Universidade do Minho
Escola de Engenharia
Mestrado Integrado em Engenharia Informática

Unidade Curricular de Visualização e Iluminação

Ano Letivo de 2019/2020

Non-Photorealistic Rendering

A78824 Mariana Lino Costa
A76387 André Ramalho
A64282 Paulo Alves

Janeiro, 2020

VI1

Índice

Índice de figuras

Resumo

Este relatório contém toda a explicação do desenvolvimento e implementação de vários métodos para a transformação de modelos de computação gráfica através de *shaders*. Para além disso apresenta a análise dos seus resultados e desempenho, e a discussão de toda a dificuldade que se enfrentou no decorrer da elaboração do projeto.

Introdução

No âmbito da unidade curricular de Visualização de Iluminação do perfil Computação Gráfica do Mestrado de Engenharia Informática na Universidade do Minho, foi proposto a exploração e desenvolvimento de *shaders* relacionadas com um tema à escolha.

O tema escolhido foi *Non-Photorealistic Rendering.* A renderização não foto-realista (NPR) é uma área de computação gráfica que se concentra em permitir uma grande variedade de estilos expressivos para a arte digital. Ao contrário da computação gráfica tradicional, que está mais focada no fotorrealismo, a NPR é inspirada em estilos artísticos, como a pintura, o desenho, a ilustração técnica e os desenhos animados. É utilizada em filmes e videojogos na forma de "*toon shading*", bem como em visualização científica, ilustração arquitetónica e animação experimental. Um exemplo de uso moderno desse método é o da animação com sombra cel.

No desenvolvimento deste projeto houve uma exploração dos diferentes estilos expressivos desta arte, a implementação de *shaders* para cada método, de forma a se conseguir comparar o desempenho e distinguir os diversos efeitos no modelo.

Na primeira secção deste relatório houve uma pequena abordagem ao tema, e a menção de todos os estilos de NPR. Numa segunda fase passou-se à implementação de *shaders* para cada estilo e apresentação dos seus resultados.

Non-Photorealistic Rendering

Numa primeira fase foi importante estudar acerca do tema escolhido, entender quais os tipos e possíveis técnicas que se poderiam desenvolver para criar *shaders* que representassem o *Non-Photorealistic Rendering.*

Antes de se definir NPR é interessante entender o que realmente significa o termo *photorealistic rendering*. *Photo* vem do grego phos, que significa luz, ou aquilo que é produzido pela luz, e *Realistic* descreve ou enfatiza o que é real. *Rendering* na Computação Gráfica refere-se ao processo por qual a representação de uma cena virtual é convertida numa imagem para visualização. O *Non-Photorealistic*, contrariamente, representa algo que não é real, que é abstrato.

Na tabela a seguir mostra uma comparação entre os termos *Photorealism* e NPR, de forma a que haja uma melhor noção e compreensão do que é NPR [1].

	Photorealism	NPR
Abordagem	Simulação	Estilo
Característica	Objetivo	Subjetivo
Influência	Simulação de processos físicos	Processos artísticos
Precisão	Preciso	Aproximado
Veracidade	Pode ser enganoso ou considerado desonesto. Os espectadores podem ser "enganados" e acreditar que uma imagem é real	Honesto, o observador vê uma imagem como apenas uma representação de uma cena
Nível de Detalhe	Difícil de evitar detalhes estranhos, muita informação, nível constante de detalhe	Pode adaptar o nível de detalhe em uma imagem para focar a atenção do espectador
Completude	Completo	Incompleto
Representação	Superfícies rígidas	Fenômenos naturais e orgânicos

O *Non-Photorealistic Rendering* é, então, uma área específica da computação gráfica, um processo pelo qual os engenheiros nessa área tentam animar e representar itens inspirados em pinturas, desenhos técnicos, desenhos animados e outras fontes que não apresentam fotorrealismo.

É frequentemente usado nos projetos atuais de animação por computador, onde uma grande diversidade de métodos cria diferentes estilos de televisão e cinema.

Existem várias técnicas e categorias de NPR, como ilustrações de caneta e tinta, renderização de pinturas, de desenhos animados, ilustrações técnicas e visualização científica.

Procurou-se estudar cada uma destas categorias e as várias técnicas que criam estes efeitos.

Pen and ink illustrations

Esta categoria de *Non-Phororealisttic* usa a arte de linha, técnicas de contorno, ou até mesmo técnicas como *hatching, cross-hatching*, entre muitas outras para criar obras de arte. [2]

Figura 1 - Exemplo de uma renderização de NPR com *Pen and ink illustrations*.

- Técnica de Hatching

Hatching é uma técnica usada para valorizar um modelo de linhas. As linhas usadas vão principalmente na mesma direção para uma área definida. Normalmente não são cruzadas, são linhas paralelas e podem ser usadas posteriormente como linhas de cross-hatching para ajudar a definir a forma do objeto. Quanto mais próximas as linhas, mais sombreado se torna o objeto.

