

# Proyecto 01

## Motor Grafica

### Computación Gráfico

#### UNAM 2022-2

Gibran Zazueta Cruz

24/marzo/2022

## 1. Introducción

El programa que se presenta a continuación implementa el renderizado de una malla 3d utilizando un método de rasterización.

Se implementa

- Lectura de archivos wavefront OBJ
- Proyección de vértices 3d a coordenadas del dispositivo
- Cálculo de normales
- Scan Conversion
- Sombreado de Phong y gouroud
- Texturizado sobre caras específicas

## 2. Leer malla 3d

Se toma la información de los vértices y normales del mesh desde un archivo formato OBJ. Para la lectura se utilizó una librería externa.

### 2.0.1. Librería *assimp*

Para leer los archivos se utiliza la libreria de assimp. *Assimp* es una librería open source que soporta multiples formatos de geometria 3d, entre llos .OBJ. Además funciona con diversos sistemas operativos y provee una interfaz de C++.

También soporta una jerarquía de nodos para mallas, materiales, texturas y animaciones de *bones*. Para el programa de este reporte se hace uso principalmente de la función *import* de la clase *Assimp::Importer* que almacena datos es una estructura llamada *aiScene*.

### 3. Proyección en perspectiva

Para lograr la proyección se implementaron las matrices de transformación necesarias. Primero la cámara recibe su posición desde la matriz de transformación mundo-cámara.

$$\begin{bmatrix} M_c^T & M_c^T o_w \\ O^T & 1 \end{bmatrix} \quad (1)$$

Donde  $M_c^T$  son los vectores unitarios del marco de la cámara y  $o_w$  es el desplazamiento con respecto al mundo. Para las dos vistas de las cámaras utilizadas en la escena resultan las siguientes matrices:

$$M_{camara1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -25 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M_{camara2} = \begin{bmatrix} 0 & 0 & 1 & -10 \\ 0 & 1 & 0 & 5 \\ -1 & 0 & 0 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad M_{camara3} = \begin{bmatrix} 0 & 0 & 1 & -10 \\ 0 & 1 & 0 & 5 \\ -1 & 0 & 0 & -10 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Por otro lado, el espacio del dispositivo tiene las dimensiones de 400x400 px y un sistema de coordenadas como se muestra en la figura

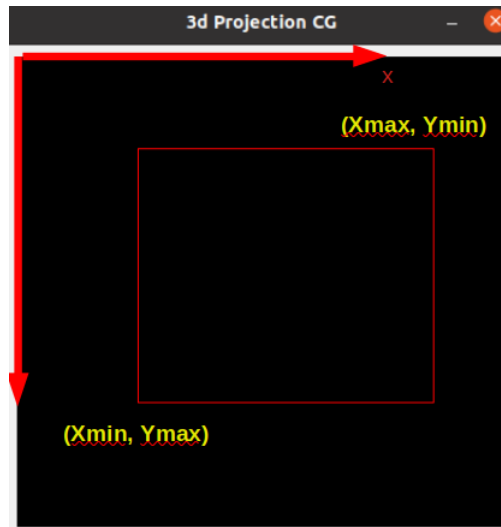


Figura 1: Ejes del plano del dispositivo

### 4. Scan Conversion

A continuación se explica el método de scan conversion

Para lograr rellenar un polígono primero se almacena la información de sus aristas en un *Buffer*.

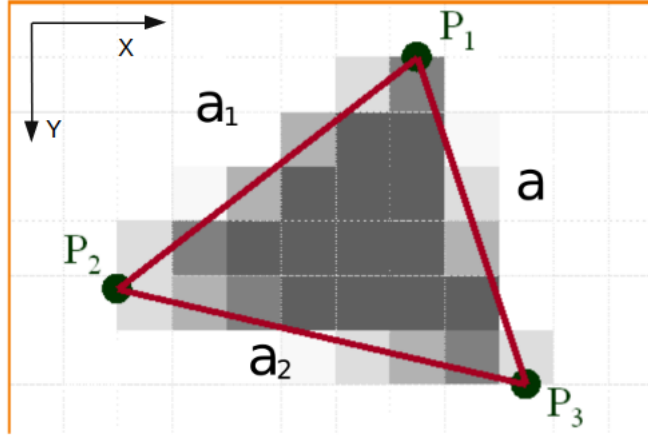


Figura 2: Algoritmo relleno de poligonos con Scan Conversion (imagen obtenida de [?])

