Apellidos:	Nombre:

Incluir vuestras respuestas, capturas y "trozos" de código pedidos en este fichero doc. Cuando se pida, adjuntad fichero fuente completo.

El programa GpO_tarea_ilu.cpp dado implementa una iluminación difusa a nivel de fragmentos. La luz es una luz lejana cuya dirección (vector dir_luz) puede cambiarse en azimuth y elevación usando las teclas de los cursores. El modelo es un cuadrado en el plano Z=0, cubriendo el área [-2,2]x[-2,2] en el plano XY. Al ser tan sencillo usamos el método de especificar los atributos de los vértices (posición x,y,z + normales nx,ny,nz) en el array vertex_data[], dentro de la función crear_cuadrado(), dónde también enviamos esos datos a la GPU, especificando los distintos atributos por vértice.

El observador está en la posición (X=4,Y=0,Z=1), mirando hacia el origen, por lo que ve el cuadrado un poco desde arriba. El eje Z está hacia arriba.

Haced un gráfico con la disposición del cuadrado en la escena, la posición del observador y los vectores normales asociados a cada vértice. Hacerle una foto y adjuntarlo. ¿Son iguales las normales asociadas a cada vértice? ¿Por qué?

Compilad/ejecutad el programa: con los parámetros usados el cuadrado se ve casi al completo, excepto por sus vértices más cercanos. Como el objeto mostrado es plano, el ángulo de su normal con la luz es constante en todos sus puntos y se ve iluminado de forma uniforme (con una luz amarillenta)

a) El objeto se muestra inmóvil al ser la matriz M usada en render_scene() la identidad. Haced oscilar el cuadrado alrededor del eje X entre -20º y 20º, usando para M una matriz de rotación (rotate). El argumento de giro podría ser algo como 20·sin(tt) o similar para que oscile en el rango pedido.

Adjuntad código para crear la matriz M. Observar como ahora la iluminación cambia al cambiar la inclinación del plano con la luz. Adjuntar captura en el momento de mayor "brillo". ¿Dónde está situada la luz en esta escena?

Modificar el código del procesador de vértices, comentando la línea en la que se aplica la matriz adjunta (transpuesta de la inversa de M) transformando las normales. En su lugar mantened las normales originales haciendo: `n_modif=n;`

Ejecutad y observar el resultado. ¿Cambia ahora la iluminación al oscilar? Explicad que está pasando.

b) Partir del código del apartado a) con el plano oscilando **(restaurar el código del procesador de vértices a su versión original correcta)**.

Lo que vamos a hacer en este apartado es modificar el código del shader para simular una **luz local o cercana**, en vez de la luz lejana ahora implementada.

Una luz cercana se define por su posición (vec3) en la escena (Lx,Ly,Lz) y por una atenuación que en este caso será:

$$I = \frac{1}{\left(1 + 0.5 \cdot d^2\right)}$$

donde d es la distancia entre la posición de la luz (Lx,Ly,Lz) y la posición del punto (vértice o fragmento) en el que estamos calculando su iluminación.

Cambios a realizar en el vertex shader (GLSL):

- Necesitamos conocer la **posición** del vértice **en coordenadas escena**. Para ello el vertex shader debe tener acceso a la matriz M por separado (declaradla de tipo uniform para poder cambiarla desde la aplicación).

- Añadir un nueva variable de salida pos_escena (out vec3), para pasar la posición del vértice al procesador de fragmentos. Para calcular esta posición del vértice en escena se aplica M (mat4) al vector pos ampliado a coordenadas homogéneas (vec4). Del resultado nos quedamos (.xyz) con sus 3 primeras componentes obteniendo por lo tanto un vec3.

Cambios a realizar en el fragment shader (GLSL):

- La propiedad "fija" de la luz es ahora su posición (pos_light) que vendrá dada por un vector vec3 con las coordenadas (x,y,z) de la luz en los ejes de la escena. La declararemos como uniform dentro del fragment shader para poder cambiarla (mover la luz) desde la aplicación.

- Como en el procesador de vértices tenemos un argumento de salida adicional, tenemos que declararlo aquí como argumento de entrada.

- La variable L (vector unitario indicando la dirección de la luz) ya no es una propiedad de la luz constante para toda la escena. Por lo tanto, deja de ser un uniform y pasa a convertirse en una variable de tipo vec3 declarada dentro del main() del fragment shader. Ahora tiene que ser calculada para cada fragmento.

- Asimismo declararemos d (distancia fragmento-luz) como un float. La distancia d entre el fragmento y la luz se necesita para calcular la atenuación de la luz local.

- Para calcular L y d hay que determinar el vector (vec3) que va desde la posición del fragmento a la de la luz (ambas en coordenadas escena), restando ambas posiciones:

  (pos_light – pos_escena).

- A partir del vector anterior podemos calcular fácilmente L y d:

  - d es la norma (función length) de ese vector.
  - L es ese mismo vector normalizado (función normalize).

