Apellidos:	Nombre:
Apellidos:	Nombre:

Responder a las preguntas y adjuntar los "trozos" de código pedidos. Entregad este fichero doc con las respuestas. Solo cuando se pida explícitamente adjuntar fichero fuente en un zip o similar.

Objetivos:

- Modelo de Phong: iluminación difusa y especular
- Implementación a nivel de vértices (Gouraud) o de fragmentos (Phong)
- Combinar el uso de iluminación y texturas en OpenGL.

Ejer1: Partir del código del día anterior de iluminación difusa con las opciones de mover la luz (cursores) y alternar (F1) entre la implementación a nivel de vértices (Gourad) o de fragmentos (Phong). Corresponde al código de los primeros dos ejercicios de la hoja anterior (ejer12.cpp).

Quitar la coloración que tenía la luz en Phong poniendo la correspondiente variable lightcolor como una luz blanca (1,1,1).

En lugar de la esfera, cargar el modelo 3D del fichero "buddha_n.bix" en la aplicación. ¿Cuántos vértices y caras tienes este modelo 3D?

Cambiar el eje de giro para que la figura gire respecto al eje Y a 20º por segundo. Poner la cámara en (0,1,2) y modificar el punto (target) hacia donde miramos para que la figura aparezca centrada en la pantalla.

Usar el método de Phong (por fragmentos) y mover la luz para que venga por la izquierda, ligeramente de arriba. Adjuntar captura.

Llevar la luz (moviendo la elevación hasta unos 90º) justo encima de la figura. Adjuntar captura. ¿Notáis algo raro? Dar un ejemplo de alguna zona que aparece iluminada y no debería estarlo. ¿De dónde viene este problema?

No adjuntar el código pero guardarlo para usarlo en el ejercicio LAB4.

Ejer2:. En este ejercicio vamos a añadir la componente especular a nuestro modelo de iluminación. Volver a partir del código del día anterior (ejer12.cpp) con el modelo de la esfera (esfera_106n.bix).

En primer lugar haremos una implementación a nivel de vértices (Gourad), por lo que modificaremos las cadenas `vertex_prog1` y `fragment_prog1`. De hecho, como en la implementación de Gourad, todo el trabajo se hace a nivel de vértices, solo tendremos que modificar el código del “vertex shader”.

Modificación del vertex shader: para calcular la componente especular necesitamos información adicional:

- La matriz (mat4 M) que pasa los vértices a coordenadas de escena. Notad que ahora tenemos la matriz MVP, en la que están "mezcladas" la matriz M, la V (cambio de vista) y la P (proyección).

- La posición de la cámara (vec3 campos) para poder calcular el vector v (dirección vértice --> observador).

Añadir estas dos nuevas variables a las ya existentes en el shader (lightdir, M_normales, MVP). Al igual que ellas, declararlas con el modificador uniform para poder cambiarlas desde la aplicación.

Dentro del main() declarar dos nuevos vectores (v y r) de tipo vec3 que serán el vector unitario desde cada vértice a la cámara (v) y el vector unitario que marca la dirección de reflexión perfecta (r).

- Calcular v normalizando el vector diferencia entre la posición cámara y la posición del vértice (ambas en coordenadas escena).

  Para conocer la posición del vértice en la escena aplicaremos la matriz M a la posición del vértice pos. Como pos es un vector vec3 y M es una matriz 4x4 debemos ampliar pos a coordenadas homogéneas (añadiendo un 1), aplicarle M y quedarnos con sus tres primeras componentes (usando .xyz en GLSL).

- Calcular r como: $\bar{r} = 2(\bar{l} \cdot \bar{n}) \cdot \bar{n} - \bar{l}$ con n la normal y l la dirección que apunta hacia la luz (lightdir en el código).

  Alternativamente podemos usar la función reflect: r = reflect(-l,n).
  La razón del signo - es que reflect() espera que l sea el vector unitario **desde** la luz, y en nuestra definición es el vector unitario **hacia** la luz

Una vez que disponemos de r y v la componente especular se calcula como:

$$\text{comp\_esp} = \cos(\bar{r},\bar{v})^n = (\bar{r} \cdot \bar{v})^n$$

En GLSL el producto escalar entre dos vectores es dot(x,y) y la potencia es pow(x,n). Usar clamp() para restringir el valor del producto escalar en (0,1). Para el coeficiente especular (exponente n) usad n=16.

