Universidade do Minho

Escola de Engenharia

Visualização em Tempo Real

Mestrado em Engenharia Informática

Non-Photorealistic Rendering

Diogo Vieira(PG50518)

Laura Rodrigues(PG50542)

Mariana Marques(PG50633)

Junho, 2023

Conteúdo

1. Introdução

A renderização **Não Foto-Realista** é uma abordagem em computação gráfica que permite criar imagens com estilos artísticos distintos, em vez de se concentrar na reprodução precisa da realidade, como na renderização Foto-Realista.

Esta abordagem envolve o uso de diversos algoritmos responsáveis por simular técnicas artísticas de desenho e pintura, tais como:

- **Mapeamento de Texturas**
- **Deteção de Bordas**
- **Filtragem de Cores**
- **Estilização de Linhas**

Para o desenvolvimento do presente projeto, opta-se pelo desenvolvimento de diversos filtros, tais como:

1. Outlines
2. Pixelation
3. Toon Shading
4. Rim Lighting
5. Gooch Shading
6. Hatching
7. Sobel Edges

2. Outlines

Na tentativa de salientar os contornos dos objetos, é implementada a seguinte estratégia:

Utilização do **Geometry Shader**, que recebe como inputs a adjacência de triângulos, e vai devolver como output uma line strip.

Como input teremos 4 triângulos, 3 de adjacência e 1 original.

De seguida, para os 4 triângulos:

- Cálculo do vetor de direção da camara para o centro
- Cálculo da norma de cada triâgulo
- Verificar se a normal do triângulo original é negativa
- Em caso verdadeiro, verificar a normal de cada triângulo, se for positiva, ou seja, contrária da original, emitir essa aresta

Deste modo, estamos a verificar, se em relação à posição da câmara, os triângulos adjacentes têm normais contrárias do triângulo original, deste modo, identificamos o outline.

Listing 2.1: outlines

```
    vec3 c = normalize(camPos - (ps[0]+ps[2]+ps[4])/3) ;
    ...
    vec3 N042 = cross(ps[4] - ps[0], ps[2] - ps[0]);
    ...
    float dotView = dot(N042, c);
    if (dotView < 0.0){
        dotView = dot(N021, c2);
        if (dotView >= 0)
            EmitLine(0,2);
        dotView = dot(N243,c4);
        if (dotView >= 0)
            EmitLine(2,4);
        dotView = dot(N405,c6);
        if (dotView >= 0)
            EmitLine(4,0);
}
```

3. Pixelation

A pixelização é um processo de aplicação de um filtro numa textura, dividindo-a em pequenos blocos de *pixeis*. Cada bloco tem uma cor homogénea calculada a partir da média da cor do pixel, desenvolvendo uma aparência granulada e irregular na imagem, tornando os detalhes menos nítidos e reconhecíveis.

3.1 Fragment Shader

Para a implementação do filtro responsável pela pixelização, foi necessário realizar um conjunto de etapas, destacando-se:

- **Coordenadas de Textura**

  É necessário ter acesso ás informações da textura que é recebida pelo pipeline.

```
vec2 texSize = textureSize(tex, 0).xy;
```

- **Número de Pixeis**

  Como já foi referido anteriormente, o número de pixeis tem de ser definido para especificar as dimensões da matriz que será aplicada à textura.

```
int pixelSize = 9;
```

- **Coordenadas do Ecrã**

  Opta-se por utilizar as coordenadas do ecrã para a implementação deste filtro, dadas por gl_FragCoord.

