_index.cpp y usadlo como nuevo fichero fuente del proyecto. En algunos entornos como Visual Studio podéis hacerlo directamente dentro de del entorno, o lo podéis incorporar al fichero de cmake y regenerar el proyecto. Vamos a hacer algunos cambios en el código para dibujar usando índices en vez de usar los vértices directamente como se ha hecho hasta ahora.

La mayoría de los cambios van a hacerse en la función crear_triangulo(), que es donde se transfieren los datos de los vértices a la GPU y se organizan en un VAO. En este programa los atributos de posición y color de los vértices están en dos variables separadas (usando dos VBO's diferentes para su transferencia). Lo que vamos a hacer es:

- Declarad un nuevo array (index) de tipo GLubyte (unsigned bytes) y de tamaño 3 y llenadlo con los índices de los vértices que definen nuestra figura. Al ser un triángulo solo tenemos tres vértices que serán 0, 1, 2.
- Tras cerrar los dos VBO's usados para mandar los datos de los vértices (pos/color) y antes de "cerrar" definitivamente el VAO debemos incluir en el VAO la nueva lista de índices. Para ello usar los comandos que están en las transparencias (Comandos de dibujo: uso de índices) para:

  a) Pedir a OpenGL (`glGenBuffers`) un identificador de buffer.
  b) Enlazarlo (con `glBindBuffer`) como un buffer del tipo de los usados para índices: GL_ELEMENT_ARRAY_BUFFER.
  c) Usando `glBufferData` transferir a la GPU los datos del nuevo array de índices que habéis definido.

  Dejar enlazado (activo) el buffer de índices y cerrar el VAO.

- Finalmente, junto al comando **`obj.Nv=3;`** (donde se guardaba el número de vértices del objeto) añadir **`obj.Ni=3;`** indicando su número de índices. En este caso coinciden, pero en general no tienen por qué ser iguales (el número de vértices será menor que el de índices ya que se re-usarán).

Adjuntad el código de vuestra función crear_triangulo() modificada.

```
objeto obj;
GLuint VAO;
GLuint buffer_pos, buffer_col, EBO;

GLfloat pos_data[3][3] = { 0.0f,  0.0000f,  1.0f,  // Posición vértice 1
```

```
                                                0.0f, -0.8660f, -0.5f,   // Posición
vértice 2
                                                0.0f,  0.8660f, -0.5f }; // Posición
vértice 3

        GLfloat color_data[3][3] = { 1.0f, 0.0f, 0.0f,  // Color vértice 1
                                                    0.0f,  1.0f,  0.0f,    // Color
vértice 2
                                                    0.0f,  0.0f,  1.0f  };  // Color
vértice 3

        GLubyte index[3] = { 0, 1, 2 }; // Indices vertices

        // Generar y enlazar el VAO
        glGenVertexArrays(1, &VAO);
        glBindVertexArray(VAO);

        // Enviar posiciones en un VBO
        glGenBuffers(1, &buffer_pos);
        glBindBuffer(GL_ARRAY_BUFFER, buffer_pos);
        glBufferData(GL_ARRAY_BUFFER,       sizeof(pos_data),       pos_data,
GL_STATIC_DRAW);
        glEnableVertexAttribArray(0);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);

        // Enviar colores en otro VBO
        glGenBuffers(1, &buffer_col);
        glBindBuffer(GL_ARRAY_BUFFER, buffer_col);
        glBufferData(GL_ARRAY_BUFFER,      sizeof(color_data),      color_data,
GL_STATIC_DRAW);
        glEnableVertexAttribArray(1);
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);

        // Crear y enlazar el EBO
        glGenBuffers(1, &EBO);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,      sizeof(index),      index,
GL_STATIC_DRAW);

        // Desenlazar el VAO (el EBO permanece asociado al VAO)
        glBindVertexArray(0);

        obj.VAO = VAO;
        obj.Nv = 3; // Número de vértices
        obj.Ni = 3; // Número de índices

        return obj;
```