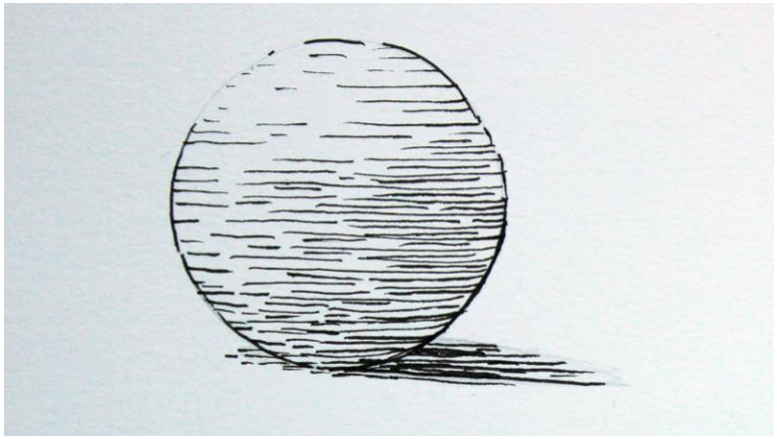


Figura 2 - Exemplo de um desenho feito com a técnica de Hatching.

- Técnica de Cross-Hatching

Cross-hatching é o desenho de duas camadas de linhas em ângulos retos para criar um padrão com aparência de malha. Várias camadas em diferentes direções podem ser usadas para criar texturas ou para criar contornos criativos. O cross-hatching é frequentemente usado para criar efeitos tonais, variando o espaçamento das linhas ou adicionando camadas adicionais de linhas. É uma forma de adicionar detalhes de desenho animado e sombreamento num objeto. Embora seja usado com mais frequência em desenhos ou pinturas à mão, pode ser usado como efeito de computação gráfica.

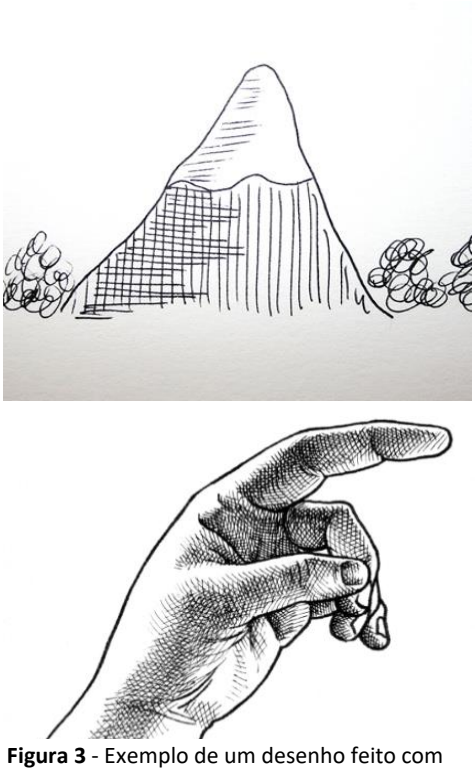


Figura 3 - Exemplo de um desenho feito com técnica de *cross-hatching*.

- Técnica de Random Lines

Linhas em várias direções também podem ser usadas para criar desenhos. Alterando a frequência do cruzamento das linhas, pode-se controlar o intervalo de valor produzido. O uso deste método pode criar uma variedade de texturas diferentes.

Figura 4 - Exemplo de um desenho com a técnica de *Random Lines*.

- Técnica de Stippling

Pontilhar é adicionar incontáveis pontos para criar do desenho. Quanto maior a concentração de pontos, mais escuro o desenho fica. Quanto mais espaço entre os pontos, mais claro.

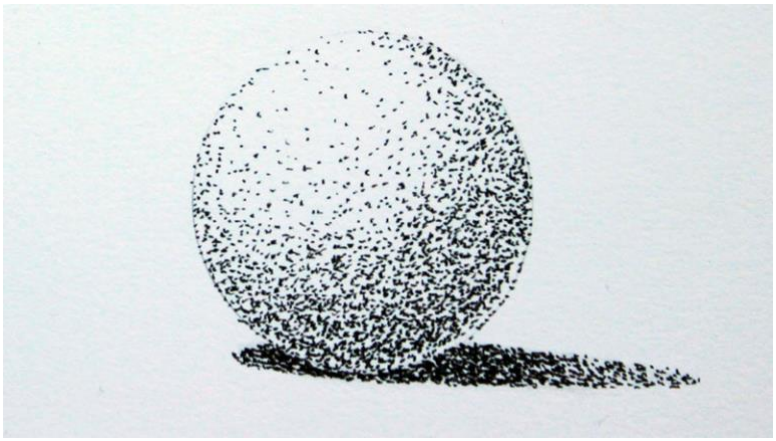


Figura 5 - Exemplo de um desenho com a técnica de Stippling.

Painterly rendering

A renderização de pinturas visa emular ou replicar estilos e aspeto da pintura e desenho. É comum o uso de texturas pintadas à mão e a utilização de imagens que funcionam como *brushes* para permitir a visualização das pinceladas.

