**Gráficos por Ordenador**

**Proyecto de la Asignatura - NPRs**

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

***Autores:***

Eric García Arribas

Rubén Llorente Canet

Lucas Martín Gil-Delgado

Índice

[Introducción 1](#_Toc167562377)

[Modelo de iluminación de Phong 1](#_Toc167562378)

[Modelo de iluminación de Blinn-Phong 4](#_Toc167562379)

[Resaltado de la silueta 8](#_Toc167562380)

[Sombreado Toon 11](#_Toc167562381)

[Stippling 14](#_Toc167562382)

[Painterly Rendering 20](#_Toc167562383)

[Pixel rendering 24](#_Toc167562384)

[Ejecución del código 28](#_Toc167562385)

[Carga de modelos .obj 29](#_Toc167562386)

## Introducción

Esta es la memoria del Proyecto 2024 Modelos de iluminación NPR (Non Photorealistic Rendering) de Grafico por Ordenador. En esta se describirán las bases sobre las que nos hemos basado a la hora de construir estos distintos modelos de iluminación (los indicados en el enunciado, Phong, Blinn-Phong, Resaltado de silueta y Toon Shading, además de los escogidos por nuestro grupo que son Stippling, Painterly Rendering y Píxel Rendering). Además, se incluye una descripción sobre la ejecución del programa y las distintas funcionalidades que se pueden llegar a hacer en la visualización como el cambio de modelo o el giro de cámara. Finalmente se mencionará brevemente como hemos logrado cargar objetos con formato .obj.

## Modelo de iluminación de Phong

### Procesado en CPU y GPU

El proceso del programa del modelo de iluminación Phong comienza en la CPU con la carga del modelo 3D, donde se importan los datos de vértices, normales y texturas en la memoria. Las texturas también se cargan desde el disco y se almacenan en la memoria de la CPU. Además, se compilan los shaders y se configuran las variables uniformes necesarias. En la GPU, el Vertex Shader transforma las posiciones de los vértices utilizando las matrices **M** y **PV**, ajusta las normales del modelo y calcula el vector desde el vértice hacia la cámara. En el Fragment Shader, las normales interpoladas se normalizan y se calcula la iluminación difusa y especular usando el modelo de Phong. Finalmente, se aplica la textura y se multiplica por la iluminación total.

### Especificación del modelo

El modelo de iluminación de Phong es un modelo empírico que se utiliza en la programación de gráficos 3D para simular cómo se comporta la luz al iluminar la superficie de un objeto tridimensional.

El modelo de iluminación Phong se basa en tres componentes básicos:

1. Componente ambiental: Luz que viene rebotada de todas las direcciones e ilumina todas las caras del objeto y se calcula como A=Ka \* Ca, siendo Ka el factor de luz ambiente y Ca el color de la luz ambiente
2. Componente difusa: Luz que llega directamente desde la fuente de luz, pero rebota en todas direcciones. Se calcula como D= Kd\*Cl\*cos(θ) donde Kd es el factor de luz difusa, Cl el color de la luz y cos(θ) es igual a max(0,N\*L) siendo N la normal de la superficie y L el vector de dirección de donde viene la luz.
3. Componente especular: La luz que llega directamente de la fuente de luz y rebota en una sola dirección, según la normal de la superficie. Se calcula como S= Ks\*Cl\*cos(α)n donde Ks es el factor de luz especular, Cl el color de la luz, n el coeficiente de brillo y cos(α)n es igual a max(0,R\*V)n siendo R el vector de la luz reflejada y V el vector que va desde la superficie del objeto hacia el observador.

Para calcular la iluminación final de Phong basta con calcular la suma de las 3 componentes I=A+D+S.

Trasladado a código se vería de la siguiente manera:  
// Vertex Shader de Phong

const char**\*** vertex\_prog1 **=** GLSL**(**

layout**(**location **=** 0**)** in vec3 pos**;** // Posición del vértice

layout**(**location **=** 1**)** in vec3 normal**;** // Normal del vértice

layout**(**location **=** 2**)** in vec2 uv**;** // Coordenadas UV del vértice

layout**(**location **=** 3**)** in float texIndex**;** // Índice de la textura

out vec3 n**;** // Normal transformada

out vec2 UV**;** // Coordenadas UV a pasar al fragment shader

out vec3 v**;** // Vector desde el vértice a la cámara

flat out int TexIndex**;** // Índice de la textura a pasar al fragment shader

uniform vec3 campos**;** // Posición de la cámara

uniform mat4 PV**;** // Matriz de proyección y vista

uniform mat4 M**;** // Matriz de modelo

void main**()** **{**

gl\_Position **=** PV **\*** M **\*** vec4**(**pos**,** 1**);** // Transformación del vértice

mat3 M\_adj **=** mat3**(**transpose**(**inverse**(**M**)));** // Matriz de adjunta de la matriz de modelo

n **=** M\_adj **\*** normal**;** // Transformación de la normal

vec4 vertex\_position\_scene **=** M **\*** vec4**(**pos**,** 1.0**);**

vec3 pos\_scene **=** vec3**(**vertex\_position\_scene**);** // Posición del vértice en el espacio de la escena

v **=** normalize**(**campos **-** pos\_scene**);** // Vector desde el vértice a la cámara

UV **=** uv**;** // Pasar coordenadas UV

TexIndex **=** int**(**texIndex**);** // Convertir índice de textura de float a int