  Os cálculos seguintes são feitos para determinar as novas coordenadas dentro do bloco de pixel tendo em conta as coordenadas do ecrã e o deslocamento necessário para centralizar o fragmento nesse bloco.

```
float x = int(gl_FragCoord.x) % pixelSize;
float y = int(gl_FragCoord.y) % pixelSize;

x = floor(pixelSize / 2.0) - x;
y = floor(pixelSize / 2.0) - y;

x = gl_FragCoord.x + x;
y = gl_FragCoord.y + y;
```

- **Cálculo da Cor**

  Para calcular a cor é definida uma nova variável de textura normalizada, dada por uv, que é posteriormente aplicada à textura inicial, dada por tex.

```
vec2 uv = vec2(x, y) / texSize;
color = texture(tex, texCoord + uv);
colorOut = color;
```

3.2 Resultado

Figura 3.1: Representação do Filtro de Pixelização

Apesar de termos conseguido o efeito pretendido, ao mover a câmara, percebe-se que a textura não está fixa mexendo-se conforme este movimento.

4. Toon Shading

Toon Shading é uma técnica de renderização Não Foto-Realista utilizada na computação gráfica para criar um visual semelhante ao de desenhos animados. Ao contrário da renderização realista, esta técnica aplica sombras planas e contornos bem definidos, criando uma estética mais estilizada e marcante.

4.1 *Vertex Shader*

4.1.1 Variáveis Uniformes

Em relação às variáveis uniformes, são definidas as seguintes matrizes **PVM**, **VM** e **m_normal** responsáveis por realizar as transformações e calcular as normais corretas nos vértices durante o processo de renderização.

- **PVM** - Combina as transformações de projeção e visualização responsáveis por projetar os objetos da cena em um espaço 2D e posicionar a câmara no espaço 3D.
- **VM** - Posiciona e orienta a câmara no espaço 3D, determinando o ponto de vista a partir do qual a cena será observada.
- **m_normal** - Ajusta corretamente as normais dos objetos face às transformações de escala, rotação e translação aplicadas aos mesmos, preservando sua direção relativa.

Listing 4.1: Matrizes Uniformes

```
uniform mat4 PVM;
uniform mat4 VM;
uniform mat3 m_normal;
```

4.1.2 Variáveis de Entrada

Em relação às variáveis de entrada, são definidas as seguintes variáveis **position** e **normal** responsáveis pela posição e normal do vértice em questão.

Listing 4.2: Variáveis de Entrada

```
in vec4 position;
in vec3 normal;
```

4.1.3 Variáveis de Saída

Em relação às variáveis de saída, são definidas as seguintes variáveis **e** e **n**.

- **e** - Transforma a posição do vértice do sistema de coordenadas do mundo para o sistema de coordenadas da câmara.
- **n** - Ajusta a normal do vértice conforme as transformações aplicadas aos objetos.

Listing 4.3: Variáveis de Saída

```
out vec3 e;
out vec3 n;
```

Cálculo das Variáveis de Saída

Em relação ao cálculo das variáveis de saída, é utilizada a matriz de visualização para colocar o vértice no espaço da câmara, *e*, e a matriz *m_normal* para ajudar a normal do vértice ás transformações geométricas, *n*.

Listing 4.4: Variáveis de Saída

```
e = -vec3(VM * position);
n = normalize(m_normal * normal);
```

Por fim, é calculada a posição final do vértice após todas as transformações necessárias, e esta é atribuída à variável **gl_Position**.

Listing 4.5: Posição Final do Vértice

```
gl_Position = PVM * position;
```

4.2 *Fragment Shader*

4.2.1 Variáveis Uniformes

Em relação ás variáveis uniformes, são definidas as seguintes variáveis **m_view**, **l_dir**, **diffuse**, **specular** e **shininess** responsáveis pela luz, visualização dos objetos e material.

- **m_view** - Representa a matriz de visualização, responsável por transformar os vértices do espaço local para o espaço da câmara.
- **l_dir** - Representa a direção da luz.

- **diffuse**, **specular** e **shininess** - Representam a cor e o brilho do material.
- **num_division** - Representa o número de divisões que a intensidade da luz será dividida.
- **rimLight** - Representa a intensidade da Luz de Aro.

Listing 4.6: Variáveis Uniformes

```
uniform mat4 m_view;
uniform vec4 l_dir;
uniform vec4 diffuse;
uniform vec4 specular;
uniform float shininess;
uniform int num_divisions;
uniform float rimLight;
```

4.2.2 Variáveis de Entrada

Em relação às variáveis de entrada, são definidas as seguintes variáveis n e e, que são recebidas no *pipeline* de renderização pelo *Vertex Shader*.