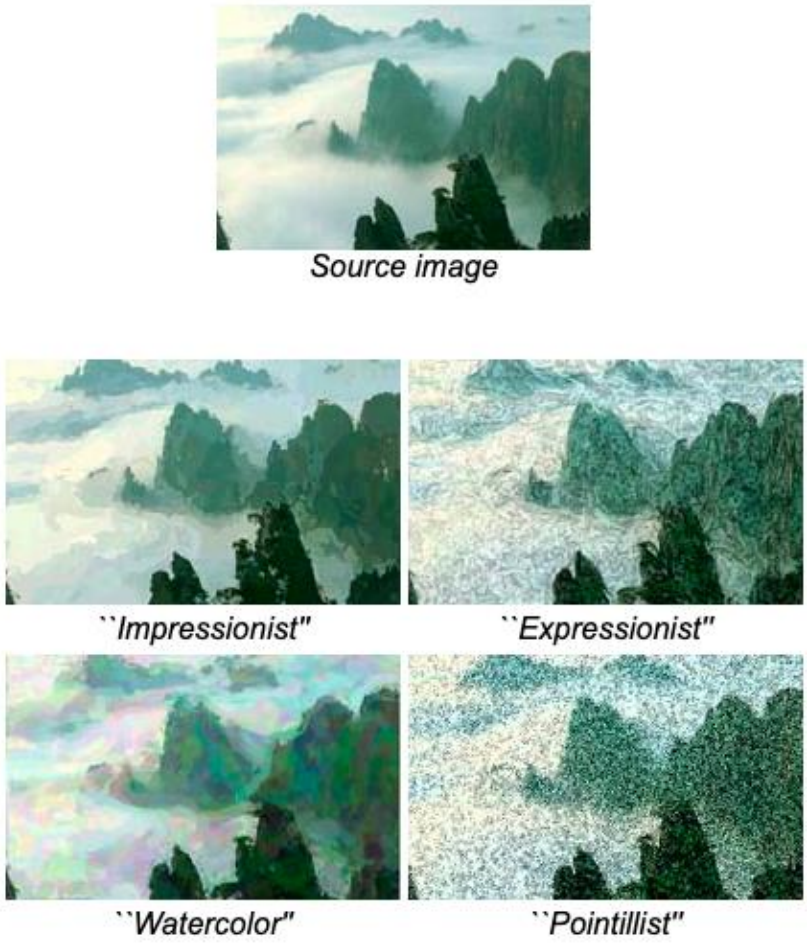


Figura 6 - Exemplos de NPR com vários estilos de painterly

A partir de um modelo de origem, cria-se uma imagem com uma aparência pintada à mão. Existem quatro estilos distintos, "impressionista", "expressionista", "Aguarela" e "Pontilhista".

O estilo "impressionista" gera uma pintura que tenta transmitir a cena dentro da foto de forma realista, não existem grandes restrições às pinceladas (*brushes*) usadas, o estilo "expressionista" usa mais restrições, pinceladas longas e as cores da pintura são

aprimoradas. O estilo “Pontilhista” é semelhante ao “Impressionista”, mas as pinceladas são limitadas, devem ser extremamente curtas quase como se fossem pontos na imagem.

Os estilos de pintura são determinados pelos parâmetros passados no código como:

- Limiar de aproximação *(threshold)*;
- Tamanho do pincel;
- Filtro de curvatura (limite ou curvatura exagerada);
- Fator de desfoque (mais desfoque nas imagens de estilo “impressionista”);
- Comprimentos mínimos e máximos do curso/*stroke*(cursos muito curtos são usados no estilo “pontilhista”);
- Opacidade (baixa opacidade para um efeito de *wash*/lavagem);
- Tamanho da grade;
- Tremulação da cor;

Cartoon rendering

A renderização em *cartoon* usa efeitos como sombreamento e distorção para obter efeitos semelhantes aos desenhos animados. Esta renderização é muito usada em filmes animados e desenvolvimento de técnicas de cronograma real para jogos. Existem várias técnicas para criar este efeito já conhecidas.

- Toon Shader

O *Toon Shader*, também conhecido por *Cel shading*, é um tipo de renderização não fotorrealista que usa um conjunto de técnicas empregadas na renderização de imagens 3D de modo que o resultado final se assemelhe aos desenhos em 2D.

O *toon Shader* começa então com um modelo 3D típico, e altera todo o seu modelo de iluminação. Os valores de iluminação convencionais (suaves) são calculados para cada pixel e, em seguida, quantizados em um pequeno número de tons discretos para criar a aparência plana característica, onde as sombras e os realces aparecem como blocos de cores em vez de serem misturados suavemente.

Basicamente, este *shader* substitui o sombreamento suave do seu modelo por faixas de cores sólidas, simulando o estilo de desenho usado em histórias de quadrinhos, desenhos animados.

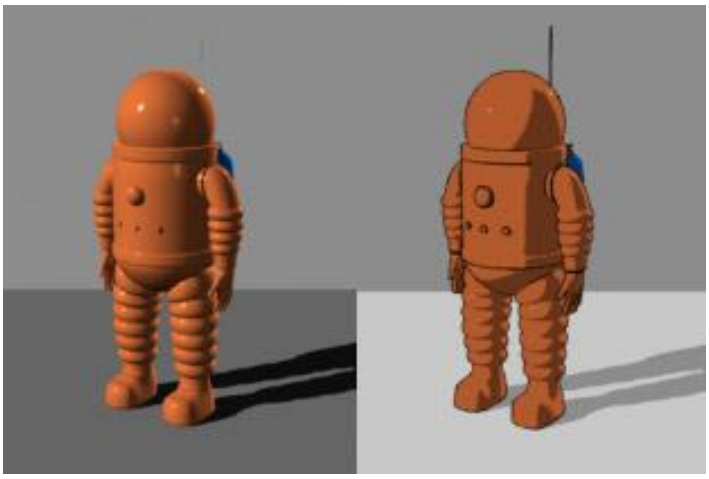


Figura 7 - Exemplo de de NPR com uma toon shader.

