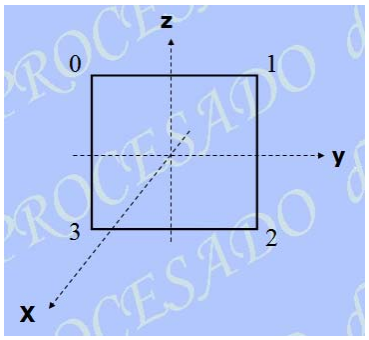
Apellidos:	Nombre:
Apellidos:	Nombre:

Objetivos: - Aprender a usar texturas en OpenGL: crear objetos textura, asignarles propiedades, etc.
- Entender el concepto de coordenadas de textura y como usarlas dentro de los "shaders" para aplicarlas sobre un modelo3D.

En este laboratorio usamos algunas funciones auxiliares para leer una imagen de un fichero .bmp o cargar los vértices y atributos de un modelo 3D desde un fichero. Al ser funciones que no vamos a tocar están en GpO_aux.cpp. No adjuntar GpO_aux.cpp en las entregas puesto que no lo vais a modificar.

Para que el programa pueda leer los datos de los ficheros de las imágenes (bmp) y los modelos 3D (.bix), éstos deben estar en el directorio de ejecución (donde aparezca el ejecutable ejemplo.exe). Usualmente será el directorio ./Debug, colgando del directorio de vuestro proyecto).

Ejer1: Partimos del programa suministrado "GpO_03.cpp". Hacer las modificaciones necesarias sobre dicho fichero. Dentro del programa, la función crea_cuadrado() envía a la GPU los datos del cuadrado de la figura adjunta con 4 vértices (0,1,2,3) formando 2 triángulos. Además de sus coordenadas espaciales XYZ, cada vértice tiene como atributo unas coordenadas de textura (u,v).
En init_scene() también se carga la imagen contenida en "foto0.bmp" asignándose (GL_TEXTURE0) al slot 0 de las texturas activas en la GPU.



Al ejecutarlo veréis sólo un cuadrado rojo, ya que hay dos cosas que faltan:

- El código del fragment shader asigna siempre el color rojo al fragmento. En lugar de eso debería muestrear la textura usando la función texture() como hemos visto en las trasparencias.
- Las coordenadas (u,v) que se listan como atributos de cada vértice en la función de crear_cuadrado() están todas inicializadas a (0,0).

a) Arreglar ambos aspectos, siguiendo las indicaciones de las transparencias. Una vez que aparezca la imagen sobre el cuadrado, hacer girar el cuadrado (con una matriz de rotación R) alrededor del eje Z a una velocidad de 60º/segundo. Como veis, una vez que hemos "pegado" la textura al cuadrado, ésta acompaña al objeto y no tenemos que preocuparnos en hacer nada más, da igual lo complicado que sea nuestro modelo o el movimiento que tenga.

Adjuntar nuevo código del fragment shader y las coordenadas (u,v) asignadas a los vértices en vertex_data[]. Adjuntar captura de la imagen resultante.

b) En init_scene() cargad una segunda textura: leerla del fichero "foto1.bmp" y asociarla al "slot" 1. Modificar el código (en render_scene) para que el programa use alternativamente las texturas del slot 0 y 1. Para ello, cambiar el valor de la variable de tipo sampler2D que tenéis definida en el "frament shader", usando:

transfer_int("nombre variable uniform", valor_asignado)

que opera de forma similar a la función transfer_mat4() que estamos usando para transferir las matrices 4x4 entre la aplicación y la GPU.

Para que la textura mostrada cambie con el tiempo, podéis calcular sin(tt) y alternar la "texture unit" (0/1) según ese valor sea positivo o negativo.
Adjuntar vuestras modificaciones del código en render_scene

c) Cambiar el valor de las coordenadas de la textura asignadas a los vértices para que en vez de cubrir el intervalo [0,1]x[0,1] tomen valores en [-0.5,2.5] en la u (horizontal) y [0,3] para la v (vertical). Adjuntad imagen resultante.