**}**

**);**

// Phong - Fragment Shader

const char**\*** fragment\_prog1 **=** GLSL**(**

in vec3 n**;** // Normal interpolada

in vec3 v**;** // Vector desde el vértice a la cámara interpolado

in vec2 UV**;** // Coordenadas UV interpoladas

flat in int TexIndex**;** // Índice de la textura

float ilu**;** // Factor de iluminación

uniform vec3 luz **=** vec3**(**1**,** 1**,** 0**)** **/** sqrt**(**2.0f**);** // Dirección de la luz

uniform sampler2D textures**[**16**];** // Array de texturas

void main**()** **{**

vec3 nn **=** normalize**(**n**);** // Normalizar la normal

vec3 r **=** reflect**(-**luz**,** nn**);** // Calcular el vector reflejado

float spec\_intensity **=** max**(**dot**(**r**,** v**),** 0.0**);** // Intensidad especular

float spec\_component **=** pow**(**spec\_intensity**,** 16.0**);** // Componente especular

float difusa **=** max**(**dot**(**luz**,** nn**),** 0.0**);** // Componente difusa

ilu **=** **(**0.1 **+** 0.6 **\*** difusa **+** 0.3 **\*** spec\_component**);** // Suma de componentes de iluminación

gl\_FragColor **=** texture**(**textures**[**TexIndex**],** UV**);** // Aplicar textura

gl\_FragColor **=** gl\_FragColor **\*** ilu**;** // Modificar color por factor de iluminación

**}**

**);**

Y nos daría resultados como estos:

Imagen que contiene parado

Descripción generada automáticamente

Imagen que contiene verde, tablero, montar a caballo, colorido

Descripción generada automáticamente

Dibujo animado de un personaje animado

Descripción generada automáticamente con confianza baja

## Modelo de iluminación de Blinn-Phong

### Procesado en CPU y GPU

En la implementación del modelo de iluminación Blinn-Phong, la CPU se encarga de nuevo de cargar el modelo 3D y las texturas en la memoria, además de compilar los shaders y configurar las variables uniformes necesarias. El proceso será similar para los siguientes shaders con respecto a lo que se prepare en el lado de la CPU. En la GPU, el Vertex Shader transforma las posiciones de los vértices, ajusta las normales y calcula el vector desde el vértice hacia la cámara, de manera similar al Phong Shader. En el Fragment Shader, se utiliza el modelo Blinn-Phong para calcular la iluminación, empleando el vector **halfVector** para la componente especular. La textura se aplica de manera similar, pero con ajustes específicos para la iluminación especular.

### Especificación del modelo

El modelo de iluminación de Blinn-Phong es una variante del modelo de iluminación de Phong. La principal diferencia entre el modelo de Phong y el de Blinn-Phong radica en cómo se calcula la reflexión especular. En lugar de usar el vector de reflexión R en el modelo de Phong, el modelo de Blinn-Phong introduce el concepto de un vector medio H, que se calcula como la mitad del vector de reflexión de esta forma: H = (L + V) / |L + V| aproximado como (L + V) / 2

El problema con el modelo de Phong es que el ángulo entre el vector de vista y el de reflexión tiene que ser menor que 90º, para ello el modelo de Blinn-Phong calcula el vector medio H para poder calcular la luz especular incluso cuando el vector es mayor de 90º.

La implementación de Blinn-Phong en código quedaría de la siguiente manera:

// Vertex Shader de Bling-Phong

const char**\*** vertex\_prog2 **=** GLSL**(**

layout**(**location **=** 0**)** in vec3 pos**;** // Posición del vértice

layout**(**location **=** 1**)** in vec3 normal**;** // Normal del vértice

layout**(**location **=** 2**)** in vec2 uv**;** // Coordenadas UV del vértice

layout**(**location **=** 3**)** in float texIndex**;** // Índice de la textura

out vec3 n**;** // Normal transformada

out vec2 UV**;** // Coordenadas UV a pasar al fragment shader

out vec3 v**;** // Vector desde el vértice a la cámara

flat out int TexIndex**;** // Índice de la textura a pasar al fragment shader

uniform vec3 campos**;** // Posición de la cámara

uniform mat4 M**;** // Matriz de modelo

uniform mat4 PV**;** // Matriz de proyección y vista

void main**()** **{**

gl\_Position **=** PV **\*** M **\*** vec4**(**pos**,** 1**);** // Transformación del vértice

mat3 M\_adj **=** mat3**(**transpose**(**inverse**(**M**)));** // Matriz adjunta de la matriz de modelo

n **=** M\_adj **\*** normal**;** // Transformación de la normal

vec4 vertex\_position\_scene **=** M **\*** vec4**(**pos**,** 1.0**);**

vec3 pos\_scene **=** vec3**(**vertex\_position\_scene**);** // Posición del vértice en el espacio de la escena

v **=** normalize**(**campos **-** pos\_scene**);** // Vector desde el vértice a la cámara

UV **=** uv**;** // Pasar coordenadas UV

TexIndex **=** int**(**texIndex**);** // Convertir índice de textura de float a int

**}**

**);**

// Bling-Phong - Fragment Shader

const char**\*** fragment\_prog2 **=** GLSL**(**

in vec3 n**;** // Normal interpolada

in vec3 v**;** // Vector desde el vértice a la cámara interpolado

in vec2 UV**;** // Coordenadas UV interpoladas

flat in int TexIndex**;** // Índice de la textura

float ilu**;** // Factor de iluminación

uniform vec3 luz **=** vec3**(**1**,** 1**,** 0**)** **/** sqrt**(**2.0f**);** // Dirección de la luz

uniform sampler2D textures**[**16**];** // Array de texturas

vec3 halfVector**;** // Half-angle vector

void main**()** **{**

halfVector **=** normalize**(**luz **+** v**);** // Calcular el vector de mitad de ángulo

vec3 nn **=** normalize**(**n**);** // Normalizar la normal

float spec\_intensity **=** max**(**dot**(**nn**,** halfVector**),** 0.0**);** // Intensidad especular

float spec\_component **=** pow**(**spec\_intensity**,** 16.0**);** // Componente especular

float difusa **=** dot**(**luz**,** nn**);** // Componente difusa

**if** **(**difusa **<** 0**)** difusa **=** 0**;**

ilu **=** **(**0.1 **+** 0.6 **\*** difusa **+** 0.3 **\*** spec\_component**);** // Suma de componentes de iluminación

gl\_FragColor **=** texture**(**textures**[**TexIndex**],** UV**);** // Aplicar textura

gl\_FragColor **=** gl\_FragColor **\*** ilu**;** // Modificar color por factor de iluminación

**}**

**);**

Y nos daría resultados como estos:



Una caricatura de una persona

Descripción generada automáticamente con confianza baja

Imagen que contiene juguete, lego, verde, colorido

Descripción generada automáticamente

## Resaltado de la silueta

### Procesado en CPU y GPU

En este programa, en la GPU, el Vertex Shader transforma las posiciones de los vértices, ajusta las normales y calcula el vector desde el vértice hacia la cámara. En el Fragment Shader, se calcula el ángulo entre la normal y la dirección de vista. Si el ángulo está en un rango específico, se dibuja la silueta en negro; de lo contrario, se aplica la textura, logrando así el efecto de resaltado de silueta.