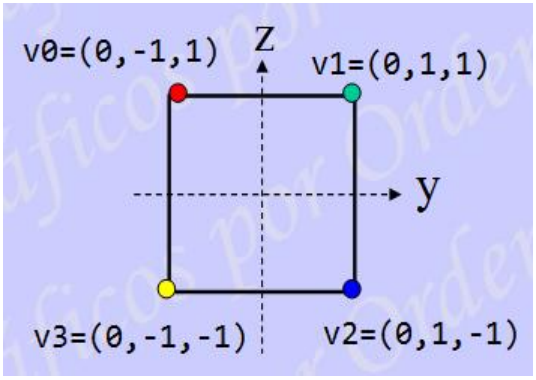
Luego, en render_scene() lo único que hay que hacer es cambiar la orden de dibujo. En vez de glDrawArrays() si queremos dibujar con índices usaremos:

```
glDrawElements(GL_TRIANGLES,triangulo.Ni,GL_UNSIGNED_BYTE,0);
```

El parámetro nuevo de glDrawElements indica el tipo de datos que usamos en el array de índices: **GL_UNSIGNED_BYTE,GL_UNSIGNED_SHORT** o **GL_UNSIGNED_INT.** Dependerá de si el array de índices se declaró como GLubyte (1 byte), GLushort (2 bytes) o GLuint (4) en el código de la aplicación. En nuestro caso, al haberlos declarados como bytes usaremos **GL_UNSIGNED_BYTE**. Al ejecutar el programa se debe ver el mismo triángulo subiendo y bajando.

En este caso no tiene sentido usar índices, porque para un triángulo no hay ningún ahorro (#índices = #vértices). Al usar figuras más complicadas donde varios triángulos comparten vértices sí que será más económico mandar la información de los vértices una sola vez y citarlos luego por sus índices.

Copiar y pegar la función crear_triangulo() en una nueva función (llamadla crear_cuadrado). Se trata de modificarla para que el objeto sea ahora un cuadrado contenido en el plano YZ, con vértices en las coordenadas indicadas en la figura adjunta: **v0=(0,-1, 1) v1=(0, 1, 1) v2=(0, 1,-1) v3=(0,-1,-1)**