Separa contribuições difusas, especulares e de reflexão espelhada da renderização, quantifica-as de gradientes suaves em faixas escalonadas. O sombreamento difuso é o mais importante para que se possa ajustar o número de faixas, os níveis de sombreamento em que as bandas aparecem e até suavizá-las ou tinturá-las individualmente.

Contornos e linhas de contorno pretas podem ser criados usando uma variedade de métodos.

- Halftone/Dot Shader

O *halftone* é uma técnica usada para reproduzir imagens sombreadas com apenas uma cor de tinta. Ele funciona criando células de meio-tom para cada pixel preenchendo parcialmente essa célula com uma cor, para que à distância pareça ter a luminosidade correta. As células de *halftone* são normalmente desenhadas usando círculos de tamanhos diferentes ou linhas de espessura diferentes, mas é possível a utilização de outros padrões.

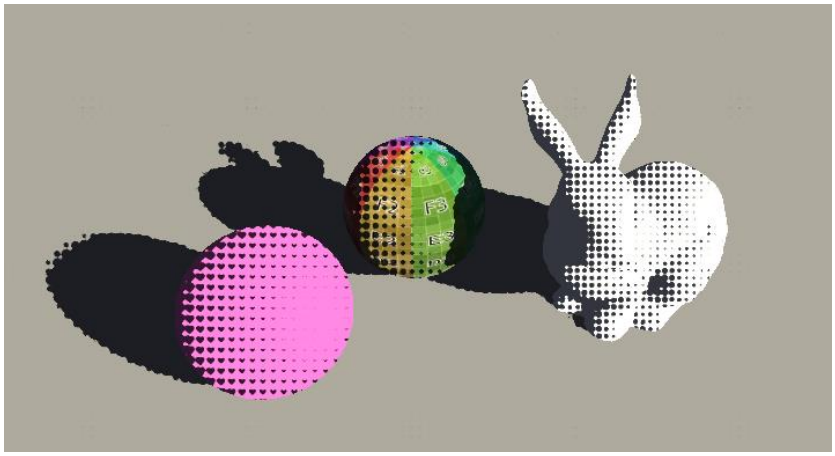


Figura 8 - Exemplos de NPR com a Halftone/Dot shader.

Technical illustrations

Para uma imagem técnica manual, são usadas linhas de borda pretas e sombreamento fosco para alcançar os resultados desejados.

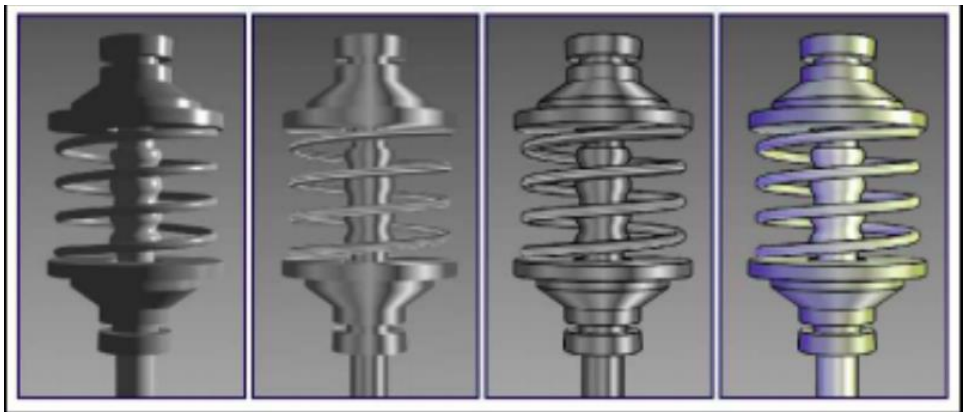


Figura 9 - Phong shading, Metal shading, edge lines, Tone shading.

Convenções em ilustrações técnicas:

- Linhas de contorno preto;
- Cores de sombreamento frias a quentes;
- Fonte de luz única, sombras raramente são usadas.

- Gooch Shader

O remodelador Gooch também substitui o sombreamento renderizado, mas procura produzir sombreamento com um brilho uniforme, onde as áreas sombreadas são pintadas com um azul frio em vez de escurecido, e as áreas brilhantes recebem um tom quente. Isso mantém os detalhes e as linhas das bordas visíveis e mantém as áreas claras distintas de um fundo branco.

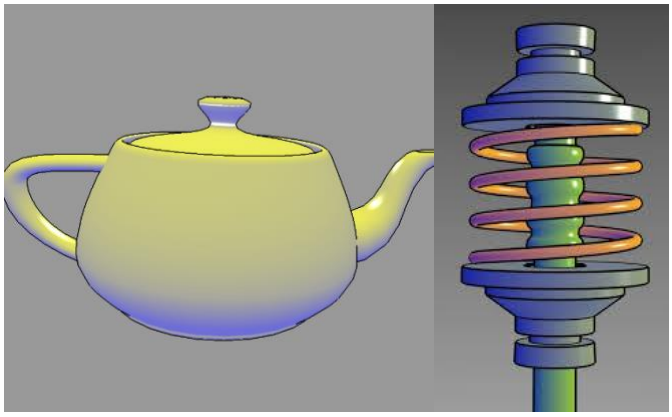


Figura 10 - Exemplo de NPR com a Gooch Shader.