### Especificación del modelo

El resaltado de silueta es una técnica utilizada para enfatizar los bordes o contornos de un objeto tridimensional. El resaltado de silueta se logra generalmente dibujando líneas a lo largo de los bordes de un objeto.

Existen múltiples técnicas para lograrlo, pero nosotros hemos optado por utilizar el vector V y N para dibujar de negro las superficies que generen un ángulo de reflexión entre 75º y 100º. Este ángulo se consigue mediante la operación |VxN|.

La implementación del resaltado de la silueta en código quedaría de la siguiente manera:

// Vertex Shader de Resaltado de Silueta

const char**\*** vertex\_prog3 **=** GLSL**(**

layout**(**location **=** 0**)** in vec3 pos**;** // Posición del vértice

layout**(**location **=** 1**)** in vec3 normal**;** // Normal del vértice

layout**(**location **=** 2**)** in vec2 uv**;** // Coordenadas UV del vértice

layout**(**location **=** 3**)** in float texIndex**;** // Índice de la textura

out vec3 n**;** // Normal transformada

out vec2 UV**;** // Coordenadas UV a pasar al fragment shader

out vec3 v**;** // Vector desde el vértice a la cámara

flat out int TexIndex**;** // Índice de la textura a pasar al fragment shader

uniform vec3 campos**;** // Posición de la cámara

uniform mat4 M**;** // Matriz de modelo

uniform mat4 PV**;** // Matriz de proyección y vista

void main**()** **{**

gl\_Position **=** PV **\*** M **\*** vec4**(**pos**,** 1**);** // Transformación del vértice

mat3 M\_adj **=** mat3**(**transpose**(**inverse**(**M**)));** // Matriz adjunta de la matriz de modelo

n **=** M\_adj **\*** normal**;** // Transformación de la normal

vec4 vertex\_position\_scene **=** M **\*** vec4**(**pos**,** 1.0**);**

vec3 pos\_scene **=** vec3**(**vertex\_position\_scene**);** // Posición del vértice en el espacio de la escena

v **=** normalize**(**campos **-** pos\_scene**);** // Vector desde el vértice a la cámara

UV **=** uv**;** // Pasar coordenadas UV

TexIndex **=** int**(**texIndex**);** // Convertir índice de textura de float a int

**}**

**);**

// Resaltado de Silueta - Fragment Shader

const char**\*** fragment\_prog3 **=** GLSL**(**

in vec3 n**;** // Normal interpolada

in vec3 v**;** // Vector desde el vértice a la cámara interpolado

in vec2 UV**;** // Coordenadas UV interpoladas

flat in int TexIndex**;** // Índice de la textura

uniform sampler2D textures**[**16**];** // Array de texturas

void main**()** **{**

vec3 nn **=** normalize**(**n**);** // Normalizar la normal

float angle **=** degrees**(**acos**(**max**(**dot**(**nn**,** v**),** 0.0**)));** // Calcular el ángulo entre la normal y la vista

**if** **(**angle **>=** 75.0 **&&** angle **<=** 100.0**)** **{**

gl\_FragColor **=** vec4**(**0.0**,** 0.0**,** 0.0**,** 1.0**);** // Silueta negra

**}**

**else** **{**

gl\_FragColor **=** texture**(**textures**[**TexIndex**],** UV**);** // Aplicar textura

**}**

**}**

**);**

Y nos daría resultados como estos:



Imagen que contiene café, tablero

Descripción generada automáticamente

Diagrama

Descripción generada automáticamente

## Sombreado Toon

### Procesado en CPU y GPU

Para el sombreado Toon (o Toon Shading), en la GPU, el Vertex Shader transforma las posiciones de los vértices, ajusta las normales, calcula el vector desde el vértice hacia la cámara y la posición del vértice en la escena. En el Fragment Shader, se calcula la iluminación difusa y especular, y se aplica una cuantización para lograr el efecto de sombreado toon. La textura se aplica y se ajusta la iluminación para crear el efecto de sombreado característico.

### Especificación del modelo

El sombreado Toon es un estilo que se utiliza comúnmente para imitar el estilo de los cómics o dibujos animados. Se caracteriza por una reducción del posible conjunto de valores que puedan tomar los colores a lo que se le llama como “niveles de color”. El único cambio necesario para lograr este efecto con respecto a Blinn-Phong es establecer estos “niveles de color” con una división de la luz especular y la difusa entre el numero de niveles deseado. Esto consigue que, en vez de un cambio gradual de la luz a lo largo de la superficie, solo están X numero de umbrales posibles que serán el número de niveles.

La implementación del Sombreado Toon en código quedaría de la siguiente manera:

// Vertex Shader de Toon Shading

const char**\*** vertex\_prog4 **=** GLSL**(**

layout**(**location **=** 0**)** in vec3 pos**;** // Posición del vértice

layout**(**location **=** 1**)** in vec3 normal**;** // Normal del vértice

layout**(**location **=** 2**)** in vec2 uv**;** // Coordenadas UV del vértice

layout**(**location **=** 3**)** in float texIndex**;** // Índice de la textura

out vec3 n**;** // Normal transformada

out vec2 UV**;** // Coordenadas UV a pasar al fragment shader

out vec3 pos\_scene**;** // Posición del vértice en el espacio de la escena

out vec3 v**;** // Vector desde el vértice a la cámara

flat out int TexIndex**;** // Índice de la textura a pasar al fragment shader

uniform vec3 campos**;** // Posición de la cámara

uniform mat4 M**;** // Matriz de modelo

uniform mat4 PV**;** // Matriz de proyección y vista

void main**()** **{**

vec4 worldPosition **=** M **\*** vec4**(**pos**,** 1.0**);**

gl\_Position **=** PV **\*** worldPosition**;** // Transformación del vértice

mat3 M\_adj **=** mat3**(**transpose**(**inverse**(**M**)));** // Matriz adjunta de la matriz de modelo

n **=** M\_adj **\*** normal**;** // Transformación de la normal

pos\_scene **=** worldPosition**.**xyz**;** // Posición del vértice en el espacio de la escena

v **=** normalize**(**campos **-** pos\_scene**);** // Vector desde el vértice a la cámara