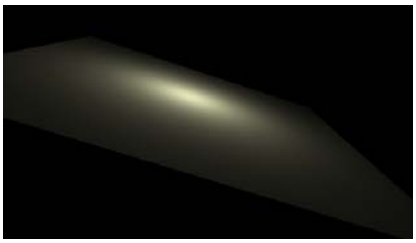
- Una vez que disponemos de L podemos usarlo (junto con la normal n) para calcular la iluminación (esto ya estaba hecho). Esa iluminación la atenuamos ahora con la fórmula anterior usando la distancia d.

Cambios a realizar en el código de la aplicación:

- Cambiar la variable con la "dirección de luz" que ya no tiene sentido en una luz local por una nueva variable (también vec3) con la "posición de la luz" en coordenadas de la escena. Inicializarla con las coordenadas (0.0,0.0,0.5), un poco por encima del centro del cuadrado.

- Para que la GPU conozca los valores de las nuevas variables "uniform" declaradas (posición de la luz y matriz M) habrá que transferirlas en la función render_scene(), usando las funciones habituales transfer_xxx().

Ejecutad el programa, Deberías ver un resultado parecido al de la figura adjunta, con el típico efecto de atenuación por la distancia de una luz local.



Adjuntad una captura del resultad y el código de los shaders (vértices y fragmentos) modificados.

Una vez que esto esté funcionando, modificar el código de la aplicación para poder mover la luz por la escena:

- En el eje Y (usando los cursores izquierda/derecha).
- En el eje Z (arriba/abajo) usando los cursores arriba/abajo.

Solo tenéis que modificar el código que hay ahora que sirve para mover el azimuth y elevación de la luz lejana. Procurad que la interfaz sea "natural" (el cursor derecha debe mover la luz a la derecha, etc).

En ambos casos, usar 0.05 como incremento/decremento de las componentes de la posición. Cada vez que la modifiquéis, imprimir por la consola la nueva posición YZ de la luz (con 2 decimales, %.2f).

Mover la luz hasta Y=1.0. Observaréis como cambia el tamaño del haz según la distancia entre la luz y el plano. Manteniendo Y=1.0 bajar hasta Z=0.3. ¿Qué pasa ahora? ¿Por qué?
Subir ahora la luz hasta Z=4 o así. Adjuntar captura. ¿Qué le pasa al "círculo" de luz sobre el plano? ¿Por qué?

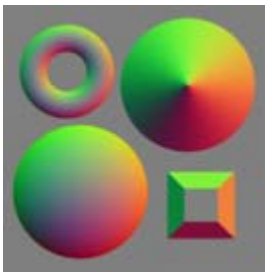
Adjuntar vuestro programa como GpO_tarea_ilu_B.cpp

c) Uso de texturas para "normal mapping"

Este ejercicio puede hacerse independientemente del anterior. Volver a partir del código original (luz lejana + cuadrado oscilante).

En los ejemplos de clase se han usado las texturas casi siempre como imágenes para superponerlas sobre los modelos 3D, dándoles más realismo.

Sin embargo comentamos que las texturas se usan para llevar información 3D (si se trata de una imagen rgb con tres componentes) a cualquier punto del modelo sin necesidad de tener un vértice en ese lugar. En este ejercicio vamos a conectar una textura (imagen rgb) a nuestro cuadrado para disponer de información adicional 3D en cada fragmento a pesar de contar solo con 4 vértices.



En la figura adjunta se muestra la imagen a usar, contenida en el fichero normales.bmp. Nuestro primer objetivo será simplemente visualizar esta imagen sobre nuestro cuadrado, como hicimos en el primer ejercicio de la clase de texturas.

Como hemos dicho nuestro objetivo último no es usar la imagen para verla, pero de esta forma es fácil verificar que la imagen se ha leído y transferido correctamente a la GPU y podemos acceder a sus datos en nuestros shaders. Cuando esto funcione usaremos los datos de la imagen para otros fines.

c1) Para ver la imagen sobre nuestro cuadrado tenemos que hacer una serie de modificaciones, tanto en la aplicación como en los shaders. La mayor parte de este código lo vimos en el primer ejercicio de la clase de texturas.

Cambios en la aplicación:

- Cargar la imagen en init_scene() usando cargar_textura_from_bmp() y asignarla al slot 0 (GL_TEXTURE0).

- En crear_cuadrado(), añadir a vertex_data[] un par de datos extras por cada vértice (fila), con las correspondientes coordenadas textura (u,v) asociadas a ese vértice. Asignad a los vértices existentes (en el orden en el que están declarados) los valores (u,v) = (0,1), (1,1), (1,0), (0,0).

- En crear_cuadrado(), cuando se especifican en el VAO la distribución de los atributos dentro del buffer VBO, hay que indicar que hay un nuevo atributo y sus características. Esto se hace añadiendo un par de líneas similares a las que ya existen para el 1er (0) y 2º (1) atributos:

```
// Especifico 3º argumento (atributo 2) del vertex shader
glEnableVertexAttribArray(2);
glVertexAttribPointer(2, ...);
```

  Buscar en las transparencias la definición de los argumentos de la segunda función y fijaros en el código ya existente para ver como lo hemos hecho para los 2 primeros argumentos. La única diferencia es que el nuevo atributo es un vec2 en vez de un vec3 (el tipo de datos es el mismo, GL_FLOAT) y que empezará en un offset distinto de los dos argumentos anteriores.