Ejer2: Partir de vuestro código final del Ejer1. Vamos a cambiar el cuadrado por un modelo 3D de una esfera con 256 vértices y 450 triángulos. Como este modelo ya tiene un número considerable de vértices, en vez de introducir los datos manualmente como hasta ahora en las variables vertex_data[] e indices[] los cargaremos de un fichero binario (extensión .bix) usando la función auxiliar:

obj=cargar_modelo("esfera_256.bix")

que lee los datos de los vértices de "esfera_256.bix", junto con el número de vértices, triángulos e índices. En este caso por cada vértice tenemos 5 datos (posición XYZ y las coordenadas UV sobre la textura).

Sustituir la creación del cuadrado por la carga del modelo de la esfera en la inicialización de la escena. ¿Se ve correctamente? ¿Cómo podrías arreglarlo? Adjuntar modificación del código necesario para arreglar problema (1 línea).

No tiene mucho sentido superponer las fotos anteriores sobre una esfera: en su lugar vamos usar las imágenes de los ficheros tierra.bmp y luna.bmp. Adicionalmente, añadir las texturas de marte.bmp y jupiter.bmp en los sucesivos "slots" 2 y 3. En vez de cambiar entre ellas con el paso del tiempo, escribir un código que muestre una u otra si pulsamos las teclas:

t (tierra), **l** (luna), **m** (marte), **j** (jupiter).

Para ello crear una variable global y asignarle los valores (0, 1, 2, 3) usando la función de eventos del teclado. Usar esa variable para seleccionar la textura a usar en render_scene().

Adjuntar código de vuestra función de teclado y las líneas de código añadidas en render_scene(). Adjuntar imagen resultante al seleccionar **j**.

Podéis añadir más realismo a la visualización añadiendo la escala de los planetas según la siguiente tabla:

Cuerpo	Tierra	Luna	Marte	Jupiter
Escala	1.0	0.27	0.53	11.0

Usar de nuevo una variable global (escala) y usar una matriz de escalado S=scale(sx,sy,sz) previa a la visualización. Modificar la variable escala en la función de eventos de teclado, junto con la asignación del número de textura.

Adjuntar captura de pantalla para el caso de Marte ¿Funciona correctamente en todos los casos? ¿Por qué?

Ejer3: Partir del código del ejercicio anterior. En vez de cargar el modelo de la esfera, cargar el modelo 3D contenido en halo.bix, junto con su textura halo.bmp. Visualizar el resultado con las siguientes modificaciones:

- Usar (0.3,0.3,0.1) como color de fondo de la escena.
- Colocar al observador en la posición (X=0,Y=2,Z=3), mirando hacia el punto (0,1,0) y haciendo que el eje Y sea la dirección up.
- Hacer girar al modelo a p.e. 40º por segundo alrededor del eje Y.
- Activar el uso del zbuffer si no lo habéis hecho en el ejercicio anterior.

Adjuntar una captura del resultado.

Éste y los otros modelos del ejercicio los he bajado de http://tf3dm.com/, en el formato OBJ (sencillo formato de texto para modelos 3D), convirtiéndolos al formato binario que usamos en este curso para cargar los datos. Los modelos se han creado con un software especializado tipo Blender,Maya,3DSmax, etc.

a) Para apreciar la complejidad del modelo 3D usado modificar la función de eventos de teclado para que pulsando la letra 't' (toggle) podamos alternar entre una visión de polígonos o de líneas. Usar las funciones ya conocidas:

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

Adjuntar una captura con la opción "wireframe"

Cargar las otras dos figuritas (spider.bix + spider.bmp y thor.bix + thor.bmp). Colocarlas (sin hacerlas girar) desplazadas una unidad sobre el eje de las X de forma que veamos el modelo inicial en el centro, Thor a la derecha y spiderman a la izquierda. Adjuntar captura con las tres figuras.

¿Cuál es el problema con spiderman? ¿Cómo podríamos hacer para verlo como las otras figuras? Adjuntar código necesario para corregirlo y ver las tres figuras correctamente.
Adjuntar captura de pantalla una vez solucionado el problema.

Una vez arreglado el problema anterior poner también a girar a Thor y a spiderman en el eje Z pero en sentido contrario a la figura de halo. ¿Veis algún problema? ¿Podríais solucionarlo?

