Apellidos, Nombre:
Apellidos, Nombre:

Responder a las preguntas y adjuntar las gráficas y "trozos" de código pedidos. Entregad este fichero con las respuestas junto con los ficheros fuente pedidos en un .rar, .7s o similar.

Objetivos:

- Conocer entorno de VisualStudio y las librerías auxiliares freeglut para manejo de ventanas/eventos y glm para crear/operar con matrices.
- Identificar que partes del código deben ejecutarse una vez (inicialización escena) y cuáles deben ejecutarse cada vez que regeneramos un frame.
- Entender la descripción de una escena, disposición ejes, etc. en OpenGL Combinar el uso de variables “uniform” con las rutinas de manejo de eventos para mover objetos por la escena o cambiar el punto de vista.
- Cambiar la posición/orientación de la cámara con el teclado y observar los resultados.

Partiremos del programa suministrado "GpO_01.cpp". Incorporarlo a vuestro proyecto, compilarlo y ejecutar. Veréis un triángulo moviéndose arriba y abajo en la pantalla. Para hacer las modificaciones pedidas, haced primero una copia del fichero dado y cambiar el fichero fuente del proyecto al nuevo fichero.

LAB 1:

1) Buscar la función render_scene() que se ocupa de refrescar la escena. Cambiar el color negro del fondo por un gris oscuro rgb=(0.2,0.2,0.2) en la asignación glCrearColor(). ¿Qué sucede si comentáis la orden glClear() siguiente? Adjuntar una captura de la pantalla.

2) Buscar en render_scene la llamada a la función perspective de GLM que crea la matriz de proyección P usada. Indicar la relación de aspecto y campo de visión vertical usado en dicha matriz de proyección.

3) Cambiar el tamaño de la ventana en una dimensión y observar lo que le pasa al triángulo. La razón es que estamos cambiando la relación de aspecto de la ventana (ancho/alto) pero no la que se indicó a la matriz de proyección con la variable aspect . Usar la función cambia_ventana (que está asignada para ejecutarse cada vez que se cambia el tamaño de la ventana) para calcular su nueva relación de aspecto (usando CurrentWidth y CurrentHeight) y actualizar con ese valor la variable aspect (usada al crear la matriz P). Observar como ahora el triángulo no se deforma aunque cambiemos la relación de aspecto de la ventana.

   Adjuntar código añadido (1 línea).

4) El movimiento del triángulo hace que se salga del campo visual un poco por arriba y por abajo. Si intentamos hacer la ventana más alta no se arregla, sólo vemos el triángulo más grande, porque se sigue manteniendo el mismo campo visual (especificado por la variable global fov al crear P). Si lo que queremos al aumentar el alto de la ventana es ver zonas que antes no veíamos, necesitamos ampliar el campo visual vertical al hacerlo. Para ello recalcular la variable fov (campo visual en º) cada vez que se cambien las dimensiones de la ventana. Hacer fov proporcional al alto (CurrentHeight) de la ventana, recordando que la ventana original tenía 450 de alto y mostraba un campo visual de 35º.
Adjuntar código añadido (1 línea)

5) Usar la rueda del ratón (o en su defecto, los botones derecho e izquierdo) para aumentar/disminuir el tamaño vertical de la ventana. Usar la función mouse() ya definida en el código y rellenarla para que aumente o disminuya el tamaño vertical de la ventana en 10 píxeles cada vez que se gire la rueda o se haga click en el ratón. Escribir primero un printf() que vuelque los valores de botón y estado para ver como varían y usarlos para cambiar el tamaño de la ventana. Al cambiar el tamaño de la ventana modificar el valor de CurrentHeight que guarda el alto de la ventana en ese momento. Tras actualizar su valor cambiar el tamaño con una llamada a glutReshapeWindow().

Recordad que además de rellenar la función mouse() debéis asignarla a los eventos de ratón usando glutMouseFunc(mouse);

Adjuntar vuestro código de la función mouse()

LAB 2: Partir del código incorporando las modificaciones del ejercicio anterior.

1) Observar el código para crear la matriz de cambio de punto de vista V y la matriz de translación T del modelo en render_scene(). Hacer un esquema indicando la posición de los ejes en la escena, la posición del observador y el movimiento del objeto sobre esos ejes.
Scannear o hacer una foto de vuestro esquema y adjuntarla.