Cuando tengáis calculada la componente especular añadirla a la iluminación final, donde ya estaban la componente ambiental y difusa. Usar un reparto de 0.1 (ambiente), 0.6 (difusa) y 0.3 (especular) entre las tres componentes.

Adjuntar vuestro código del vertex_shading

Modificación de la aplicación: En la aplicación (render_scene) tenéis que transferir la matriz **M** (mat4) y la posición de la cámara **campos** (vec3) al programa de la GPU usando las funciones de transferencia adecuadas.

Cambiar la posición de la luz y observar el resultado. Veréis que no es muy bueno, ya que se notan mucho las facetas del modelo subyacente. Cambiar el modelo por el de esfera_520_n.bin (520 caras, unas 5 veces el original).
¿Es aceptable ahora el "rendering" de la componente difusa? ¿Y el de los brillos especulares?

Una vez que os funcione la parte especular a nivel de vértices el paso siguiente es llevarnos los cálculos al nivel de fragmentos (Phong) como hicimos el día anterior con la componente difusa. Ahora modificaremos los programas de las cadenas vertex_shader2/fragment_shader2.

En este caso, en el procesador de vértices dejaremos solamente el cálculo de la visual (v) y la modificación de las normales (normal_T). Ambos vectores serán declarados como salida (out) en el procesador de vértices (quitar su declaración dentro del main() y pasados como entrada (in) al procesador de fragmentos donde realizaremos el resto de los cálculos.

Pasar también la normalización de v y normal_T al fragment shader. Es mejor no normalizarlos en el vertex shader porque no hay garantía de que su interpolación (que es lo que recibe el procesador de fragmentos, cortesía del rasterizador) mantenga su carácter unitario.
Adjuntar vuestro nuevo código del vertex_shading y fragment_shader
Adjuntar 2 capturas mostrando las diferencias entre Gouraud y Phong.
Adjuntar vuestro programa completo como ejer2.cpp.

Ejer 3: En el programa anterior los parámetros del modelo (reparto entre las iluminaciones ambiente (0.1), difusa(0.6) y especular (0.3), así como el exponente especular (n=16) estaban fijados en el código del shader.

Modificar el programa anterior (en el código GLSL y la aplicación) para que podamos experimentar cambiando la relación entre las componentes de luz ambiente, difusa y especular y el valor del exponente de reflexión especular:

- **En el código GLSL** (hacedlo solamente en la **implementación de Phong**): declarar un nuevo parámetro uniform de tipo vec4. Sus componentes serán los valores de ambiente/difusa/especular + exponente. Usar dichos valores al calcular la iluminación.

- En la aplicación: crear una variable global de tipo vec4 inicializándola a (0.1, 0.6, 0.3, 16). Usar las teclas 'qwer' para incrementar los valores de sus 4 componentes (con incrementos de 0.05 para los factores de iluminación y de 1 para exponente). Usad las teclas 'asdf' para bajar los valores (con los mismos saltos). Volcar por pantalla los 4 valores cada vez que los cambiéis.

En render_scene(), transferirla a la GPU usando transfer_vec4() (GpO_aux), cuyo uso es similar al de similares funciones como transfer_vec3(), etc.

Comprobad el código con el modelo de la esfera, adjuntando capturas con:

- Solo iluminación ambiente (Ca=0.3,Cd=0,Ce=0)
- Solo difusa: Ca=0, Cd=1, Ce=0
- Solo especular Ca=0, Cd=0, Ce=1, con n=25

A continuación, cargar el modelo 3D del fichero "halo_n.bix", haciendo las siguientes modificaciones:

- Poner al observador en (0,1,2) y haciendo que su target sea (0, 0.9, 0).
- Cambiar el eje de giro al eje Y (0,1,0) y hacerle girar a 10º/segundo.