Listing 4.7: Variáveis de Entrada

```
in vec3 n;
in vec3 e;
```

4.2.3 Variáveis de Saída

Em relação às variáveis de saída, é definida apenas a variável *color* responsável pela cor final no vértice.

Listing 4.8: Variáveis de Saída

```
out vec4 color;
```

Cálculo da Cor

1. **Normalizar as Variáveis de Entrada**

   ```
   vec3 nn = normalize(n);
   vec3 ee = normalize(e);
   ```

2. **Calcular o Vetor da Direção da Luz**

   Reajustar as coordenadas do vetor l do sistema de coordenadas do espaço do mundo

para o sistema de coordenadas da câmara, através da matriz *m_view*.

```
vec3 l = normalize(vec3(m_view * -l_dir));
```

3. **Calcular a Intensidade da Luz Difusa**

   A intensidade da luz difusa é calculada através do produto escalar entre o vetor da direção da luz, *l*, e o vetor da normal, *nn*.

```
float i = max(0.0, dot(l,nn));
```

4. **Calcular a Intensidade do Aro de Luz**

   A intensidade do aro de luz é dada pela variável *rimLightIntensity* e é calculada pelo produto escalar entre *ee* e *n*.

```
float rimLightIntensity = dot(ee, n);
```

   O resultado é invertido para aumentar a intensidade nas áreas onde o vetor de visão, *ee*, está alinhado com a normal, *n*.

```
rimLightIntensity = 1.0 - rimLightIntensity;
```

   Os resultados negativos, adquirem o valor nulo.

```
rimLightIntensity = max(0.0, rimLightIntensity);
```

   Os restantes valores sofrem uma interpolação que reproduz um efeito de contorno ao redor das áreas iluminadas, onde os valores acima de abaixo de rimLight ficam nulos e os valores acima de rimLight ficam a 1. A variável rimLight é escolhida pelo utilizador permitindo aumentar ou diminuir o efeito da luz.

```
rimLightIntensity = smoothstep(rimLight, rimLight + 0.1, rimLightIntensity);
```

5. **Calcular a Especular**

   A componente especular é dada pela variável *spec* e apenas é calculada quando incide luz no vértice em questão.

   Em relação ao cálculo da intensidade, é utilizado o produto escalar entre o vetor intermédio, *h*, e o vetor da normal *n*.

   Em relação ao cálculo do brilho, a intensidade é elevada à variável *shininess*.

Figura 4.1: Rim Lighting a vermelho no teapot

```
if (i > 0.0) {
      vec3 h = normalize(l + ee);
      float intSpec = max(dot(h,n), 0.0);
      spec = specular * pow(intSpec,shininess);
}
```

6. **Calcular a Intensidade da Luz**

   A Intensidade da Luz é calculada com base no número de divisões, quanto menor o número de divisões, maior é diferença da intensidade da luz entre divisões, e vice-versa.

```
float division = 1.0 / float(num_divisions);
i = ceil((i +0.00001) * num_divisions) * division;
```

7. **Cálculo da Cor**

   Por fim, a variável *color* é calculada tendo em conta a intesidade da componente difusa, especular e do Aro de Luz.

```
color = i * diffuse + rimLight + spec;
```

4.3 Resultado

Figura 4.2: Filtro Toon no teapot

5. Gooch Shading

5.1 Vertex Shader

O ficheiro *.vert* da técnica **Gooch** iguala-se ao ficheiro *.vert* da técnica de **Toon**.

5.2 Fragment Shader

5.2.1 Variáveis Uniformes

Em relação às variáveis uniformes, são definidas as seguintes variáveis **m_view**, **l_dir**, **coolColor**, **warmColor**, **specular**, **shininess** e **alpha** e **beta** responsáveis pela visualização dos objetos, luz e cor do material.