Intercambiar las texturas de forma que ningún modelo tenga asociada su propia textura. Adjuntar captura de pantalla.

Adjuntar código fuente del ejercicio como ejer3.cpp

Ejer4: En este ejercicio vamos usar 2 texturas simultáneamente dentro del mismo shader (no alternando entre ellas como hasta ahora). Partir del código del ejercicio 2 (esfera girando con una textura superpuesta) y modificarlo de la siguiente forma:

En el fragment shader:

- Declarar una segunda variable de tipo sampler2D. Simplemente duplicar la declaración ya existente dándole otro nombre a la variable.

- Muestrear ambas texturas en las mismas coordenadas (UV), duplicando el código de muestrear la 1ª textura. Guardar los resultados en dos variables vec3 distintas (col1 y col2).

- Para asignar col1 o col2 al color del fragmento vamos a usar la posición del fragmento en la pantalla, en particular su posición en el eje Y (altura) que podemos acceder con gl_FragCoord.y. Si la coordenada Y es menor de 225 (parte de abajo de la ventana) usar el resultado de la primera textura. En caso contrario usar el resultado de la segunda.

En la aplicación (inicialización de la escena)

- Cargar tierra.bmp para la 1ª textura (texture slot 0) y "jupiter.bmp" para la 2ª textura (texture unit 1).

En la aplicación (render_scene):

- Dar los valores adecuados (0 y 1) a las dos variables sampler2D del fragment_shader usando transfer_int().

Adjuntar código del fragment shader modificado y una captura del resultado.

Con el código anterior teníamos una transición brusca entre un planeta y otro. Por debajo de gl_FragCoord.y=225 usamos col1 y por encima usamos col2. En muchas ocasiones deseamos transicionar de una forma más suave entre dos imágenes. Para eso podemos usar la función mix() de GLSL que recibe tres argumentos:

```
color=mix(col1,col2,t);
```

La función devuelve una mezcla entre col1 y col2: color = (1-t)·col1 + t·col2 de forma que para t=0 obtenemos col1, para t=1, col2 y para valores intermedios una mezcla entre los dos.

En nuestro caso tenemos que dar un valor a t (entre 0 y 1) que dependa de la coordenada Y del fragmento en la ventana. Podríamos calcular el parámetro t nosotros mismos, pero GLSL dispone de otra función para hacer esto:

```
t=smoothstep(Y0,Y1,Y);
```

smoothstep() devuelve 0 si Y<Y0, 1 si Y>Y1 y un valor entre 0 y 1 para los valores intermedios. Usad 150 y 300 para los valores de Y0 e Y1.
Adjuntad la imagen resultante y el código usado.

Finalmente vamos a ver como usar la 2ª textura para decidir si se pinta o no un fragmento. Partir del código anterior y substituir la 2ª textura por la contenida en "tablero.bmp".

Dentro del fragment shader, en lugar de combinar ambas texturas, volver a asignar al parámetro de salida el resultado de muestrear la 1ª textura (col1). El resultado será nuestra conocida imagen de la Tierra girando.

El valor de la segunda textura (col2) lo usaremos para decidir si pintamos o no el fragmento con el valor de col1. Para ello tenemos el comando **discard** en GLSL: si se ejecuta ese comando se descarta el fragmento con el que estamos trabajando antes de pintarlo en la pantalla.

Usaremos el valor de la segunda textura para decidir si descartar o no el fragmento. La 2ª textura es una imagen en blanco y negro, por lo que basta comprobar una de sus componentes, por ejemplo la roja, col2.r. Si col2.r es 0 descartaremos el fragmento.

Adjuntar código del fragment shader modificado y una captura del resultado.
¿Por qué no se ve lo que hay detrás de los "agujeros"? ¿Cómo arreglarlo? Modificar el código (1 línea) y adjuntar captura gráfica del resultado.

En otras aplicaciones una de las texturas podría ser una imagen y la otra un mapa de sombras para modular la luz de forma realista. Otra posibilidad es tener varias texturas (hierba, roca, nieve) para usar con un paisaje. En función de la altura del terreno se usaría una u otra textura.