2) Usando las teclas especiales de los cursores (códigos GLUT_KEY_UP y GLUT_KEY_DOWN) cambiar la posición del observador a lo largo del eje X: pulsando UP la cámara debe acercarse al origen y pulsando DOWN alejarse. Usad un incremento/decremento de 0.1 en la coordenada X.
¿Qué variable debéis cambiar? ¿En qué función debéis hacerlo?
Recordad que con la librería GLM podéis incrementar/decrementar un vector haciendo simplemente v += vec3(, ,).
En ambos casos cada vez que se pulse la tecla tras cambiar la posición, volcar por consola la posición de la coordenada X del observador.
Adjuntar el código añadido a vuestro programa (2/3 líneas).

3) Una vez codificada la funcionalidad anterior, correr el programa y pulsar el cursor UP para acercaros. El triángulo se hace más grande, llenando vuestro campo visual cuando pasa por delante.
Poneros a X=0.7 y luego a X=0.3 del origen. ¿Qué ocurre? ¿Por qué?
Ahora alejaros. Observar como el objeto se va reduciendo.
¿Notáis algo extraño si os alejáis mucho? ¿A qué distancia del origen estabais? JUSTIFICAR.

4) Buscar el código que causa el movimiento del triángulo en la función render_scene(). Modificarlo para que el triángulo, en vez de subir/bajar en el eje Z, describa una trayectoria circular de radio 3 en el plano XY.

Adjuntar código añadido (basta con modificar una línea).
¿Cambia el tamaño del objeto? ¿Por qué si no se cambia su escala?
Acercaros con los cursores. ¿Qué pasa si os ponéis en X=2.5?
¿Qué se observa si os colocáis en X=18?

LAB 3: Partir del código incorporando las modificaciones del ejercicio anterior.

Hasta ahora tenemos un solo objeto (triángulo) en la escena. Vamos a ver cómo añadir un segundo objeto. Para no tener que volver a transferir los datos de otro objeto a la GPU volveremos a pintar un segundo triángulo (en una trayectoria distinta).

Para dibujar un 2º triángulo simplemente repetimos dos veces la orden de dibujar en la función de rendering. Obviamente, si damos las dos órdenes seguidas el segundo triángulo se pondrá encima del primero y no los vamos a distinguir.

Lo que tenemos que hacer es cambiar la matriz M (que se encarga de posicionar un objeto en la escena) de acuerdo a una nueva trayectoria antes de volver a dar la orden de dibujar.

Usar para la matriz M del primer triángulo la matriz de translación anterior que describía un círculo en el plano XY.

Usar para la M del segundo triángulo la matriz del código original que movía el objeto de arriba/abajo en el eje Z.

Adjuntar código de vuestra función 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?

En la entrega, adjuntar a este fichero doc el código de este programa (con nombre lab123.cpp, incorporando las modificaciones de los 3 ejercicios).

LAB 4: Podemos usar el código anterior para ilustrar la diferencia en el orden de aplicar los operadores de giro (R), translación (T) y escala (S) al modelo:

- Usar la matriz de translación T correspondiente a la trayectoria circular en el plano XY de radio 3.
- Crear una matriz de rotación R (usando la función glm::rotate) que aplique un giro al objeto de 80º por segundo alrededor del eje X.
- Finalmente, crear una matriz S de escalado (glm::scale) con factores 1.0, 0.5 y 1.5 en los ejes X, Y y Z respectivamente.

Usar la matriz M1=TRS para un 1er triángulo, M2=SRT para el 2º y M3=RTS para un tercero. Comprobad visualmente como los tres triángulos no recorren la misma trayectoria.
Adjuntar una captura de pantalla y el código usado en render_scene().

LAB 5: Dibujar 3 triángulos describiendo una trayectoria circular (radio 2) en el plano YZ (con X=0) con una velocidad de 1 vuelta cada 5 segundos.

Los triángulos deben estar separados por 120º = (2*π)/3 rads, esto es, si F es la fase de uno de ellos (argumento del correspondiente sin/cos) la de los otros dos debe ser F-(2*π)/3 y F-(4*π)/3 .