Se siguen las aristas de la figura, una por una. Se calcula la pendiente de la recta y se va avanzando una unidad (de pixel) por  $y$ , calculando el valor correspondiente para  $x$ . Este proceso se debe realizar en orden, es decir, las aristas se siguen en orden horario o antihorario.

Tomando como ejemplo la figura 2, un procesamiento antihorario sería:  $a_1 \rightarrow a_2 \rightarrow a_3$ .

La idea es dividir las aristas del polígono entre un 'lado izquierdo' y un 'lado derecho' que permita colorear los pixeles desde un borde hasta el otro, por cada coordenada  $y$ . Con un procesamiento de este tipo y para un polígono convexo el buffer almacenará 2 coordenadas de  $x$  por cada coordenada de  $y$ .

La manera en que se clasifican los bordes, entra parte derecha e izquierda, es tomando en cuenta el valor menor entre los dos vértices que forman la arista (por ejemplo  $P_{1y}$  y  $P_{2y}$  para la arista  $a_1$ ).

Para un recorrido antihorario, mientras el valor del primer punto de la arista sea menor al del segundo se considera que esta pertenece al lado izquierdo. Cuando el valor del segundo punto de la arista sea menor al primero, la recta pertenece al lado derecho.

La información de las coordenadas  $x$  (derecha e izquierda) correspondientes a cada coordenada  $y$  se almacena en el buffer.

Una vez se tiene la información del buffer, se recorre la figura de arriba hacia abajo (tomando en cuenta el eje de coordenadas del dispositivo, esto corresponde a un incremento en  $y$ ), donde por cada valor de  $y$  se colorean los pixeles desde la coordenada  $x$  derecha hasta la  $x$  izquierda.

Esta implementación está basada en la información presentada en [1]

## 5. Shading

Para el sombreado se utilizan los métodos de sombreado de Gouroud y Phong

El modelo de iluminación de Phong presenta las siguientes ecuaciones de sus componentes ambiental, difusa y especular.

*Ambiental*

Se calcula como

$$I_A = \kappa_A \Lambda_A \quad (2)$$

La luz *difusa* se calcula como

$$I_D = \kappa_D \Lambda_D(\vec{n} \cdot \vec{l}) \quad (3)$$

*Especular* El calculo es el siguiente

$$I_E = \kappa_E \Lambda_E(\vec{o} \cdot \vec{l})^\rho \quad (4)$$

donde  $\vec{o}$  es el vector de dirección del observador. La ecuación (3) también se puede escribir

$$I_E = \kappa_E \Lambda_E(\vec{o} \cdot \vec{h})^\rho \quad (5)$$

donde  $h = 2(\vec{o} \cdot \vec{l})\vec{n} - \vec{l}$

## 5.1. Materiales y Escena

Los objetos cuentan con 2 posibles materiales a seleccionar, estos se definen dentro del código como:

### Material 1

- Ambiental = 0.0, 0.0, 0.0, 1.0,
- Difusa = 0.50, 0.50, 0.50, 1.0,
- Especular 0.70, 0.70, 0.70, 1.0
- $\rho = 32.0$ .

### Material 2

- Ambiental = 0.23125, 0.23125, 0.23125, 1.0,
- Difusa = 0.2775, 0.2775, 0.2775, 1.0,
- Especular 0.773911, 0.773911, 0.773911, 1.0
- $\rho = 89.6$ .