UV **=** uv**;** // Pasar coordenadas UV

TexIndex **=** int**(**texIndex**);** // Convertir índice de textura de float a int

**}**

**);**

// Toon Shading - Fragment Shader

const char**\*** fragment\_prog4 **=** GLSL**(**

in vec3 n**;** // Normal interpolada

in vec3 v**;** // Vector desde el vértice a la cámara interpolado

in vec2 UV**;** // Coordenadas UV interpoladas

in vec3 pos\_scene**;** // Posición del vértice en el espacio de la escena interpolada

flat in int TexIndex**;** // Índice de la textura

float ilu**;** // Factor de iluminación

uniform vec3 luz **=** vec3**(**1**,** 1**,** 0**)** **/** sqrt**(**2.0f**);** // Dirección de la luz

uniform sampler2D textures**[**16**];** // Array de texturas

void main**()** **{**

vec3 nn **=** normalize**(**n**);** // Normalizar la normal

vec3 luz\_final **=** normalize**(**luz **-** pos\_scene**);** // Normalizar la luz final

vec3 v\_final **=** normalize**(**v**);** // Normalizar la vista final

vec3 r **=** reflect**(-**luz**,** nn**);** // Calcular el vector reflejado

float spec\_intensity **=** max**(**dot**(**r**,** v\_final**),** 0.0**);** // Intensidad especular

float spec **=** pow**(**spec\_intensity**,** 16.0**);** // Componente especular

spec **=** floor**(**spec **\*** 3.0**)** **/** 3.0**;** // Ajuste de la especular

float difusa\_aux **=** max**(**dot**(**luz**,** nn**),** 0.0**);** // Componente difusa auxiliar

difusa\_aux **=** floor**(**difusa\_aux **\*** 5.0**)** **/** 5.0**;** // Ajuste de la difusa auxiliar

vec3 baseColor **=** texture**(**textures**[**TexIndex**],** UV**).**rgb**;** // Color base

vec3 ambiente **=** 0.1 **\*** baseColor**;** // Color ambiente

vec3 difusa **=** difusa\_aux **\*** baseColor**;** // Color difuso

vec3 specular **=** baseColor **\*** spec**;** // Color especular

gl\_FragColor **=** vec4**(**ambiente **+** difusa **+** specular**,** 1.0**);** // Color final

**}**

**);**

Y nos daría resultados como estos:

Imagen que contiene sostener, hombre

Descripción generada automáticamente



Imagen que contiene juguete, verde, montar a caballo, lego

Descripción generada automáticamente

## Stippling

### Procesado en CPU y GPU

Para este programa del proceso de Stippling, el Vertex Shader transforma las posiciones de los vértices, ajusta las normales y calcula el vector desde el vértice hacia la cámara. En el Fragment Shader, se calcula la iluminación difusa y especular, y se convierte la iluminación resultante a escala de grises, habiendo implementado un modelo de iluminación Blinn-Phong sobre el shader. Luego, se aplica un patrón de punteado basado en la detección de bordes y la variación local de la textura, creando así el efecto de punteado.

### Especificación del modelo

El Stippling es una técnica que utiliza pequeños círculos o puntos del mismo color para crear una imagen reconocible. El propósito del Stippling es crear la impresión de luces y sombras en una obra de arte, ya que se puede representar un rango completo de valores con la disposición de los puntos.

Para lograrlo, primero se establece un patrón de punteado en una matriz 4x4. Después de calcular todos los tipos de iluminaciones convertimos el color base de la escena a escala de grises.

Se usa el operador de Sobel para detectar bordes en la textura. Esto implica tomar muestras de la textura en varias direcciones alrededor del fragmento actual y aplicar una matriz de Sobel para calcular un valor de detección de bordes. El resultado de la detección de bordes se invierte para que los bordes fuertes tengan valores cercanos a 1.0 y las áreas planas tengan valores cercanos a 0.0.

Se calcula la variación local en la textura tomando muestras en un área pequeña alrededor del fragmento actual (en los pixeles colindantes al actual) y comparando esos valores con la luminancia del fragmento actual. Este valor de variación se usa para ajustar el umbral del punteado.

A continuación, calculamos un umbral ajustado basado en la variación local.

Por último, para determinar el color del fragmento comprobamos si el valor de detección es menor que el del punteado multiplicado por el umbral para pintarlo de negro o, en caso contrario, de blanco.

La implementación del Stippling en código quedaría de la siguiente manera:

/\* Vertex Shader de Stippling \*/

const char**\*** vertex\_prog5 **=** GLSL**(**

layout**(**location **=** 0**)** in vec3 pos**;** // Posición del vértice

layout**(**location **=** 1**)** in vec3 normal**;** // Normal del vértice

layout**(**location **=** 2**)** in vec2 uv**;** // Coordenadas de textura

layout**(**location **=** 3**)** in float texIndex**;** // Índice de textura

out vec3 n**;** // Normal del vértice para el fragmento

out vec2 UV**;** // Coordenadas de textura para el fragmento

out vec3 v**;** // Vector de vértice a cámara para el fragmento

flat out int TexIndex**;** // Índice de textura para el fragmento (sin interpolación)

uniform vec3 campos**;** // Posición de la cámara en coordenadas del mundo

uniform mat4 M**;** // Matriz de modelo

uniform mat4 PV**;** // Matriz de proyección y vista

void main**()** **{**

vec4 worldPosition **=** M **\*** vec4**(**pos**,** 1.0**);** // Calcula la posición del vértice en el espacio del mundo

gl\_Position **=** PV **\*** worldPosition**;** // Transforma la posición del vértice a coordenadas de clip

mat3 M\_adj **=** mat3**(**transpose**(**inverse**(**M**)));** // Calcula la matriz de la inversa y transpuesta de la matriz de modelo

n **=** normalize**(**M\_adj **\*** normal**);** // Calcula la normal del vértice en el espacio del mundo

vec3 pos\_scene **=** vec3**(**worldPosition**);** // Calcula la posición del vértice en el espacio de la escena

v **=** normalize**(**campos **-** pos\_scene**);** // Calcula el vector del vértice a la cámara