Cada uno de los tres triángulos debe estar girando alrededor de un eje de rotación distinto a 50º por segundo. Los ejes de rotación de los 3 triángulos deben corresponder con los ejes X, Y y Z respectivamente. Usar la función rotate para generar las correspondientes matrices de rotación.
Adjuntar una captura de pantalla con el resultado en este doc y el código de vuestro programa (lab5.cpp).

LAB 6: En el fichero original (GpO_01.cpp) buscar el código GLSL del "vertex shader" (una cadena de texto definida al principio del programa) y eliminar el punto y coma al final de la siguiente línea:

```
layout(location = 0)in vec3 pos;
```

introduciendo un error en el programa GLSL (el que se ejecuta en la GPU).
Volver a compilar el programa. ¿Aparecen errores? ¿Por qué?
Ejecutarlo. ¿Funciona? Observar los avisos que se vuelcan a la consola.

En el "fragment shader" ya se conoce la posición que el fragmento ocupará en la ventana y se puede acceder a ella a través de la variable gl_FragCoord.
Usar la componente y de esa variable (gl_FragCoord.y) para multiplicar el color del fragmento por un valor que sea 0 si estamos cerca de la parte de abajo de la pantalla y 1 si el fragmento está en la parte de arriba. Asumir que se mantiene el tamaño original de ventana (ancho = 600, alto = 450)

Adjuntar código del "fragment shader" modificado y una captura de la imagen resultante donde se observe el efecto del cambio de código.

LAB 7: Vamos partir del código original (GpO_01.cpp) pero ahora dibujando usando índices en vez de los vértices directamente. Basta añadir algunas líneas en la creación del VAO y cambiar la orden de dibujo en la función render_scene() según se explica en las transparencias.

- En crear_triangulo declarar un array (indices) de tipo GLubyte (enteros sin signo de 1 byte) y llenadlo con los índices de los vértices a usar. Como solo tenemos 3 serán 0,1,2.
- Tras haber cerrado el VBO usado para mandar los datos de los vértices y antes de "cerrar" el VAO debemos mandar a la GPU el listado de vértices. Usar los comandos (3) indicados en las transparencias para:
  a) Pedir a OpenGL un identificador de buffer.
  b) Enlazarlo como un buffer de tipo GL_ELEMENT_ARRAY_BUFFER
  c) Transferir (glBufferData) los datos de los índices.

  Dejar enlazado el buffer de índices y cerrar el VAO.
- Junto al comando obj.Nv=3; donde guardamos el número de vértices del objeto añadir obj.Ni=3, para guardar el número de índices (que no tienen por qué coincidir).

- Ya solo queda cambiar la orden de dibujo en render_scene. Con índices la orden usada es glDrawElements() en vez de glDrawArrays():

```
glDrawElements(GL_TRIANGLES, N_indices, GL_UNSIGNED_BYTE, 0);
```

  El único parámetro nuevo de glDrawElements (ver transparencias) es el tipo de datos usado para los índices. Puede ser GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT o GL_UNSIGNED_INT según hallamos declarado los índices como GLubyte (1 byte), GLushort (2 bytes) o GLuint (4).

Adjuntad código de vuestra nueva función crear_triangulo().

En este caso no tiene mucho sentido usar índices, porque con solo tres vértices no tenemos ningún ahorro (Ni = Nv). Al usar figuras más complicadas donde varios triángulos comparten el mismo vértice si que es más económico mandar la info de los vértices una sola vez y citarlos luego por sus índices.

Modificar ahora la definición del array vertex_data[]_para que el objeto sea un cuadrado de coord. (0,-1,1), (0,1,1), (0,1,-1) y (0,-1,-1) contenido en el plano YZ. Mantened los colores rojo, verde, azul para los 3 primeros vértices y usar el amarillo (1,1,0) como color para el 4º vértice.
Adjuntad nuevo código para crear vertex_data.

¿Cuál sería ahora vuestro vector de índices?. Adjuntadlo.
¿Cuántos vértices e índices tiene nuestro objeto? Usadlos en vuestro código para asignar los valores adecuados a obj.Nv y obj.Ni.

Adjuntad código completo de vuestro programa incorporando el uso de índices y mostrando un cuadrado en vez de un triángulo (llamadlo lab7.cpp)