Usando la implementación de Phong, adjuntad capturas para las siguientes valores de los coeficientes (ambiente/difusa/especular/exponente):

- 0.10, 0.90, 0.00
- 0.05, 0.10, 0.85, n=8
- 0.10, 0.50, 0.70, n=30

Observar que con los cambios en los coeficientes podemos dar la impresión de que el tipo de material cambia. Si en vez de ser comunes para toda la escena hacemos que dichos coeficientes cambien para las distintas partes del modelo (pasándoselos como atributos de los vértices), podríamos simular diferencias p.e. entre partes plásticas y metálicas de la armadura, etc.

Adjuntar código fuente como ejer3.cpp

Ejer4: En los ejercicios anteriores es como si los modelos fuesen estatuillas de color blanco, ya que el color final de cada fragmento solo dependía de su iluminación. Normalmente nuestros modelos 3D tendrán sus propios colores y el color de cada fragmento será una mezcla (multiplicación) entre el color del modelo en cada punto y la mayor o menor iluminación recibida. En este ejercicio combinaremos el uso de texturas e iluminación.

Para simplificar, partiremos del código del ejercicio1 (solo iluminación difusa), modificando la implementación de Phong en vertex_prog2 y fragment_prog2. Aseguraros de que en init_scene() seleccionamos dicha implementación para que corra por defecto (usando `glUseProgram()`).

Como vimos, la información del color en un modelo complejo se suele dar en una textura. Por lo tanto, en este ejercicio necesitaremos un modelo que nos suministre en cada vértice su posición (xyz), coordenadas de la textura (u,v) y normal (nx,ny,nz). En el modelo que usaremos el orden en el que están

guardados los atributos es: pos (atributo0), coordenadas textura (atributo1) y normal (atributo2). Debido a esto tendremos que cambiar la declaración de parámetros de entrada (in) en el vertex shader:

```
layout(location =0) in vec3 pos;
layout(location =1) in vec2 uv;
layout(location =2) in vec3 normal;
```

De esta forma le indicamos al vertex shader que esperamos 3 argumentos de entrada por cada vértice, indicando su tipo y el orden en el que va a llegar.

Aparte de esta modificación, los cambios son mínimos:

En el **vertex_shader** no vamos a hacer nada con la información de texturas salvo pasarsela al fragment_shader. Simplemente:

- Declaramos un argumento adicional de salida (out) llamado UV
- Dentro del programa copiamos uv a UV.

En el **fragment shader** repetimos lo que hicimos en la clase de texturas:

- Declaramos UV como argumento de entrada (in).
- Creamos una variable uniform de tipo sampler2D.
- Dentro del main() usamos la función texture() para muestrear imagen. Luego mezclamos (multiplicando) el color obtenido de la textura con el valor de la iluminación ya calculado por nuestro código.

En **la aplicación** (inicialización de escena):

- Cargar los datos del modelo 3D del fichero "halo_tn.bix" (vértices con las coordenadas de textura + normales).
- Cargar (usando cargar_textura_from_bmp) la textura contenida en el fichero "halo.bmp" y asociarla a GL_TEXTURE0 (slot 0).
- Asegurarnos de darle el valor 0 a la variable tex en el programa de la GPU para que se use el slot adecuado.

Adjuntar una captura del resultado, colocando la luz de forma que venga ligeramente de abajo (elev -30 o -40) y con un ángulo de unos 45º (azimuth) a la derecha del observador. Adjuntar vuestro código como ejer4.cpp

Ejer5: Se trata de juntar un poco de todo lo que hemos aprendido de modelos, texturas e iluminación para hacer una pequeña demo de la Tierra rotando sobre si misma iluminada por una luz que representa el Sol. Aunque sabemos que es al revés, la luz del Sol girará alrededor de la Tierra en nuestra demo para ilustrar el paso de las estaciones.

Podéis usar el código del ejercicio anterior como punto de partida, sobre el que haremos las siguientes modificaciones (sobre el código de Phong):

- Como modelo de la esfera usaremos el del fichero “esfera_961_tn.bix” (961 vértices) que contiene datos de las posiciones de los vértices (atrib 0), coordenadas de textura (atrib 1) y normales (atrib 2). Es el mismo orden de argumentos que el usado en el ejercicio anterior.