Scientific visualization

A visualização científica usa técnicas de *splatting* e desenho de linha com o objeto de adicionar e aumentar a sua precisão. Isto é usado para visualização eficaz de grandes dimensões multidimensionais de conjuntos de dados.

Figura 11 - Exemplo de NPR para visualização científica.

Implementação e Resultados

Numa fase seguinte, após a análise de todas as técnicas referidas na secção anterior, passou-se à implementação de algumas *shaders* de forma a criar todos os efeitos de desenho supracitados. É de salientar que houve uma tentativa de se implementar pelo menos um *shader* para cada tipo e estilo de NPR.

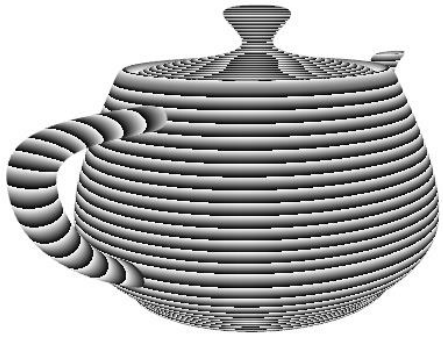
Pen and ink illustrations

- Hatching

Para se criar o efeito de Hatching seguiu-se o seguinte algoritmo para o *fragment shader*:

1. Fez-se com que a intensidade de cada linha variasse entre 0 e 1 e depois voltasse imediatamente a 0, formando uma *sawtooth wave;*

```
float sawtooth = fract((V) * stripes);
```



2. No entanto, esse não era o efeito pretendido. Para que não houvesse mudanças bruscas entre as linhas e fossem mais uniformes, calculou-se a multiplicação do valor de *sawtooth* por 2, subtraiu-se 1 e ficou-se com o valor absoluto do resultado. Dessa forma, a função começava em 1, ia gradualmente para 0 e de seguida regressava a 1.

```
float triangle = abs(2.0 * sawtooth - 1.0);
```

3. Como as linhas não têm todas a mesma espessura, o próximo passo foi garantir que isso acontecesse.

```
float dp = length(vec2(dFdx(V), dFdy(V)));
float logdp = -log2(dp * 8.0);
float ilogdp = floor(logdp);
float stripes = exp2(ilogdp);
float sawtooth = fract((V) * stripes);
float triangle = abs(2.0 * sawtooth - 1.0);
float transition = logdp - ilogdp;
triangle = abs((1.0 - transition) * triangle - transition);
```

4. De modo a criar o sombreamento no objeto, calculou-se a intensidade da luz para modificar o *thresholder* usado na função *smoothstep*. Nas áreas mais iluminadas o *threshold* diminui, tornando as linhas escuras mais finas,

enquanto que nas áreas sombreadas, o *threshold* aumenta e as linhas pretas ficam mais espessas.

```
float LightIntensity = max(dot(DataIn.l_dir, DataIn.normal), 0.0);
const float edgew = 0.3;  // width of smooth step
float edge0  = clamp(LightIntensity - edgew, 0.0, 1.0);
float edge1  = clamp(LightIntensity, 0.0, 1.0);
float square = 1.0 - smoothstep(edge0, edge1, triangle);
```

Figura 12 - Resultado final da renderização usando a técnica de hatching.

- Cross-Hatching com textura

Para criar o efeito Cross-Hatching, começou-se por calcular a intensidade da luz para cada pixel através do produto escalar entre o vetor normal normalizado com o vetor da direção da luz.

```
vec3 n = normalize(DataIn.normal);
float intensity = max(0.0, dot(n,DataIn.l_dir));
```

Figura 13 - Resultado final da renderização usando uma técnica de *cross-hatching*.

Depois, para cada intervalo de intensidade da luz, usou-se duas texturas e foi feito uma interpolação linear entre as duas, de modo a ter uma transição suave entre estas.

- Cross-Hatching

Para se criar este efeito de cross-hatching, usou-se como principal variável, a intensity que indica a quantidade de luz que recebe.

Ao algoritmo funciona da seguinte forma:

1. Desenha o branco por todo o objeto.
2. Vai desenhando o preto consoante a quantidade de intensidade de luz de forma a criar o efeito de cross-hatching:
    - Desenha linhas retas em diagonal da esquerda para a direita em todo o objeto, ou seja, onde a intensidade é menor que 1.0;