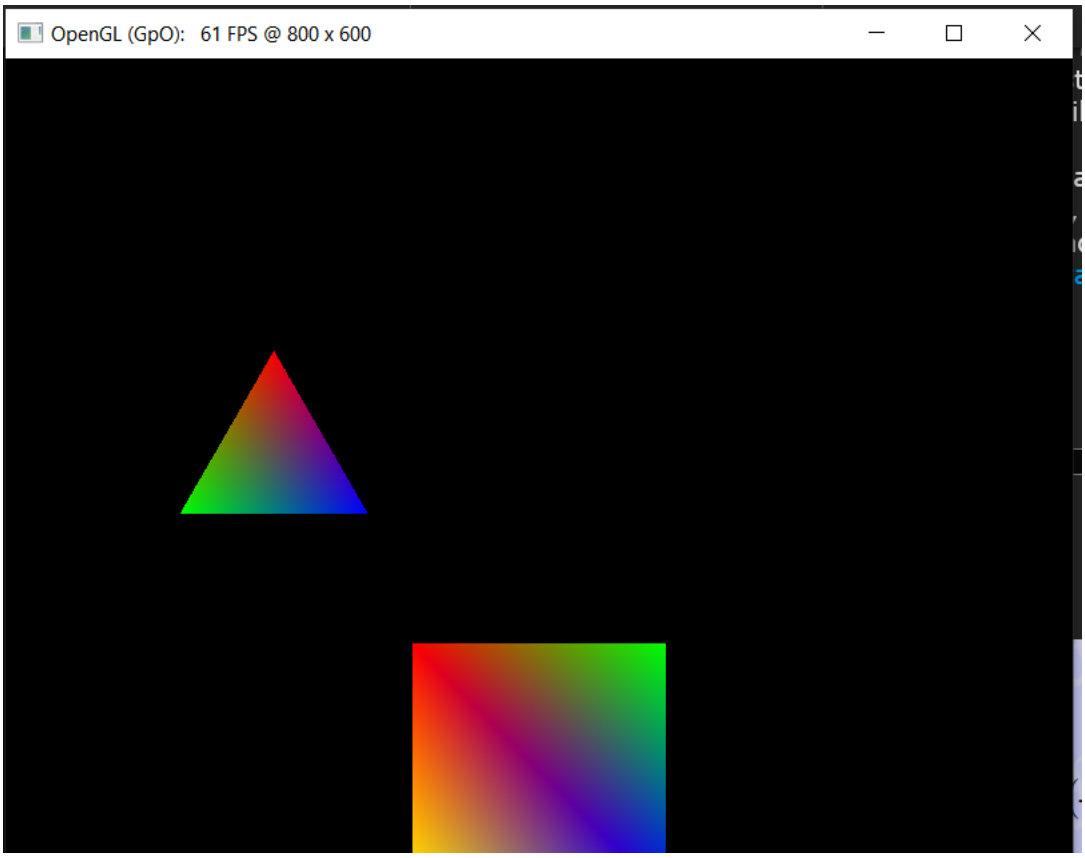


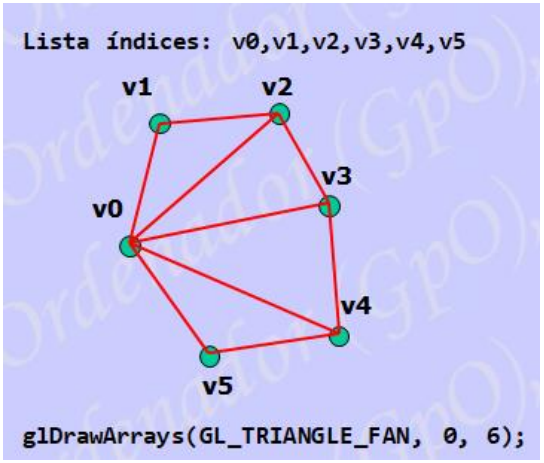
Modificar la definición de pos_data (ahora será un array de 4x3) usando las posiciones de los 4 vértices anteriores. Añadid también una línea adicional a color_data para asignar el color del 4ª vértice. Mantened los primeros vértices con los colores que tenían antes (rojo, verde y azul) y asignar el color amarillo (1, 1, 0) al 4º vértice. Adjuntad el nuevo código que asigna valores a pos_data y color_data.

¿Cuál sería un posible vector de índices describiendo el cuadrado? Adjuntadlo.
012023
¿Cuántos vértices e índices tiene ahora el nuevo objeto? Vértices: 4, índices: 6
Usadlos en el código para asignar valores adecuados a los camops obj.Nv y obj.Ni. Ejecutad el nuevo programa y comprobad que el objeto que se mueve es un cuadrado.

En este caso el vector de índices tendrá 6 índices (3+3) para describir los 2 triángulos que forman nuestro cuadrado. Esto es necesario porque la primitiva de la orden de dibujo (GL_TRIANGLE) describe cada triángulo por separado.

Añadir código en render_scene() para dibujar dos figuras siguiendo diferentes trayectorias como se hizo en el ejercicio 3, pero ahora haciendo que una de ellas sea un triángulo y la otra el cuadrado, en vez de ser las 2 iguales. Ejecutarlo y adjuntad una captura del resultado.





Ejer 6: Además de GL_TRIANGLE, OpenGL dispone de otras primitivas que en ciertos casos nos permiten describir una figura de una forma más económica.

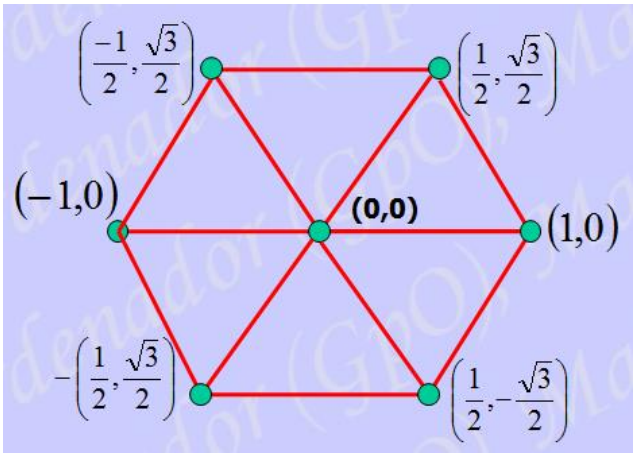
Una de estas primitivas es GL_TRIANGLE_FAN en la que el 1er índice de la lista corresponde a un vértice que se re-usa en todos los demás triángulos. En este caso, los tres primeros índices forman un triángulo y cada triángulo adicional se forma con el primer índice de la lista, el último índice del triángulo anterior y el siguiente índice. En la figura adjunta, para la lista de índices dada: v0,v1,v2,v3,v4,v5, los 4 triángulos dibujados serían:

(v0,v1,v2) (v0,v2,v3) (v0,v3,v4) (v0,v4,v5)

De esta forma esta figura puede describirse con una lista con sólo 6 índices, en vez de tener que usar 12 (v0,v1,v2, v0,v2,v3, v0,v3,v4, v0,v4,v5), que sería el caso si se usa la primitiva habitual GL_TRIANGLE.