Finalmente, la escena a renderizar cuenta con 2 luces (blanca y azul) y 3 cámaras. Se disponen de la siguiente manera:

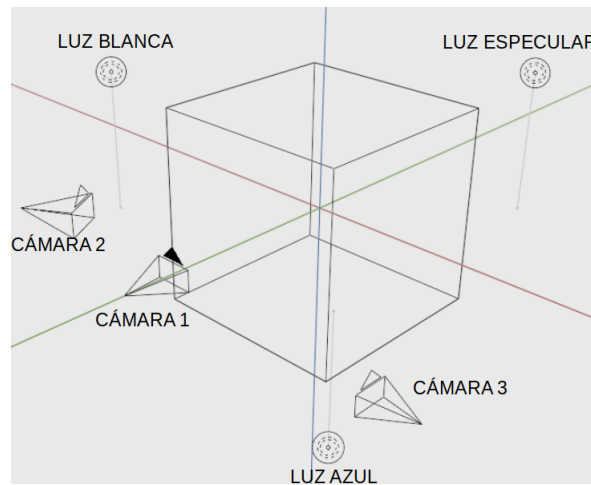


Figura 3: Posición de las componentes de la escena

## 6. Estructura del código

### 6.1. CubeObject

Almacena la información del objeto a renderizar. Esto incluye coordenadas de los vertices, definición de las caras del polígono (por medio de índices a los vertices), normales de las caras, normales de los vértices, coordenadas UV e información sobre el material (coeficientes  $k_a, k_d, k_e$ )

En el constructor de la clase se definen los 2 materiales a utilizar. En los materiales también se puede cargar una textura, correspondiente a ese material.

Esta clase también tiene métodos para realizar rotación de euler en  $x$ ,  $y$  y  $z$ , y para calcular las normales de las caras y de los vértices. La información de las normales también se puede almacenar desde un archivo OBJ

### 6.2. raster

*raster* es la clase más importante ya que en su método pipeline se realiza la proyección, scan conversion y sombreado

Para la proyección se utiliza la clase ***camProjection***. En el método *projectPoint* de esta clase se realiza proyección en perspectiva u ortogonal, además se mapea a coordenadas del dispositivo. De este método se obtienen estas coordenadas e información de profundidad

Después se calcula en los vértices los valores para los vectores de normales ( $N$ ), posición de luz ( $L$ ) y observador ( $O$ ). Con estos se pueden calcular el valor de color en los vértices (útil para realizar la interpolación de sombreado de gouroud), o se pueden interpolar para, posteriormente, obtener el sombreado de Phong.

A continuación, con el método *fillCubeFace*, se utiliza el algoritmo de scan conversion descrito anteriormente para rellenar los pixeles del polígono y a su vez para interpolar  $N, L, O$ , Color, profundidad y coordenadas de textura

la primera interpolación sobre los bordes se realiza con el método *scanline*. Se busca aplicar scanline entre todos los vértices del polígono. Posteriormente se realiza la interpolación horizontal, para ello se utilizan las funciones *scanConversion* y *horInterpolation*

EN el caso de que se requiera dibujar textura, es en este método que se recupera la información de la imagen solo para las caras que así lo requieran

### 6.3. lights

La clase *lights* almacena la información de las luces de la escena. Se define su intensidad, color y posición

### 6.4. renderwindow

Se utiliza la biblioteca de **qt** para pintar sobre el canvas. Al dibujar un pixel se toma en cuenta la profundidad interpolada (o mas bien el inverso  $1/z$ ). El punto solo se dibujará en el canvas si no existe otro pixel en la misma posición con profundidad menor (es decir, un pixel que ocuya al anterior)

### 6.5. mainWindow

Desde *mainwindow.cpp* se llama a la función *importFile()* que está definida en *functions.cpp*. La función recibe el path del archivo (como una cadena `std::string`) y apunadores al contenedor de vertices y caras del objeto *cubeObject*,

Después de esto, en *mainwindow* se crean los objetos de clase *light* para las luces blanca, azul y especular, se especifican intensidad, color y posicionamiento. También se crea el grid de botones que conforma la GUI de la aplicación. Finalmente se llama a la función *drawObject()*

En la función *drawObject()* se llama a la función pipeline del objeto raster, que como ya se mencionó relaja todo lo referente a la proyección, relleno y sombreado.

## 7. Ejecutar el programa

En la carpeta de build se puede ejecutar el programa con el archivo *GraphicsEngine-Run*. Desde la consola de comandos de linux:

```
bash
```

```
./GraphicsEngine-Run
```

En la carpeta principal está el código fuente. Para generar el ejecutable primero se genera el Makefile con

```
bash
```

```
qmake GraphicsEngine.pro
```

Después se construye el proyecto con *make*

## 8. Instrucciones de uso

Se presenta la interfaz del programa.