  Fijaros que uno de los argumentos de `glVertexAttribPointer()` es el salto (en bytes) que hay que dar para empezar a leer los atributos del siguiente vértice. Ese argumento ha cambiado (ya que ahora hay 2 floats adicionales por vértice) y **también debe cambiarse en la descripción de los atributos anteriores (0,1).**

Cambios en el vertex shader:

- Añadir la declaración del tercer atributo de entrada uv y asignarselo sin tocar a un nuevo argumento de salida UV (para mandarlo al fragment shader).

Cambios en el fragment shader:

- Declarar UV como argumento de entrada y un objeto de tipo sampler2D de tipo uniform.
- En el main() comentar el código actual (iluminación) y simplemente usar las coordenadas UV para muestrear el objeto textura, asignando el resultado al color del fragmento (como se hizo en clase).

Si todo ha ido bien deberíais ver la imagen del fichero bmp superpuesta sobre nuestro cuadrado. Adjuntad captura.

Adjuntad código del fichero fuente como GpO_tarea_ilu_C1.cpp

c2) Todo lo anterior ha sido para verificar que teníamos acceso a los datos de la imagen dentro del fragment shader. La idea ahora es no usar los datos de la imagen para dar color al fragmento, sino con otro objetivo.

El dato que extraemos de la imagen es un vec3, conteniendo información de los canales rojo (r), verde (g) y azul(b) de un píxel de la imagen. En lugar de un color para asignar al fragmento, nosotros lo vamos a interpretar como una perturbación a aplicar a la normal del fragmento.

En nuestro código, cada fragmento recibe un vector normal heredado de los vértices que componen el polígono. Como en este caso tenemos un plano, todos los fragmentos reciben la misma normal (0,0,1), hacia "arriba" y por eso aparecen todos con la misma iluminación.

Ahora partiremos de la normal heredada de los vértices pero usaremos los datos de la imagen para perturbarla, de forma que no apunte hacia el mismo sitio en todos los fragmentos. El resultado será una sensación de relieve 3D en nuestro plano independiente de su forma geométrica (sigue siendo un plano con sólo 4 vértices).

La imagen en el shader tiene valores entre 0 (oscuro) y 1 (claro). Como queremos codificar perturbaciones positivas y negativas restaremos 0.5 a todas las componentes rgb de la imagen. El resultado es un vector con componentes entre -0.5 y 0.5 que usaremos para perturbar la normal.

Si os fijáis en la imagen usada muchas zonas son un gris medio (0.5,0.5,0.5). Una vez restado 0.5 a esas componentes queda una perturbación nula (0,0,0). En esos puntos no se cambia nada, la normal seguirá apùntando hacia arriba y nuestro plano seguirá viéndose como un plano.

Otros puntos muestran una coloración, principalmente rojiza o verdosa, por que sus dos primeras dos componentes (rg) tienden a apartarse del valor medio (0.5). Cuando quitemos 0.5 a sus valores tendremos perturbaciones entre -0.5 y 0.5 en (nx,ny) que apartarán al vector normal de la vertical. La ausencia de color azul en la imagen indica que la componente azul de la imagen tiende a ser del orden de 0.5, lo que indica que la componente Z de la normal no se cambiará mucho.

Se trata de (en el fragment shader):

- definir un vector vec3 delta_n, donde guaradremos la perturbación a aplicar a la normal.
- Muestrear la textura y asignar sus componentes rgb a dicho vector, restando 0.5 a cada una de sus componentes.
- Sumar delta_n a la normal heredada y normalizar el resultado: esa será la nueva normal del fragmento.
- Usar la normal anterior en el cálculo de la iluminación como se hacía antes y asignar el valor de la iluminación al fragmento.

Adjuntad código del fragment shader.
Ejecutad el programa y adjuntad una captura del resultado.

Debéis ver como las sombras cambian según la inclinación del plano, como si los "objetos" de la imagen tuviesen un cierto relieve.

Moved la luz azimuth (cursores izquierda y derecha) y observar como las sombras giran con la luz, lo que refuerza la sensación de que nuestro modelo 3D es una especie de una chapa deformada y no el plano que realmente sigue siendo.

Adjuntad código del fichero fuente como GpO_tarea_ilu_C2.cpp

En nuestro código hay un aspecto que no se ha implementado correctamente. Las normales son transformadas (para acompañar al movimiento) en el vertex shader y luego se les añade la perturbación (en el fragment shader). En realidad deberíamos haber perturbado primero las normales y luego transformarlas.

Describir como implementaríais esta modificación en el código. (No hace falta hacerlo, basta describir que tareas se harían en cada uno de los shaders).