Modificar el vector de índices usando esta nueva técnica para describir el cuadrado con sólo 4 índices. Recordad usar la primitiva GL_TRIANGLE_FAN al dar la orden de dibujar. Adjuntad líneas de código modificado.

```
GLubyte index[6] = { 0, 1, 2, 3 };

glDrawElements(GL_TRIANGLE_FAN, cuadrado.Ni, GL_UNSIGNED_BYTE,
0);
```



Partiendo del código anterior (usando la primitiva GL_TRIANGLES_FAN) se trata de modificarlo para que dibuje un hexágono en vez de un cuadrado.

Para construir el hexágono usaremos los siete vértices con las coordenadas que se muestran en la figura. Respecto a los colores usaremos el blanco (1,1,1) en el vértice central y para los otros vértices usaremos rojo (1,0,0), amarillo (1,1,0), verde (0,1,0), cian (0,1,1), azul (0,0,1) y magenta (1,0,1). Dad un listado de los de índices que representarían la figura.

```
GLubyte index[8] = {
        0, 1, 2, 3, 4, 5, 6, 1
};
```

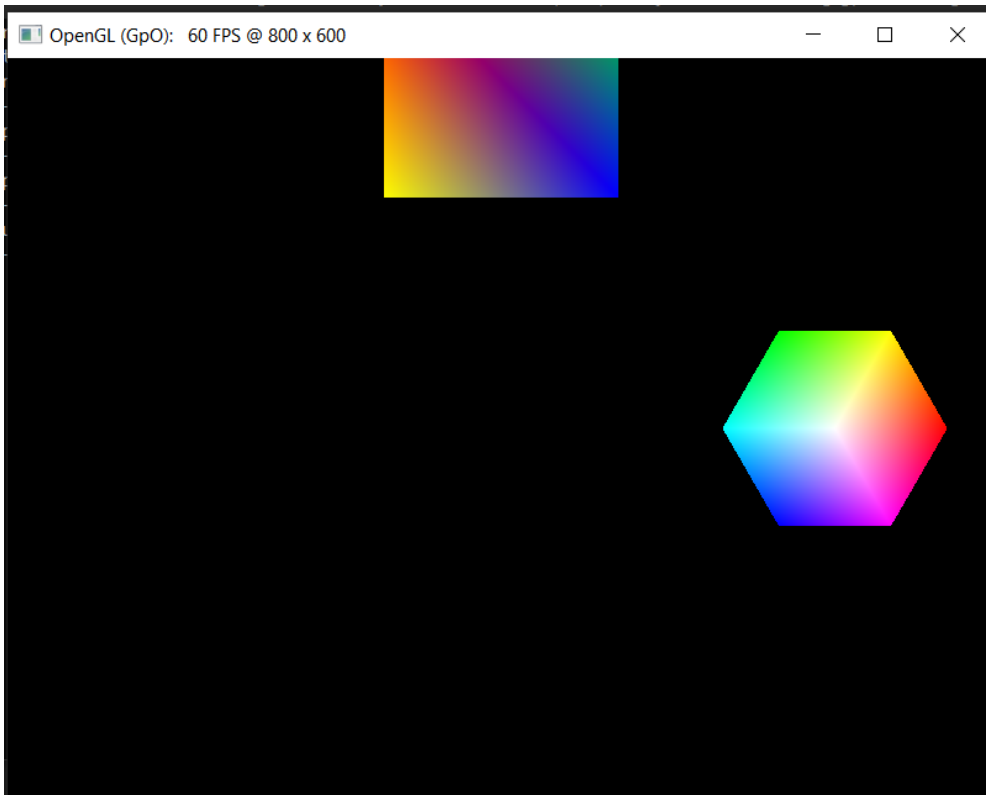
Recordad que el vértice en (0,0) debe ser el 1er índice al ser usado por todos los triángulos de la figura.