- **l_dir** - Representa a direção da luz.
- **m_view** - Representa a matriz de visualização, responsável por transformar os vértices do espaço local para o espaço da câmara.
- **coolColor**, **warmColor**, **alpha** e **beta** - Responsável pela cor fria e quente do material, obtida através do peso atribuído a cada uma, respetivamente, que pode ser alterado pelo utilizador.
- **specular** e **shininess** - Representam a cor e o brilho do material.

Listing 5.1: Variáveis Uniformes

```
uniform mat4 m_view;
uniform vec4 l_dir;
uniform vec3 coolColor = vec3(0.0f, 0.0f, 0.8f);
uniform vec3 warmColor = vec3(0.4f, 0.4f, 0.0f);
uniform float alpha = 0.25;
uniform float beta = 0.5;
uniform vec4 specular;
uniform float shininess;
```

5.2.2 Variáveis de Entrada

Em relação ás variáveis de entrada, são definidas as seguintes variáveis e e n, que são recebidas no *pipeline* de renderização pelo *Vertex Shader*.

```
in vec3 n;
in vec3 e;
```

5.2.3 Variáveis de Saída

Em relação ás variáveis de saída, é definida apenas a variável *color* responsável pela cor final no vértice.

```
out color;
```

Cálculo da Cor

- **Normalizar as Variáveis de Entrada**

```
vec3 nn = normalize(n);
vec3 ee = normalize(e);
```

- **Calcular o Vetor da Direção da Luz**

Reajustar as coordenadas do vetor l do sistema de coordenadas do espaço do mundo para o sistema de coordenadas da câmara, através da matriz *m_view*.

```
vec3 l = normalize(vec3(m_view * -l_dir));
```

- **Calcular a Intensidade da Luz**

A intensidade da luz difusa é calculada através do produto escalar entre a o vetor da direção da luz e a normal.

```
float i = max(dot(l,norm), 0.0);
```

- **Cálculo das Cor Fria e Quente**

Cálculo da Cor Fria, dada por *coolColor*, e Quente, dada por *warmColor*, através de um pequeno ajuste dado por *alpha* e *beta*.

```
vec3 finalCool = coolColor + alpha * * vec3(1);
vec3 finalWarm = warmColor + beta * * vec3(1);
```

- **Interpolação**

Cálculo do valor de interpolação, dada por *lerp*, através do intensidade da luz difusa. Posteriormente, esse valor é aplicado ás cores finais de forma oposta, isto é, $1 - lerp$ para a cor fria e $lerp$ para a cor quente.

```
float lerp = (1.0 + diff) / 2.0;

finalCool = (1 - lerp) * finalCool;
finalWarm = lerp * finalWarm;
```

- **Calcular a Especular**

  A componente especular é dada pela variável *spec* e apenas é calculada quando incide luz no vértice em questão.

  Em relação ao cálculo da intensidade, é utilizado o produto escalar entre o vetor intermédio, h, e o vetor da normal n.

  Em relação ao cálculo do brilho, a intensidade é elevada à variável *shininess*.

```
if (diff > 0.0) {
      vec3 h = normalize(l + ee);
      float intSpec = max(dot(h,n), 0.0);
      spec = specular * pow(intSpec,shininess);
}
```

- **Cálculo da Cor Final**

  Por fim, é calculada a cor final onde se soma as componentes da cor fria, quente e especular.

```
outColor = vec4(finalCool + finalWarm, 1.0) + spec;
```

5.3 Resultado

Figura 5.1: Filtro Gooch no teapot

6. Cross Hatching

O nosso filtro funciona, à semelhança do Toon Shading, pela divisão do objeto em secções de acordo com a intensidade da luz i. Assim, foram usadas 6 texturas com o objetivo de imitar este estilo de desenho. Da textura 0 à 5 estas vão ficando cada vez mais preenchidas.