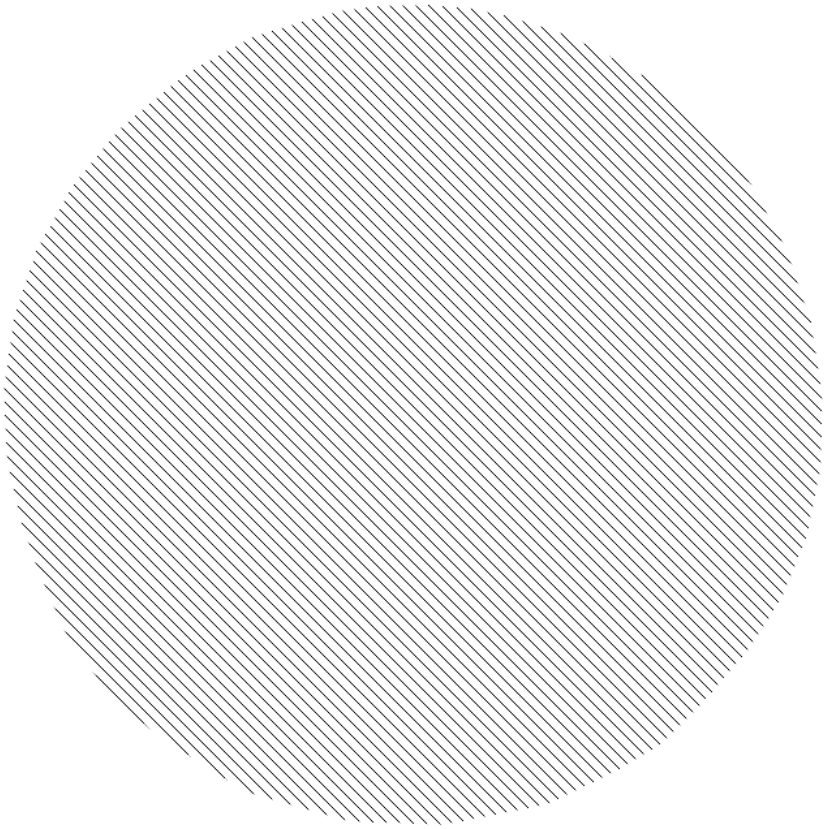


Figura 14 - Resultado da renderização quando *intensity* < 1.

- Até à intensidade menor que 0,75 ele desenha linhas retas em diagonal da direita para a esquerda, criando o primeiro cruzamento de linhas;

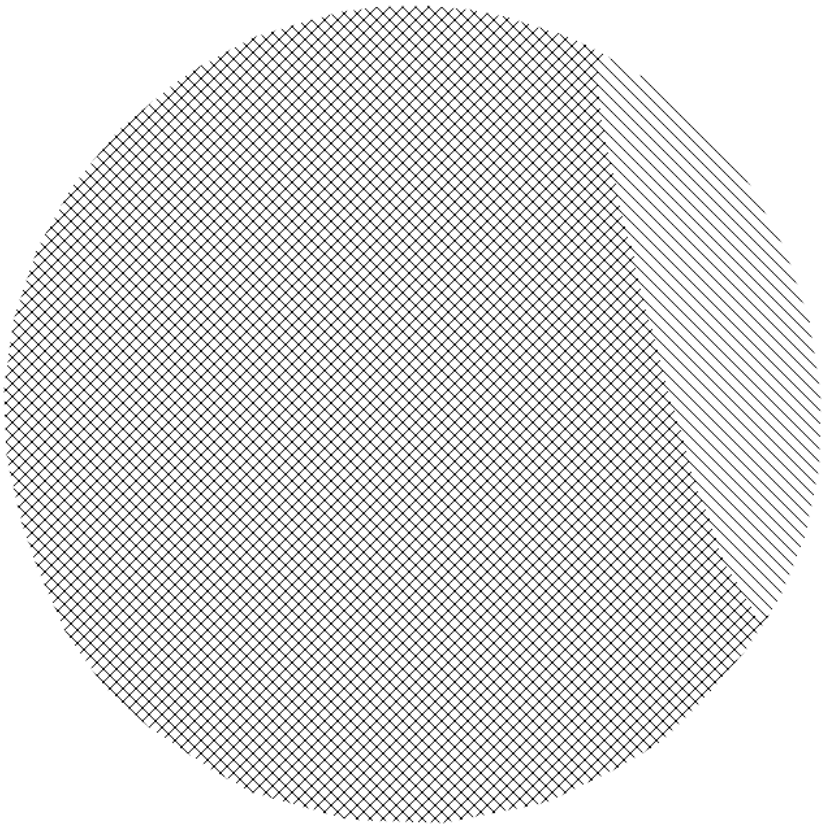


Figura 15 - Resultado da renderização até quando *intensity* < 0.75.

- Até à intensidade menor que 0.5, desenha novamente linhas retas diagonais da esquerda para a direita, mas ligeiramente mais abaixo que as primeiras, criando mais um cruzamento de linha que produz um efeito de sombreado ao objeto.