Adjuntad el código de la función crear_hex() y una captura mostrando las dos figuras (hexágono y cuadrado) moviéndose por la pantalla.

```
objeto crear_hex(void)
{
        objeto obj;
        GLuint VAO;
        GLuint buffer_pos, buffer_col, buffer_idx;

        GLfloat pos_data[7][3] = {
                {0.0f, 0.0f, 0.0f},  // v0 (centro)
                {0.0f, 1.0f, 0.0f},  // v1
                {0.0f, 0.5f, 0.866f},  // v2
                {0.0f, -0.5f, 0.866f},  // v3
```

```
            {0.0f, -1.0f, 0.0f},  // v4
            {0.0f, -0.5f, -0.866f},  // v5
            {0.0f, 0.5f, -0.866f}   // v6
      };

      GLfloat color_data[7][3] = {
            {1.0f, 1.0f, 1.0f},  // v0 (blanco)
            {1.0f, 0.0f, 0.0f},  // v1 (rojo)
            {1.0f, 1.0f, 0.0f},  // v2 (amarillo)
            {0.0f, 1.0f, 0.0f},  // v3 (verde)
            {0.0f, 1.0f, 1.0f},  // v4 (cian)
            {0.0f, 0.0f, 1.0f},  // v5 (azul)
            {1.0f, 0.0f, 1.0f}   // v6 (magenta)
      };

      GLubyte index[8] = {
            0, 1, 2, 3, 4, 5, 6, 1
      };

      // Crear y enlazar el VAO
      glGenVertexArrays(1, &VAO);
      glBindVertexArray(VAO);

      // Crear y enlazar el VBO para posiciones
      glGenBuffers(1, &buffer_pos);
      glBindBuffer(GL_ARRAY_BUFFER, buffer_pos);
      glBufferData(GL_ARRAY_BUFFER,       sizeof(pos_data),       pos_data,
GL_STATIC_DRAW);
      glEnableVertexAttribArray(0);
      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);

      // Crear y enlazar el VBO para colores
      glGenBuffers(1, &buffer_col);
      glBindBuffer(GL_ARRAY_BUFFER, buffer_col);
      glBufferData(GL_ARRAY_BUFFER,      sizeof(color_data),      color_data,
GL_STATIC_DRAW);
      glEnableVertexAttribArray(1);
      glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), 0);

      // Crear y enlazar el EBO para índices
      glGenBuffers(1, &buffer_idx);
      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer_idx);
      glBufferData(GL_ELEMENT_ARRAY_BUFFER,      sizeof(index),      index,
GL_STATIC_DRAW);

      // Desenlazar el VAO
      glBindVertexArray(0);
```

```
        obj.VAO = VAO;
        obj.Nv = 7;  // Número de vértices
        obj.Ni = 12;  // Número de índices

        return obj;
}
```



Carga del código de los "shaders" desde un fichero.

Por comodidad en este curso el código de los shaders (en lenguaje GLSL) lo incluiremos como unas cadenas de texto dentro de nuestra aplicación. De esta forma al hacer entregas no tendremos que adjuntad aparte dichos programas.

En este apartado veremos otra opción (más adecuada en una aplicación más "compleja"). El código de ambos shaders se escribirá en dos ficheros distintos (por ejemplo "prog.vs" y "prog.fs"). Dichos códigos los leeremos luego desde la aplicación antes de compilarlos. Se trata de:

- Crear un fichero "prog.vs" conteniendo el código del vertex shader:

```
#version 330 core
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 color;
out vec3 col;
uniform mat4 MVP=mat4(1.0f);
void main()
{
  gl_Position = MVP*vec4(pos,1);
```

```
    col = color;
}
```

- Crear otro fichero "prog.fs" con el código del fragment shader:

```
#version 330 core
in vec3 col;
void main()
 {
  gl_FragColor = vec4(col,1);
 }
```

Para no tener problemas para encontrarlos aseguraros de que ambos ficheros están en el directorio de ejecución en el directorio data. Dentro del programa comentad el código inicial (ya no necesario) donde se definían las cadenas de texto vertex_prog y fragment_prog. Luego añadid este código en la función de inicialización de la escena (antes de compilar los shaders):

```
char* vertex_prog = leer_codigo_de_fichero("prog.vs");
char* fragment_prog = leer_codigo_de_fichero("prog.fs");
```

De esta forma cargamos en las mismas cadenas de texto el código leído de los correspondientes ficheros usando la función **leer_codigo_de_fichero** (definida en GpOaux.cpp).