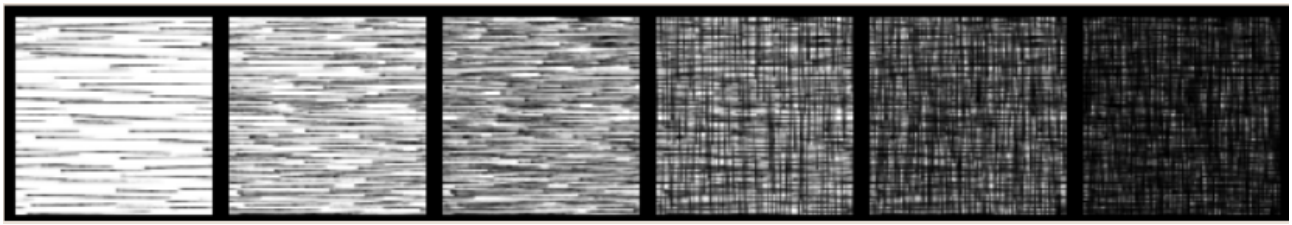

Figura 6.1: Texturas

Posteriormente é usada a função pré-definida mix para suavizar a transição entre estas texturas.

```
float step = 1. / 6.;
if( i <= step )
    c = mix( tex5, tex4, 6. * i );
if( i > step && i <= 2. * step )
    c = mix( tex4, tex3, 6. * ( i - step ) );
if( i > 2. * step && i <= 3. * step )
    c = mix( tex3, tex2, 6. * ( i - 2. * step ) );
if( i > 3. * step && i <= 4. * step )
    c = mix( tex2, tex1, 6. * ( i - 3. * step ) );
if( i > 4. * step && i <= 5. * step )
    c = mix( tex1, tex0, 6. * ( i - 4. * step ) );
if( i > 5. * step )
    c = mix( tex0, vec4( 1. ), 6. * ( i - 5. * step ) );
```

6.1 Resultado

Figura 6.2: Filtro Hatch

7. Sobel Edges

O filtro Sobel é usado em algoritmos de detecção de contornos. Este calcula o gradiente da intensidade da imagem em cada ponto, dando a direcção da maior variação de claro para escuro e a quantidade de variação nessa direcção. Assim, obtém-se uma noção de como varia a luminosidade em cada ponto, de forma mais suave ou abrupta

X – Direction Kernel		
-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel		
-1	-2	-1
0	0	0
1	2	1

Figura 7.1: Matrizes de Sobel

Inicialmente cria-se assim uma matriz que permitirá saber se numa dada posição do vetor n é 0 ou um valor positivo ou negativo.

Posteriormente, recorrendo a *sobel_edge_h* e a *sobel_edge_v* obtém-se assim a matriz final que permite obter o efeito Sobel.

```
 vec4 n[9];
make_kernel( n, moments, texCoordV);
vec4 sobel_edge_h = n[2] + (2.0*n[5]) + n[8] - (n[0] + (2.0*n[3]) + n[6]);
vec4 sobel_edge_v = n[0] + (2.0*n[1]) + n[2] - (n[6] + (2.0*n[7]) + n[8]);
vec4 sobel = sqrt((sobel_edge_h * sobel_edge_h) + (sobel_edge_v * sobel_edge_v));
color = vec4(1 - sobel.rgb,1.0 );
```

Infelizmente a implementação deste filtro permaneceu com problemas, ficando assim para trabalho futuro.

8. Conclusão

Através da nossa pesquisa e de diversas tentativas de implementações, foi-nos possível testar o potencial destes filtros para transformar imagens digitais em representações artísticas visualmente cativantes.

Os filtros desenvolvidos neste artigo oferecem uma ampla gama de efeitos artísticos, cada um com o seu próprio apelo estético único.

Embora este artigo tenha introduzido com sucesso um conjunto diversificado de filtros, ainda há muito espaço para melhorias e desenvolvimentos futuros. Implementações futuras poderiam incluir: melhorar os filtros existentes de Pixelation e Sobel Edges e introduzir novos, como por exemplo, Pontilhismo ou outros estilos de pintura/desenho, olho de peixe, etc.