UV **=** uv**;** // Pasa las coordenadas de textura al fragmento

TexIndex **=** int**(**texIndex**);** // Pasa el índice de textura al fragmento

**}**

**);**

// Stippling - Fragment Shader

const char**\*** fragment\_prog5 **=** GLSL**(**

uniform sampler2D textures**[**16**];** // Arreglo de muestreadores de texturas

uniform vec3 luz**;** // Dirección de la luz

uniform vec3 lightPos**;** // Posición de la luz

uniform float umbral **=** 1.45**;** // Umbral para el punteado

in vec3 n**;** // Vector normal

in vec3 v**;** // Vector de vista

in vec2 UV**;** // Coordenadas UV

flat in int TexIndex**;** // Índice para el arreglo de texturas

out vec4 col**;** // Color de salida

// Matriz de umbrales para el patrón de punteado

const float stipplePattern**[**16**]** **=** float**[**16**](**

0.0**,** 0.6**,** 0.3**,** 0.8**,**

1.0**,** 0.4**,** 0.9**,** 0.5**,**

0.7**,** 1.0**,** 0.2**,** 0.6**,**

0.3**,** 0.8**,** 0.4**,** 0.9

**);**

void main**()** **{**

vec3 nn **=** normalize**(**n**);** // Normalizar el vector normal

vec3 lightDir **=** normalize**(**luz **-** v**);** // Calcular la dirección de la luz

float diff **=** max**(**dot**(**nn**,** lightDir**),** 0.0**);** // Componente difusa

vec3 diffuse **=** diff **\*** texture**(**textures**[**TexIndex**],** UV**).**rgb**;** // Aplicar textura a la luz difusa

vec3 viewDir **=** normalize**(-**v**);** // Dirección de vista

vec3 reflectDir **=** reflect**(-**lightDir**,** nn**);** // Dirección de reflexión

float spec **=** pow**(**max**(**dot**(**viewDir**,** reflectDir**),** 0.0**),** 16.0**);** // Componente especular

vec3 specular **=** spec **\*** texture**(**textures**[**TexIndex**],** UV**).**rgb**;** // Aplicar textura a la luz especular

vec3 baseColor **=** texture**(**textures**[**TexIndex**],** UV**).**rgb**;** // Color base de la textura

vec3 result **=** **(**0.1 **\*** baseColor**)** **+** diffuse **+** specular**;** // Combinar color base, difuso y especular

// Convertir a escala de grises

float luminance **=** dot**(**result**,** vec3**(**0.299**,** 0.587**,** 0.114**));**

// Detección de bordes usando el filtro Sobel

vec2 texelSize **=** vec2**(**1.0**)** **/** textureSize**(**textures**[**TexIndex**],** 0**);**

float edgeDetection **=** 0.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** **-**1**)).**r **\*** **-**1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**0**,** **-**1**)).**r **\*** **-**2.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** **-**1**)).**r **\*** **-**1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** 1**)).**r **\*** 1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**0**,** 1**)).**r **\*** 2.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** 1**)).**r **\*** 1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** **-**1**)).**b **\*** **-**1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**0**,** **-**1**)).**b **\*** **-**2.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** **-**1**)).**b **\*** **-**1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** 1**)).**b **\*** 1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**0**,** 1**)).**b **\*** 2.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** 1**)).**b **\*** 1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** **-**1**)).**g **\*** **-**1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**0**,** **-**1**)).**g **\*** **-**2.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** **-**1**)).**g **\*** **-**1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** 1**)).**g **\*** 1.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**0**,** 1**)).**g **\*** 2.0**;**

edgeDetection **+=** texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** 1**)).**g **\*** 1.0**;**

edgeDetection **=** 1.0 **-** edgeDetection **/** 4**;**

// Calcular la variación local de la textura

float variation **=** 0.0**;**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** **-**1**)).**r **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** **-**1**)).**r **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** 1**)).**r **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** 1**)).**r **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** **-**1**)).**g **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** **-**1**)).**g **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** 1**)).**g **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** 1**)).**g **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** **-**1**)).**b **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** **-**1**)).**b **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(-**1**,** 1**)).**b **-** luminance**);**

variation **+=** abs**(**texture**(**textures**[**TexIndex**],** UV **+** texelSize **\*** vec2**(**1**,** 1**)).**b **-** luminance**);**

variation **/=** **(**4.0 **\*** 3**);** // Promedio de la variación

// Ajustar el umbral del patrón de punteado basado en la variación

float adjustedThreshold **=** mix**(**umbral**,** 0.5**,** variation**);**

// Aplicar el patrón de punteado basado en la detección de bordes y la variación local