- El modelo se colocará en el origen, con el eje Z hacia arriba. El observador se colocará en el eje X, mirando hacia el origen, y a una distancia tal que la Tierra llene la ventana.

- Se cargará la textura "tierra2.bmp" que se usará para recubrir la esfera con un mapa de la Tierra.

- Para simular los días, haremos rotar a la Tierra sobre sí misma (eje Z) a 90º por segundo. Esto corresponde a 4 segundos para 1 vuelta (360º). Por lo tanto en nuestra simulación 1 día pasa en 4 segundos.

  Compilar los cambios anteriores y ajustar los parámetros para ver la Tierra girando correctamente alrededor del Polo. Adjuntar captura.

- Como en el ejemplo anterior los colores de la textura se combinan en el shader con una iluminación difusa usando una luz lejana (Sol). El reparto será 0.05 de ambiente y 0.95 de difusa. Para que la frontera noche-día esté más definida usaremos para la iluminación difusa la raíz cúbica del coseno del ángulo entre la normal y la luz (en vez de directamente el coseno).

- En vez de mover la posición de la luz (Sol) con los cursores, haremos girar automáticamente la posición de la luz para simular el paso un año, variando su azimuth (az) y elevación (el). Usar como vector dirección de la luz:

  $$[\ \cos(el)\cdot\cos(az)\ ,\ \cos(el)\cdot\sin(az)\ ,\ \sin(el)\ ] \quad (1)$$

  y transferir dicho vector a la GPU antes de pintar la escena.

- En un principio usaremos el=0 (el Sol siempre estaría sobre el Ecuador). El azimuth se incrementará con el tiempo para simular el "movimiento" del Sol alrededor de la Tierra. El azimuth debería tardar 1 año en dar la vuelta completa (360º = 2π rads). Con nuestra velocidad de 1 dia = 4

seg, un año serían 1060 segundos (unos 20 minutos). Esto es demasiado lento, así que lo vamos a acelerar, haciendo que el azimuth avance a razón de 0.02 rads/seg.

- Añadir una variable elev_obs y usarla para cambiar la elevación del observador sobre el Ecuador. En función de esta variable, la posición del observador sería:

  $$pos\_obs = R \cdot [\cos(elev\_obs), 0, \sin(elev\_obs)]$$

  donde R es la distancia la que habéis tenido que colocar el observador para ver la Tierra llenando la pantalla.

Compilar los cambios anteriores. Deberíais ver como se mueve la línea día-noche, pero siempre pasando por los polos. Adjuntar captura donde se aprecie la separación día-noche.

Las estaciones son debidas al hecho de que el Sol no “gira” en el mismo plano que el Ecuador. El eje de la Tierra está inclinado 23.5 grados respecto al plano de “giro” del Sol (eclíptica). Como en nuestra simulación la Tierra está girando "derecha" lo que haremos es inclinar el recorrido del Sol. Esto equivale a qe la luz, durante una vuelta en azimuth, debe variar también su elevación, oscilando entre -23.5º y 23.5º. Una sencilla relación que cumple esto (no necesariamente realista) entre la elevación y el azimuth de la luz es:

$$elev = 23.5º \text{ (en rads)} \times \cos(azimuth). \quad (2)$$

En resumen, el azimuth de la luz se incrementa proporcionalmente al tiempo en la simulación (como hacíamos antes), pero ahora la elevación de la luz cambia según la fórmula anterior (2). En render_scene() calcular la dirección de la luz en función de su azimuth y elevación (1), transfiriéndola a la GPU antes de dar la orden de pintar.

Correr la demo y cambiar la posición del observador para ver los Polos. Podréis ver como se mantienen en la sombra o la luz (dependiendo de la estación) durante la mitad del año. Adjuntar una captura de la Tierra vista desde unos 70º sur durante el verano austral.

Adjuntar vuestro programa como ejer5.cpp