Al ejecutar el programa debéis ver los mismos resultados, ya que el código de los shaders es el mismo que teníamos. La ventaja es que ahora podemos introducir cambios (probad con alguno de los cambios del ejercicio 4) y ver su resultado volviendo a correr la aplicación, sin tener que recompilarla.

Si queremos una interacción más dinámica al modificar los "shaders" podemos optar por hacer un "hot-loading". Se trata de cambiar y recompilar el código del shader (p.e. al pulsar una cierta tecla) MIENTRAS la aplicación principal se está ejecutándo. De esta forma podremos ver los cambios inmediatamente, sin tener que cerrar y volver a lanzar la aplicación.

Para hacer esto podemos usar la función que maneja las interrupciones de teclado (como hicimos con los cursores). Al pulsar cierta tecla, volveríamos a leer los shaders de los ficheros prog.vs y prog.fs y procederíamos a recompilarlos. Utilizad por ejemplo la tecla F1, cuyo código es **GLFW_KEY_F1**. Adjuntad el código añadido a la función "CallBack" de teclado.

```
// Callback de pulsacion de tecla
static void KeyCallback(GLFWwindow* window, int key, int code, int action, int mode)
{
	fprintf(stdout, "Key %d Code %d Act %d Mode %d\n", key, code, action, mode);
```

```
	if (key == GLFW_KEY_ESCAPE) glfwSetWindowShouldClose(window, true);

	if (key == GLFW_KEY_F1 && action == GLFW_PRESS) {
		printf("Recargando shaders...\n");

		// Leer los shaders desde los archivos
		char* vertex_prog = leer_codigo_de_fichero("data/prog.vs");
		char* fragment_prog = leer_codigo_de_fichero("data/prog.fs");

		// Compilar los nuevos shaders
		GLuint VertexShaderID = compilar_shader(vertex_prog, GL_VERTEX_SHADER);
		GLuint FragmentShaderID = compilar_shader(fragment_prog, GL_FRAGMENT_SHADER);

		// Reenlazar los shaders al programa
		glDeleteProgram(prog);  // Eliminar el programa anterior
		prog = glCreateProgram();
		glAttachShader(prog, VertexShaderID);
		glAttachShader(prog, FragmentShaderID);
		glLinkProgram(prog);
		check_errores_programa(prog);

		// Limpieza final de los shaders una vez compilado el programa
		glDetachShader(prog, VertexShaderID);
		glDeleteShader(VertexShaderID);
		glDetachShader(prog, FragmentShaderID);
		glDeleteShader(FragmentShaderID);

		glUseProgram(prog);  // Usar el programa actualizado
	}
}
```

Probad a modificar un shader y ver los resultados inmediatamente al pulsar F1 sobre la ventana de la aplicación.

En este caso podemos tener un problema si cuando modificamos los shaders cometemos algún error (p.e. de sintaxis). En ese caso la ventana se cerrará y tendremos que corregir el shader y volver a lanzar la aplicación. Esto sucede porque en las rutinas que escribí en GpO_aux.cpp no anticipé este posible uso dinámico y si se detecta un error en los shaders se cierra la ventana de la aplicación (ya que en el modo estático si el shader era incorrecto la aplicación no iba a poder dibujar nada por pantalla).

Para implementarlo correctamente deberíamos tener más control sobre lo que pasa en el caso de tener errores al compilar los shaders. Por ejemplo

podríamos volcar un mensaje de aviso del error cometido y no dar el control al nuevo programa, manteniendo el shader anterior que estaba funcionando. Así, ante un error, la aplicación nos avisaría y se podría volver a modificar el código de los programas en el fichero correspondiente. Al recargarlos (en caso de haber corregido los errores) veríamos los resultados del nuevo shader.

```
	// Compilear Shaders
	GLuint VertexShaderID = compilar_shader(vertex_prog, GL_VERTEX_SHADER);
	GLuint FragmentShaderID = compilar_shader(fragment_prog, GL_FRAGMENT_SHADER);

	GLint success;
	glGetShaderiv(VertexShaderID, GL_COMPILE_STATUS, &success);
	if (!success) {

		fprintf(stderr, "Error compiling vertex shader. Shader not updated.\n");
		return;  // No se continúa con el proceso de actualización del shader
	}

	glGetShaderiv(FragmentShaderID, GL_COMPILE_STATUS, &success);
	if (!success) {
		fprintf(stderr, "Error compiling fragment shader. Shader not updated.\n");
		return;  // No se continúa con el proceso de actualización del shader
	}
```