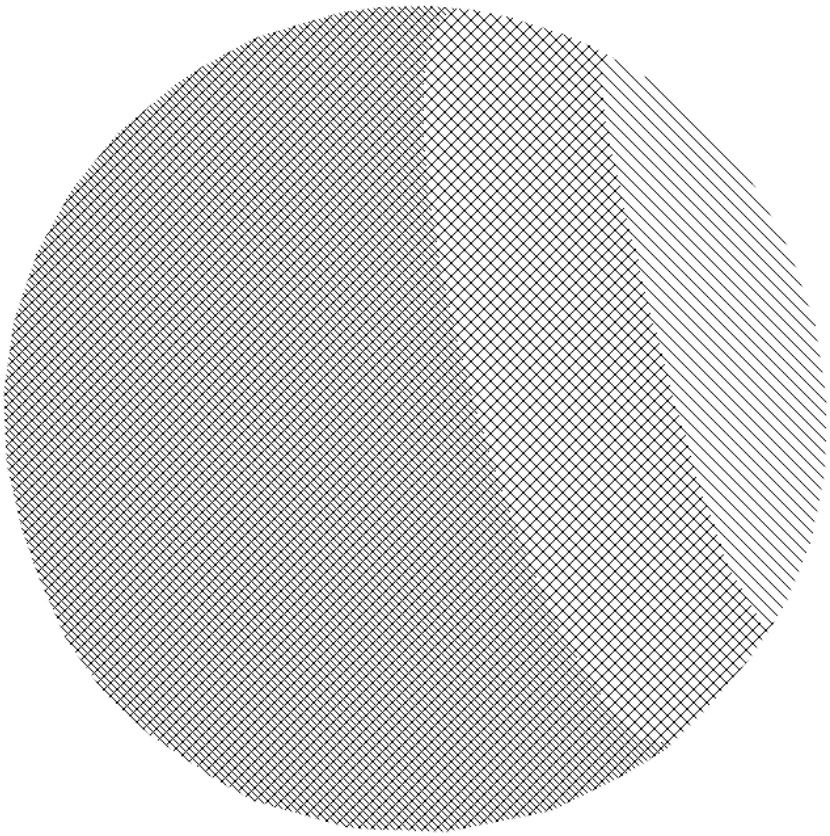


Figura 16 - Resultado da redenrização até quando *intensity* < 0.5.

- Por fim, desenha as últimas linhas diagonais da direita para a esquerda, também mais abaixo que as duas primeiras, para a intensidade menor que 0.25;

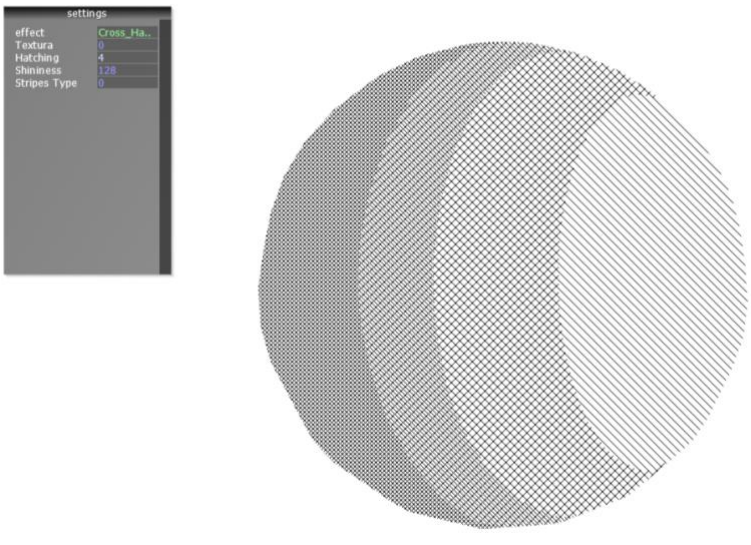


Figura 17- Resultado da renderização da *shader* com a técnica cross-hatching.

Painterly Illustrations

- *Oil painting effect shader*

Para este *shader* em específico usou-se o filtro de Kuwahara. O filtro de Kuwahara é um filtro não linear para reduzir ruído e preservar contornos, diferentemente dos filtros tradicionais lineares passa-baixo que normalmente reduzem o ruído, mas causam *blur* nos contornos.

Ao contrário de um filtro comum que usa 1 kernel, o de Kuwahara usa 4 kernels, um para cada bloco relativo ao pixel, todos tendo *overlap* de 1 pixel como mostra a seguinte figura 18.