int x **=** int**(**mod**(**gl\_FragCoord**.**x**,** 4.0**));**

int y **=** int**(**mod**(**gl\_FragCoord**.**y**,** 4.0**));**

int index **=** x **+** y **\*** 4**;**

float threshold **=** stipplePattern**[**index**];**

**if** **(**edgeDetection **<** threshold **\*** adjustedThreshold**)** **{**

col **=** vec4**(**0.0**,** 0.0**,** 0.0**,** 1.0**);** // Negro

**}**

**else** **{**

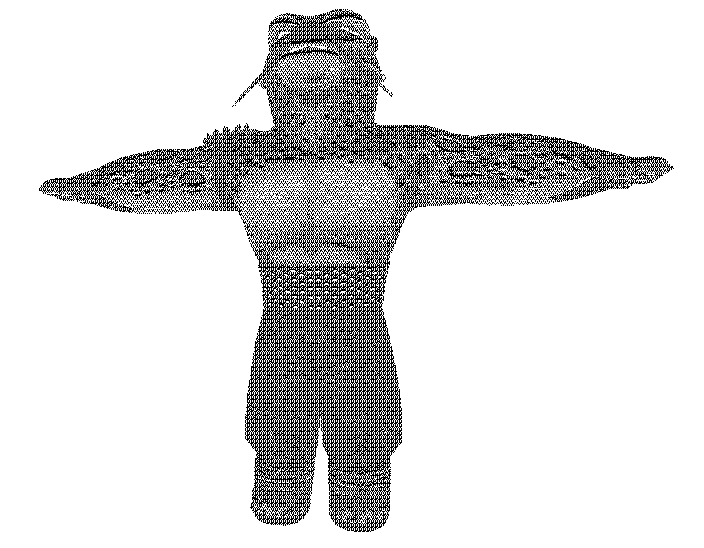
col **=** vec4**(**1.0**,** 1.0**,** 1.0**,** 1.0**);** // Blanco

**}**

**}**

**);**

Y nos daría resultados como estos:



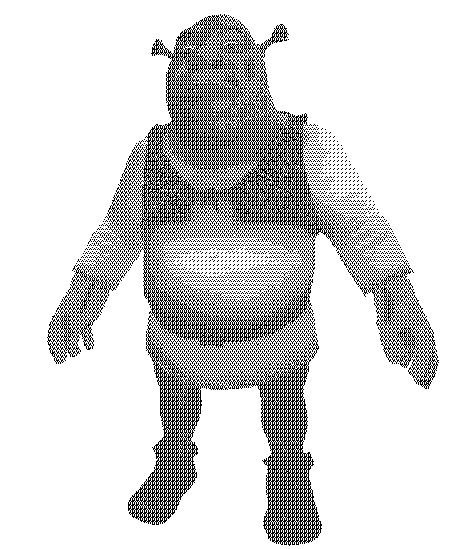
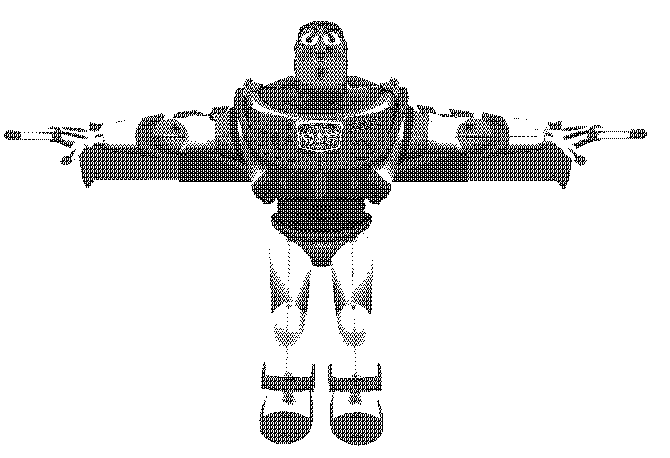


Imagen en blanco y negro

Descripción generada automáticamente con confianza media



## Painterly Rendering

### Procesado en CPU y GPU

Para este programa, en la GPU, el Vertex Shader transforma las posiciones de los vértices y transmite las coordenadas de textura y el índice de textura al Fragment Shader. En este último shader, se promedian los colores de la textura dentro del área del pincel y se calcula la variación de color en cada cuadrante. El cuadrante con la menor variación se selecciona y el color promedio de ese cuadrante se establece como el color del fragmento, logrando así un efecto artístico.

### Especificación del modelo

El Painterly Rendering es una técnica de renderizado no fotorrealista que busca imitar el estilo de las pinturas hechas a mano. Este enfoque se basa en la idea de que una pintura se construye a partir de una serie de pinceladas. En lugar de tratar cada píxel de una textura de forma individual, Painterly Rendering trata de representar la imagen como una serie de pinceladas. Estas pinceladas se consiguen a través de las siguientes operaciones realizadas en el fragmentador:

* Sobre el píxel actual que procesa el fragmentador se toman 4 matrices de pixeles del tamaño de “brocha” escogido más 1, es decir, (brocha+1)X(brocha+1). Las matrices son (respecto al píxel actual): superior izquierda y derecha e inferior izquierda y derecha.
* Para cada matriz, se suma el valor de todos sus pixeles, además, se suma también en otra variable el cuadrado de dichos valores de cada matriz.
* Por último, se evalúa el cuadrante cuya varianza es menor para dar una apariencia de suavizado (como cuando se pinta). Para lograrlo, se calcula el promedio de los valores de los pixeles de cada matriz y con el promedio se obtiene la varianza, en caso de ser menor que el valor mínimo de varianza encontrado actualmente se establece como color definitivo para el píxel actual el promedio de los pixeles de la matriz con menor varianza.

La implementación del Painterly Rendering en código quedaría de la siguiente manera:

/\* Vertex Shader de Painterly y Pixel Rendering \*/

const char**\*** vertex\_prog6 **=** GLSL**(**

// Entrada del vértice: posición

layout**(**location **=** 0**)** in vec3 pos**;**

// Entrada del vértice: normal

layout**(**location **=** 1**)** in vec3 normal**;**

// Entrada del vértice: coordenadas de textura

layout**(**location **=** 2**)** in vec2 uv**;**

// Entrada del vértice: índice de la textura

layout**(**location **=** 3**)** in float texIndex**;**

// Salida plana del índice de textura hacia el fragment shader

flat out int TexIndex**;**

// Salida de las coordenadas de textura hacia el fragment shader

out vec2 UV**;**

// Uniformes: posición de la cámara y matrices de transformación

uniform vec3 campos**;** // Posición de la cámara en coordenadas del mundo

uniform mat4 M**;** // Matriz de modelo

uniform mat4 PV**;** // Matriz de proyección-vista

void main**()**

**{**

// Convertir el índice de textura a entero y pasarlo al fragment shader

TexIndex **=** int**(**texIndex**);**

// Pasar las coordenadas de textura al fragment shader

UV **=** uv**;**

// Calcular la posición del vértice en el espacio del mundo

vec4 worldPosition **=** M **\*** vec4**(**pos**,** 1.0**);**

// Calcular la posición final del vértice en el espacio de pantalla

gl\_Position **=** PV **\*** worldPosition**;**

**}**

**);**

// Painterly Rendering - Fragment Shader

const char**\*** fragment\_prog6 **=** GLSL**(**

// Entrada plana del índice de textura desde el vertex shader

flat in int TexIndex**;** // Índice de la textura actual.

