Apellidos, Nombre:
Apellidos, Nombre:

Partir del programa "GpO_04.cpp" y modificarlo según las instrucciones.

Ejer 1: Tal como está, el programa carga un modelo 3D de una esfera con 277 vértices (guardado en el fichero "esfera_277_n.bix"). En este modelo, cada vértice viene, además de con su posición (xyz), con su vector normal (nx,ny,nz) para poder usarlo en el cálculo de la iluminación.

En el programa la esfera mueve alrededor del origen y gira sobre si misma en torno al eje Y (0,1,0). Hay una luz lejana que ilumina la escena desde arriba a la derecha. Para implementar la iluminación se usa un modelo de iluminación difusa usando la implementación de GOURAD (por vértices).

Al hacer los cálculos de iluminación sólo en los vértices, esta implementación de Gouraud es muy dependiente del nivel de detalle del modelo. Cambiar (en init_scene) el modelo de la esfera usada por el de "esfera_106_n.bix" (con 106 vértices) y el de "esfera_520_n.bix" (520 vértices). Adjuntar una captura del resultado en ambos casos.

a) En este apartado vamos a ver cómo compilar y usar varios programas en la GPU. En la función init_scene se han compilado dos programas distintos:

- prog1 (a partir de las cadenas de texto vertex_prog1 y fragment_prog1)
- prog2 (a partir de las cadenas de texto vertex_prog2 y fragment_prog2)

El comando glUseProgram(prog1) hace que el primer programa sea el activo y será el que se ejecute al dar cualquier orden de dibujar.

Si activáis prog2 veréis que el resultado es el mismo (sólo cambia ligeramente el tono de la luz para darnos cuenta del cambio). Salvo esa diferencia los dos códigos son idénticos (ambos implementando iluminación de Gouraud).

En este apartado queremos cambiar entre un programa y otro sin volver a compilar, simplemente pulsando una tecla. Escribir el correspondiente código para que cuando pulsemos la tecla 'F1' alternemos entre los dos programas. Basta tener una variable con el estado actual y usar la función glUseProgram() para seleccionar uno u otro. Adjuntar código añadido. Notad que F1 es una tecla "especial".

b) Una vez que podemos alternar entre los 2 programas, se trata ahora de programar el 2º programa para que haga una implementación de Phong (por fragmentos) de la iluminación difusa.

La iluminación se implementa en el programa (shader) que se ejecuta en la GPU, escritos en lenguaje GLSL. He aquí el código usado en este programa:

```
// Código procesador vertices (Gourad) en vertex_prog1
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 normal;
out vec3 color;
uniform mat4 MVP;
uniform mat4 M_normals;
uniform vec3 lightdir=vec3(1,0,0);
const vec3 lightcolor=vec3(1.0f,1.0f,1.0f);

void main(){
float difusa; vec3 normal_T;

  gl_Position =  MVP*vec4(pos,1);

  normal_T=(M_normals*vec4(normal,0.0f)).xyz;
  normal_T=normalize(normal_T);  //Ambiente + difusa

  difusa=clamp(dot(lightdir,normal_T),0.0f,1.0f);
  color=0.1+0.9*difusa*lightcolor;  //Ambiente + difusa
}

// Código procesador de fregmentos (Gourad) en fragment_prog1
in vec3 color;  // Entrada: color vertice (interpolados en fragmentos)
out vec3 color_fragmento;
void main()
{
  color_fragmento = color;
}
```

En este apartado se trata de modificar el código GLSL del 2º programa para que implemente el método de Phong. Las cadenas de texto a modificar son vertex_prog2 y fragment_prog2. No tocar el código de las cadenas originales vertex_prog1 y fragment_prog1, para que el primer programa siga haciendo lo mismo y poder comparar ambas iluminaciones.

En el caso anterior todo el trabajo se hace en el procesador de vértices (Gouraud calcula iluminación por vértices). El procesador de fragmentos solo recibe el color interpolado para cada fragmento y se lo asigna a su salida.

En una implementación de Phong debemos llevar la mayoría de estos cálculos al procesador de fragmentos.

En el procesador de vértices dejaremos únicamente la aplicación de la matriz MVP a la posición del vértice y la transformación que sufre la normal en cada vértice. El parámetro de salida del procesador de vértices será ahora la

normal transformada normal_T (en lugar del color final). El resto de las operaciones deben desplazarse al programa de fragmentos, cuya variable de entrada será ahora normal_T (la salida del procesador de vértices).

Cambiar los parámetros de salida del "vertex shader" / entrada del "fragment shader" (ahora es normal_T en vez de color). Llevar los cálculos de la normalización de normal_T, el producto escalar de la componente difusa y la asignación de lux ambiente/luz difusa al procesador de fragmentos (junto con las variables auxiliares usadas en esos cálculos).