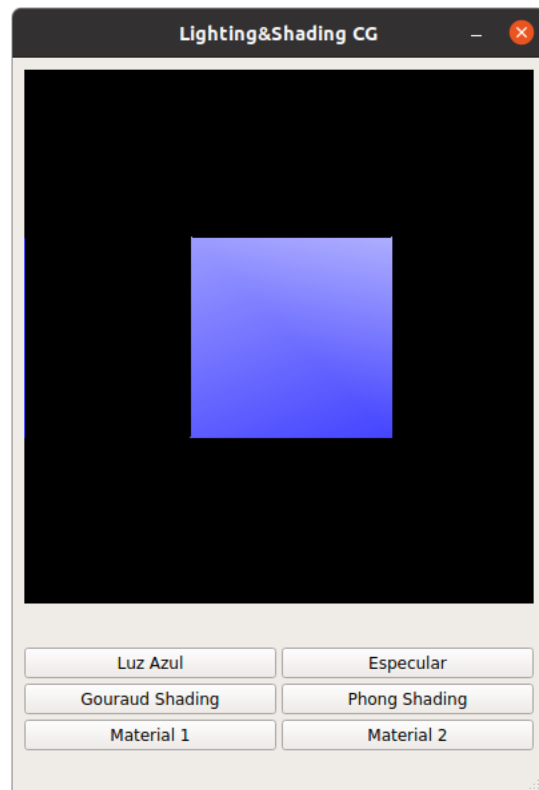


Figura 4: Interfaz gráfica del programa

Para cambiar entre las camaras se utilizan las teclas de los numeros

- "1". Cambia a la cámara 1
- "2". Cambia a la cámara 2
- "3". Cambia a la cámara 3

Se puede rotar el objeto 45 sobre el eje  $y$ . Para hacerlo se presiona la tecla R del teclado.

Los botones *Luz Azul* y *Especular* encienden o apagan las luces correspondientes. El programa inicia con todas las luces activas.

*Gouraud Shading* activa el Sombreado por Gouraud y *Phong Shading* el sombreado de Phong.

Es posible cambiar entre archivos desde el código. En la variable path de mainwindow se cambia la extension por el archivo correspondiente. Por defecto es *bunny*, los demás archivos se encuentra en la carpeta '*objectfile*' del proyecto.