// Entrada de las coordenadas de textura desde el vertex shader

in vec2 UV**;** // Coordenadas de textura.

// Uniforme: array de texturas

uniform sampler2D textures**[**16**];**

// Uniforme: tamaño del pincel

uniform int brushSize**;** // Tamaño del pincel.

// Arrays para acumular los colores y los cuadrados de los colores

vec3 coloresCuadrante**[**4**];** // Array para acumular los colores promediados.

vec3 cuadradoCuadrantes**[**4**];** // Array para acumular los cuadrados de los colores.

// Función para acumular colores en los cuadrantes

void cuadrante\_pintar**(**in vec2 UV**,** in vec2 texSize**,** int brushSize**)** **{**

// Iterar sobre un área cuadrada alrededor del pixel actual

**for** **(**int i **=** **-**brushSize**;** i **<=** brushSize**;** i**++)** **{**

**for** **(**int j **=** **-**brushSize**;** j **<=** brushSize**;** j**++)** **{**

// Obtener el color de la textura en la posición desplazada

vec3 texColor **=** texture2D**(**textures**[**TexIndex**],** UV **+** vec2**(**i**,** j**)** **/** texSize**).**rgb**;**

// Determinar el cuadrante correspondiente

int quadrant **=** **(**i **<=** 0 **?** 0 **:** 2**)** **+** **(**j **<=** 0 **?** 0 **:** 1**);**

// Acumular el color y el cuadrado del color en el cuadrante correspondiente

coloresCuadrante**[**quadrant**]** **+=** texColor**;**

cuadradoCuadrantes**[**quadrant**]** **+=** texColor **\*** texColor**;**

**}**

**}**

**}**

void main**()**

**{**

// Obtener el tamaño de la textura en píxeles

vec2 texSize **=** textureSize**(**textures**[**TexIndex**],** 0**);**

// Calcular el área del pincel al cuadrado

float brushSizeSquare **=** float**((**brushSize **+** 1**)** **\*** **(**brushSize **+** 1**));**

// Inicializar los arrays a cero

**for** **(**int k **=** 0**;** k **<** 4**;** k**++)** **{**

coloresCuadrante**[**k**]** **=** vec3**(**0.0**);**

cuadradoCuadrantes**[**k**]** **=** vec3**(**0.0**);**

**}**

// Acumular colores y cuadrados de colores en los cuadrantes

cuadrante\_pintar**(**UV**,** texSize**,** brushSize**);**

// Variable para almacenar el valor mínimo de la suma de los cuadrados de los colores

float minColor **=** 1.0**;**

// Calcular el color promedio y la variación en cada cuadrante

**for** **(**int i **=** 0**;** i **<** 4**;** i**++)** **{**

// Promediar los colores acumulados en el cuadrante

coloresCuadrante**[**i**]** **/=** brushSizeSquare**;**

// Calcular la variación de color en el cuadrante

cuadradoCuadrantes**[**i**]** **=** abs**(**cuadradoCuadrantes**[**i**]** **/** brushSizeSquare **-** coloresCuadrante**[**i**]** **\*** coloresCuadrante**[**i**]);**

// Sumar las variaciones de color en los tres canales (r, g, b)

float sumColorSquare **=** cuadradoCuadrantes**[**i**].**r **+** cuadradoCuadrantes**[**i**].**g **+** cuadradoCuadrantes**[**i**].**b**;**

// Determinar el cuadrante con la menor variación de color

**if** **(**sumColorSquare **<** minColor**)** **{**

minColor **=** sumColorSquare**;**

// Asignar el color del cuadrante con la menor variación al color del fragmento

gl\_FragColor **=** vec4**(**coloresCuadrante**[**i**],** 1.0**);**

**}**

**}**

**}**

**);**

Y nos daría resultados como estos:

Imagen que contiene parado

Descripción generada automáticamente

Imagen que contiene sostener, cara, tablero, hombre

Descripción generada automáticamente

Imagen que contiene verde, vistiendo, viendo, cara

Descripción generada automáticamente

Imagen que contiene tabla, juguete, lego, colorido

Descripción generada automáticamente

## Pixel rendering

### Procesado en CPU y GPU

En el Pixel Rendering, el Vertex Shader transforma las posiciones de los vértices y transmite las coordenadas de textura y el índice de textura al Fragment Shader. En este shader, se ajustan las coordenadas de textura para crear un efecto de pixelación al agrupar los fragmentos en bloques y asignarles el mismo color de textura. Esto resulta en un efecto visual de pixelación, donde cada bloque de píxeles tiene un color uniforme basado en la textura original.

### Especificación del modelo

La técnica de pixel rendering implica renderizar la textura en base a un numero de pixeles horizontales y verticales especificados los cuales tienen que ser menores que el numero de pixeles que se tenia originalmente en dicha textura. De esta forma se consigue “pixelizar” la textura haciéndola menos detallada y más “pixel art”. Para lograrlo, establecemos 2 variables que nos permitirán indicar a partir de que “pixel” miramos el siguiente pixel de la textura original, es decir, a partir del número de pixeles empezamos a colorear el pixel actual con el siguiente pixel de la textura original. Esto hará que hagamos X grupos de pixeles colindantes con el mismo valor que en la textura original tenían distintos valores.