Como no tenemos mucha práctica en programar el código GLSL, procurad hacer cambios pequeños y volver a ejecutar para ver si funciona. Recordad que los errores de compilación saldrán por la consola al ejecutar el programa, no al compilar el proyecto. Una vez que os funciones, adjuntar el código GLSL modificado y una captura del resultado usando ambas iluminaciones con el modelo de la esfera que tiene 106 vértices. Incluso con pocos vértices la iluminación es ahora mucho más suave. ¿Dónde se sigue notando la poca resolución de este modelo?

Adjuntar código fuente como ejer12.cpp

Ejer 2: Partir del código (ejer12.cpp) del ejercicio anterior. En el código GLSL podéis ver como la variable **lightdir** está declarada como una constante (const), lo que implica que la dirección desde la que viene la luz está fijada y no se podía cambiar.

La dirección de una luz lejana se suele definir a través de dos ángulos: su elevación y azimut. El primero gobierna si la luz viene de arriba o abajo y el segundo el "lado" (si viene de la derecha, izquierda, de frente, etc.). Nuestro objetivo es poder cambiar la dirección de la luz desde la aplicación, a través de las teclas de los cursores. Para ello deberemos:

- Definir un par de variables globales az y elev en la aplicación (de tipo float) e inicializarlas como az=0, elev=π/4.

- Escribir una función para que cuando pulsemos las teclas de los cursores (GLUT_KEY_UP, GLUT_KEY_DOWN, etc.) se modifiquen dichas variables en saltos de +/- 0.02 (rads).

- En la función render_scene() declarar una variable (tipo vec3) donde se guardará el vector unitario de la dirección de la luz, cuyas componentes, en función de los ángulos elev y az son:

(cos(elev)*cos(az) , sin(elev), cos(elev)*sin(az))

Si corréis el nuevo código no veréis ninguna diferencia, porque el programa en la GPU que se ocupa de la iluminación no se ha enterado de nada de lo que hemos hecho en la aplicación. Para que todo esto tenga algún efecto debéis:

- En el código GLSL cambiar la declaración de lightdir de const a uniform. De esta forma se podrá modificar desde la aplicación. Hacedlo en los dos programas (implementaciones de gourad y phong).

- En render_scene(), ANTES de las ordenes de dibujo, usar la función transfer_vec3() para transferir el vector con la dirección de la luz desde la aplicación a la variable "lightdir" del programa de la GPU.

Ejecutar el programa (con iluminación de Phong), mover la luz, y adjuntar una captura de la imagen con la luz viniendo por la izquierda, ligeramente desde abajo.

Adjuntar código fuente como ejer12.cpp (incluyendo el ejercicio 1 y 2)

Ejer 3: Partir del código del ejercicio anterior (con la capacidad de alternar entre ambas iluminaciones y de mover la luz). Hacer los siguientes cambios:

- En init_scene, cargar el modelo cubo_a.bix en vez de la esfera.
- Cambiar la ubicación de la cámara (campos) a la posición (0,1,3)

Mirar en la ventana de comandos la información sobre el número de vértices y triángulos del modelo cargado. ¿Cuántos son? ¿Corresponden a un cubo?

Ejecutarlo y elegir (F1) la implementación de Phong para visualizarlo.
Adjuntar una captura de pantalla usando iluminación de Phong. ¿Os parece correcto lo que se ve? ¿Es uniforme la iluminación para una misma cara en un instante dado (como debería ser)?

Cambiar ahora el modelo por el contenido en el fichero "cubo_b.bix".
¿Se visualiza ahora correctamente el cubo?

Observar como ahora la iluminación si que es uniforme en cada cara. Esto es lo correcto porque las caras son planas y en un momento dado el ángulo con la luz en todos sus puntos es el mismo. Alternar entre Phong y Gourad, ¿hay diferencias entre ambas implementaciones?

Sin embargo nosotros sabemos que en Phong el color es calculado para cada fragmento a partir del correspondiente vector normal interpolado. ¿Cómo es qué no vemos diferencias dentro de una cara si cada fragmento deber estar recibiendo un vector normal ligeramente distinto (interpolado a partir de los datos de las normales en los vértices)? Nota: Mirar la siguiente pregunta por si os da alguna idea.

¿Cuantos vértices tenía el cubo de antes? ¿De cuántos vértices se compone ahora nuestro modelo del cubo? (mirarlo en la consola, en la información volcada al cargar el fichero).

¿Por qué no podía funcionar bien el modelo del primer cubo? ¿Cuál es el truco para que funcione correctamente el 2º modelo?

Nota: pensar en el problema que supone representar correctamente un cubo trabajando en un sistema que solo puede asignar normales a los vértices.