Ejer5: Partir del código del ejercicio 2, mostrando el modelo de una esfera con la textura de la Tierra superpuesta. En este modelo 3D cada vértice lleva asociadas, además de su posición (xyz) las coordenadas (u,v) que usamos para saber que punto de la foto de la Tierra deseamos pintar en esa posición. En el código del vertex shader se corresponden al parámetro de entrada uv (vec2) que es simplemente copiado en el parámetro de salida UV, que es a su vez usado por el fragment shader para muestrear el objeto textura.

En este caso tan sencillo de una esfera es posible deducir los valores de u y v a partir de las coordenadas (xyz), ya que corresponden respectivamente a las coordenadas de longitud y latitud escaladas en el intervalo [0,1].

Vamos a modificar el vertex shader para calcular nosotros los correspondiente valores de u y v. La relación entre (u,v) y la posición sobre la esfera (xyz) es:

u = 0.5+atan(y,x)/(2π); v = 0.5 + atan(z/p)/ π donde p=sqrt(x^2+y^2)

La posición (xyz) está disponible en el parámetro pos y podéis acceder a sus respectivas componentes haciendo pos.x, pos.y o pos.z. El valor de p es la norma de la proyección del vector pos sobre el plano xy y la forma más sencilla de calcularlo en GLSL es hacer p=length(pos.xy). Cualquier variable nueva que vayáis a usar declararla haciendo p.e. float p=length(pos.xy);
También podéis declarar pi como una constante (fuera del cuerpo de main) haciendo `const float pi = 3.14159265358979;`

Tras calcular las coordenadas u y v cread con ellos el vector UV=vec2(u,v); en vez de usar los datos del modelo (UV=uv; en el código original).

Modificar poco a poco el código GLSL y recordad que si cometéis algún error de sintaxis no aparecerá al compilar en VS sino en tiempo de ejecución. Una vez que no tengáis errores adjuntad el código añadido al vertex_shader.

Ejecutar el programa. ¿Observáis algún problema? Adjuntad una captura donde se vea. ¿A qué creéis que puede ser debido? ¿Por qué creéis que no aparece al usar los datos del modelo?

Ejer6: Volver a partir del código del ejercicio 2 (esfera girando con imagen superpuesta), usando “tierra.bmp” para la textura de superposición.

Añadir una segunda esfera, con las siguientes características:

- Escalada (S) en un factor 0.3 con respecto a la primera.
- Girando (R) sobre sí misma (eje Z) a 12º por segundo.
- Describiendo una trayectoria circular (R=4) en el plano XY con un periodo de 30 segundos.
- Recubierta con la textura de la imagen “luna.bmp”

Colocar al observador en eje X a distancia d=6 unidades del origen. Activar las opciones de "culling" y "zbuffer" para visualizar correctamente la escena.

Adjuntar código de la función de rendering y una captura de la escena.

Usar el teclado para poder mover el observador de la siguiente forma:

- Cursores arriba/abajo: nos moveremos en elevación manteniendo la distancia al origen (radio) constante. Moviéndonos hacia arriba veríamos el sistema Tierra/Luna desde “arriba” (Polo Norte) y viceversa.

- Teclas w/s para alejarme/acercarme (cambiando distancia d al origen).

La posición del observador se calcula a partir de estas 2 variables como:

(X= d *cos(elev), Y = 0.0, Z = d*sin(elev))

Crear un par de variables globales d y elev (de tipo float) y hacer que se modifiquen al pulsar las teclas indicadas. Para los incrementos usad 0.0175 rads (1º) para la elevación y 0.05 para la distancia d. En cada modificación, volcar los valores de elev (en grados) y d.

En la función de refrescar la pantalla, calcular pos_obs y recrear la matriz V.

Adjuntar vuestro código de eventos de teclado y una captura del sistema vista desde una elevación de 70º sur.

Meteros “dentro” de la Tierra (d<1), ¿qué veis? Quitar GL_CULL_FACE en las opciones y volver a probar. Adjuntar la imagen resultante (p.e con d=0.8).

Adjuntar código de este ejercicio como ejer6.cpp