La implementación del Pixel Rendering en código quedaría de la siguiente manera:

/\* Vertex Shader de Painterly y Pixel Rendering \*/

const char**\*** vertex\_prog6 **=** GLSL**(**

// Entrada del vértice: posición

layout**(**location **=** 0**)** in vec3 pos**;**

// Entrada del vértice: normal

layout**(**location **=** 1**)** in vec3 normal**;**

// Entrada del vértice: coordenadas de textura

layout**(**location **=** 2**)** in vec2 uv**;**

// Entrada del vértice: índice de la textura

layout**(**location **=** 3**)** in float texIndex**;**

// Salida plana del índice de textura hacia el fragment shader

flat out int TexIndex**;**

// Salida de las coordenadas de textura hacia el fragment shader

out vec2 UV**;**

// Uniformes: posición de la cámara y matrices de transformación

uniform vec3 campos**;** // Posición de la cámara en coordenadas del mundo

uniform mat4 M**;** // Matriz de modelo

uniform mat4 PV**;** // Matriz de proyección-vista

void main**()**

**{**

// Convertir el índice de textura a entero y pasarlo al fragment shader

TexIndex **=** int**(**texIndex**);**

// Pasar las coordenadas de textura al fragment shader

UV **=** uv**;**

// Calcular la posición del vértice en el espacio del mundo

vec4 worldPosition **=** M **\*** vec4**(**pos**,** 1.0**);**

// Calcular la posición final del vértice en el espacio de pantalla

gl\_Position **=** PV **\*** worldPosition**;**

**}**

**);**

// Pixel Rendering - Fragment Shader

const char**\*** fragment\_prog7 **=** GLSL**(**

// Entrada plana del índice de textura desde el vertex shader

flat in int TexIndex**;**

// Entrada de las coordenadas de textura desde el vertex shader

in vec2 UV**;**

// Uniforme: array de texturas (hasta 16 texturas)

uniform sampler2D textures**[**16**];**

// Uniforme: multiplicador para ajustar el número de píxeles

uniform float multiplier **=** 0.2f**;**

// Desplazamiento en x e y para el efecto pixelado

float Despx **=** 1 **/** 100.f**;**

float Despy **=** 1 **/** 100.f**;**

// Nuevas coordenadas de textura después del efecto pixelado

vec2 newUV**;**

void main**()**

**{**

// Obtener el tamaño de la textura actual en píxeles

vec2 texSize **=** textureSize**(**textures**[**TexIndex**],** 0**);**

// Calcular el desplazamiento en x e y basado en el tamaño de la textura y el multiplicador

Despx **=** 1.0 **/** **(**texSize**[**0**]** **\*** multiplier**);**

Despy **=** 1.0 **/** **(**texSize**[**1**]** **\*** multiplier**);**

// Ajustar las coordenadas de textura para crear el efecto pixelado

newUV**.**x **=** floor**(**UV**.**x **/** Despx**)** **\*** Despx**;**

newUV**.**y **=** floor**(**UV**.**y **/** Despy**)** **\*** Despy**;**

// Asignar el color del fragmento basado en las nuevas coordenadas de textura pixeladas

gl\_FragColor **=** texture2D**(**textures**[**TexIndex**],** newUV**);**

**}**

**);**

Y nos daría resultados como estos:

Imagen que contiene hombre, vistiendo, parado, sostener

Descripción generada automáticamente

Imagen que contiene tablero, café, llevar, verde

Descripción generada automáticamente

Imagen que contiene viendo, cara, hombre, café

Descripción generada automáticamente

Una caricatura de una persona

Descripción generada automáticamente con confianza media

## Ejecución del código

El código está desarrollado en el entorno de la asignatura por lo que utilizando el CMakeList para compilar y ejecutando como el resto de las entregas funcionara.

Una vez se ejecute el programa aparecerá la ventana con el modelo de iluminación de Phong. Pulsando las teclas numéricas se cambiará de modelo de la siguiente forma:

* 1->Phong
* 2->Blinn-Phong
* 3->Resaltado de siluetas
* 4->Toon Shading
* 5->Stippling
* 6->Painterly Rendering
* 7->Píxel Rendering

Además, también se ha implementado una forma para mover la cámara y la iluminación del modelo. Para mover la cámara se utilizará las teclas WASD como se utiliza típicamente en videojuegos, es decir, W arriba, A izquierda, S abajo y D derecha. Este movimiento se hará alrededor de la figura en el eje horizontal, pero de arriba abajo sin tener en cuenta la posición de la figura para el eje vertical. Para la iluminación se utilizarán las teclas de las “flechas” del teclado. Además, con la tecla R se reiniciará la posición de la cámara a la original.

Por último, para los modelos de Stippling, Painterly Rendering y Pixel Rendering se ha implementado que pulsando determinadas teclas se pueda cambiar el umbral de puntos, cambiar tamaño del pincel y cambiar el tamaño de los pixeles respectivamente. Para este cambio en Stippling se usarán las teclas O para aumentar y L para disminuir, en Painterly Rendering I para aumentar y K para disminuir y en Pixel Rendering U para aumentar y J para disminuir.

## Notas y Aclaraciones

### Carga de modelos .obj

Debido a la escasa oferta de archivos .bix en internet, ha sido necesario implementar una función que haga posible la carga de objetos .obj junto a su textura a nuestro programa. Estos objetos están compuestos por el archivo que da “volumen” a la figura que sería el .obj, las texturas en formato .png y un archivo adicional que es el encargado de asociar los vértices del objeto a su coordenada dentro de las texturas, típicamente un .mtl.

### Requisitos técnicos

Para poder compilar y utilizar el código de la entrega, si se está utilizando un sistema operativo Linux (o similar), será preciso asegurarse de que se tiene la dependencia ‘libassimp-dev’, la cual se puede instalar con el siguiente comando:

sudo apt install libassimp-dev

### Colaboración

Cabe destacar que esta implementación ha sido llevada a cabo con la ayuda del grupo compuesto por Aníbal Antonio Rivero Ríos, Andrés García Solórzano y Luis Escaño Márquez basándonos en el siguiente repositorio de Github:

https://github.com/JoeyDeVries/LearnOpenGL