## 9. Programa en ejecución

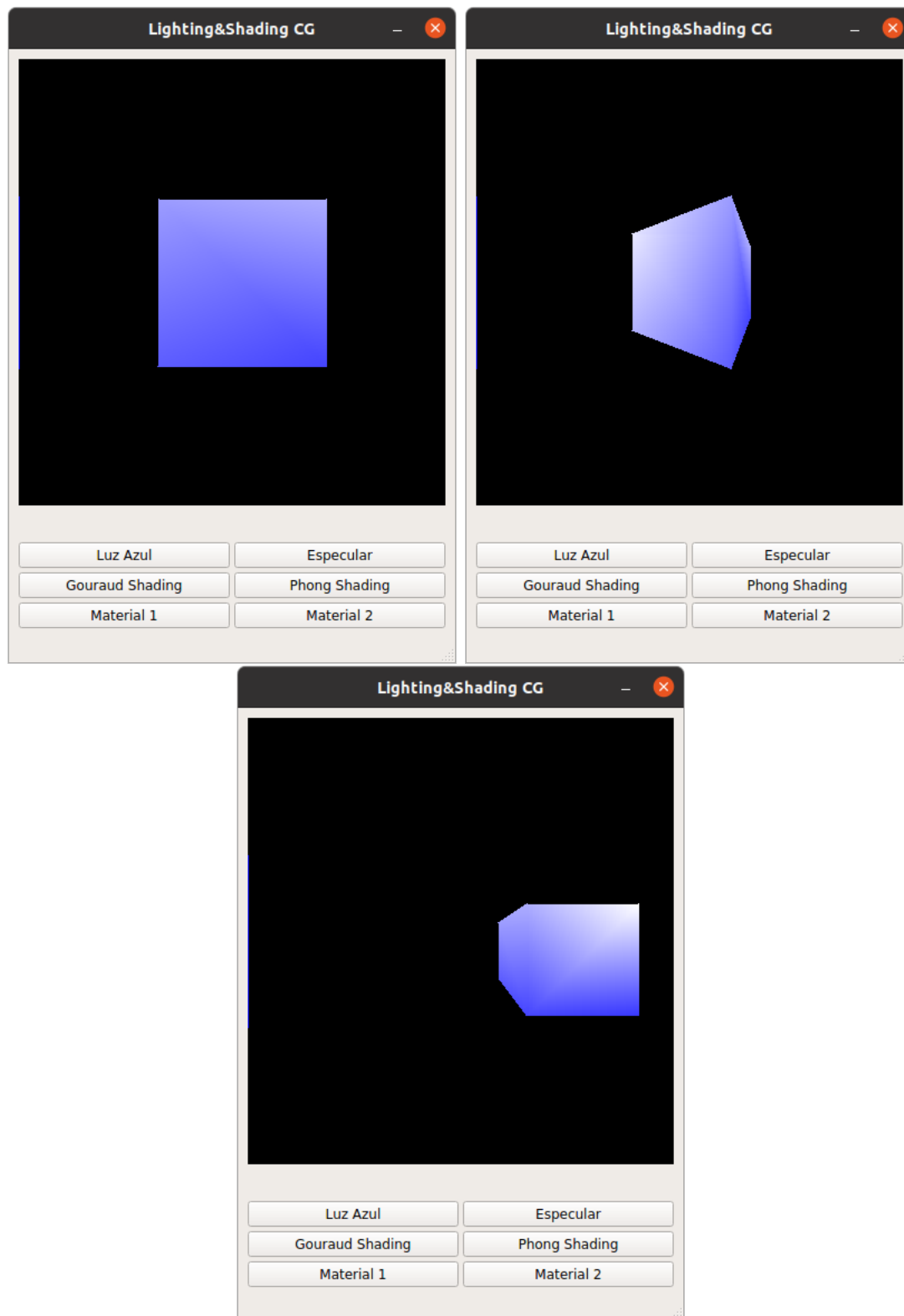


Figura 5: Gouraud Shading. Cámara 1, 2 y 3. Material 1



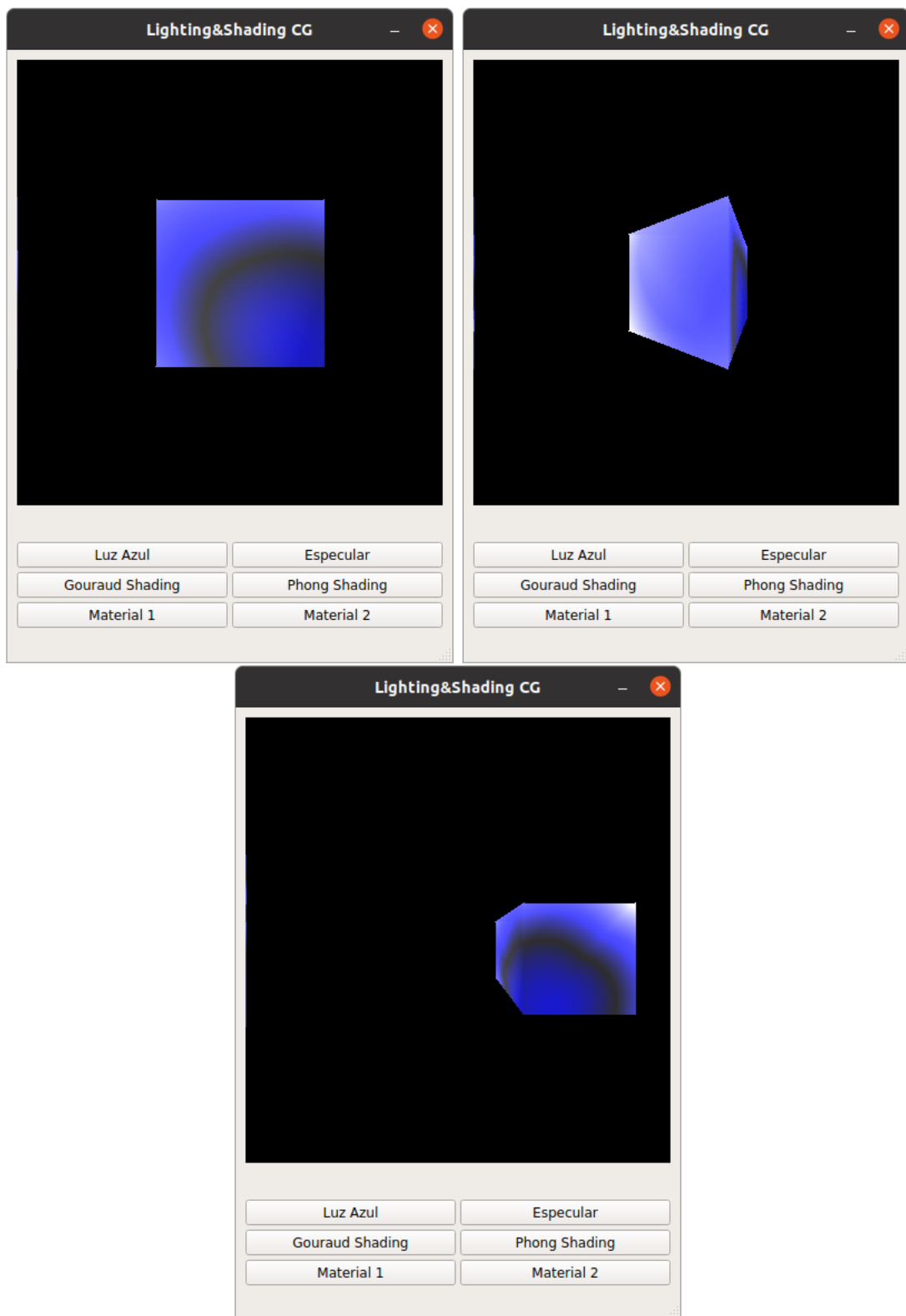


Figura 6: Phong Shading. Cámara 1, 2 y 3. Material 2

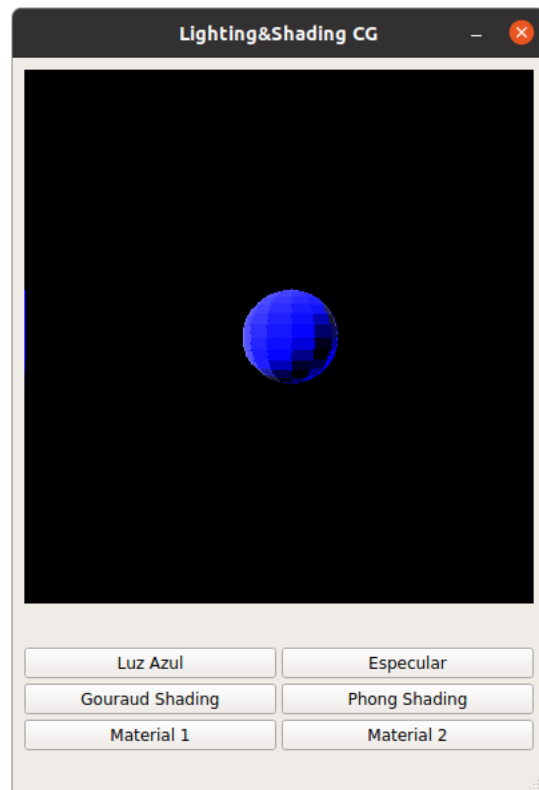


Figura 7: Renderizado esfera

## Referencias

- [1] Upssala Universit. Introduction to polygons. ([http : //www.it.uu.se/edu/course/homepage/grafik1/ht06/Lectures/L02/LinePolygon/xpolyd.htm](http://www.it.uu.se/edu/course/homepage/grafik1/ht06/Lectures/L02/LinePolygon/xpolyd.htm))