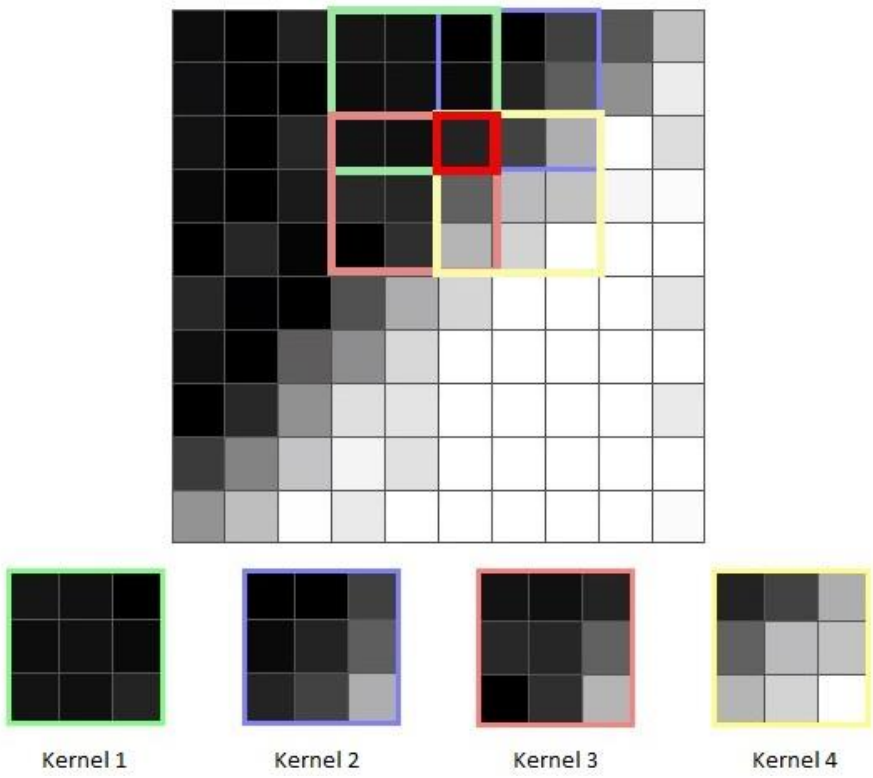


Figura 18 – Representação dos quatro kernels de Kuwahara.

O *shader* tal como o filtro de *Kuwahara* funciona da seguinte forma:

1. Criou-se 4 *kernels* e escolheu-se um raio, sendo este raio o tamanho dos kernels. Um raio de tamanho 2 significa que os kernels são de 4 pixéis.
2. De seguida, para cada um dos *kernels*, vai-se percorrer os pixéis dentro deste raio, utilizando dois ciclos e guardado a soma destes valores e também do quadrado das cores. Isto vai ser posteriormente usado para calcular a média e variância que se vai precisar para determinar a cor final dos pixéis.
3. O próximo passo, tendo os valores somados dos pixéis no raio que se escolheu dentro de um vetor do tipo vec3 (quer-se 3 canais para ter RGB) e tendo em outro vetor similar os quadrados das intensidades, é calcular a média e variância.
4. Selecionou-se uma variância mínima que vai ser utilizada para escolher um dos kernels, essa variância vai ser depois modificada dependendo se é ou não maior que as respetivas variâncias de cada kernel.
5. Calculou-se a variância em si de cada kernel, como se está a trabalhar com RGB (3 canais) deve-se posteriormente calcular o valor total da variância, através da soma da variância dos 3 canais.
6. Só resta comparar esse valor mínimo com as restantes e, a que for menor é a escolhida.
7. No final o pixel fica com a cor do vetor que continha a média com menor variância deles todos.

Algumas notas sobre o algoritmo utilizado:

- O raio utilizado para os *kernels* determina o número de iterações para cada ciclo e para cada *kernel*. Números pequenos dão bons resultados tanto visualmente como em performance, enquanto números grandes reduzem drasticamente a performance e tornam o modelo cada vez mais infiel ao original. Quanto maior o raio escolhido, mais homogéneo ficam as imagens.
- Visto que não foi utilizado paralelização, esse impacto é absoluto.
- O facto deste filtro preservar contornos, deve-se ao facto de se optar sempre pela menor variância, e os *kernels* perto dos contornos costumam ter grande variância, ou seja, não são escolhidos.

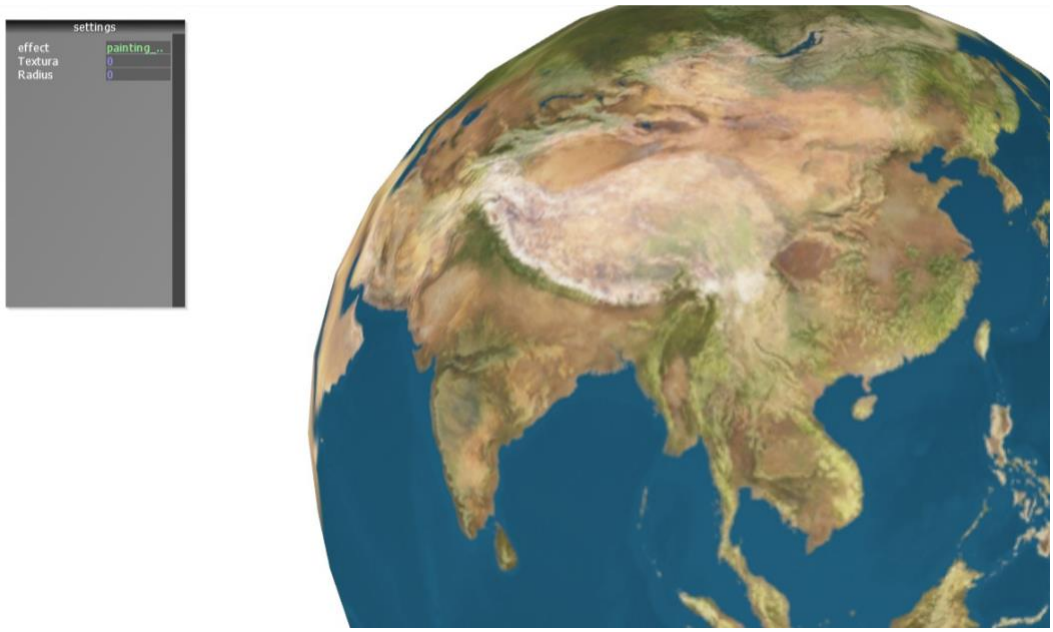


Figura 19 - Modelo original, sem filtro de painterly.

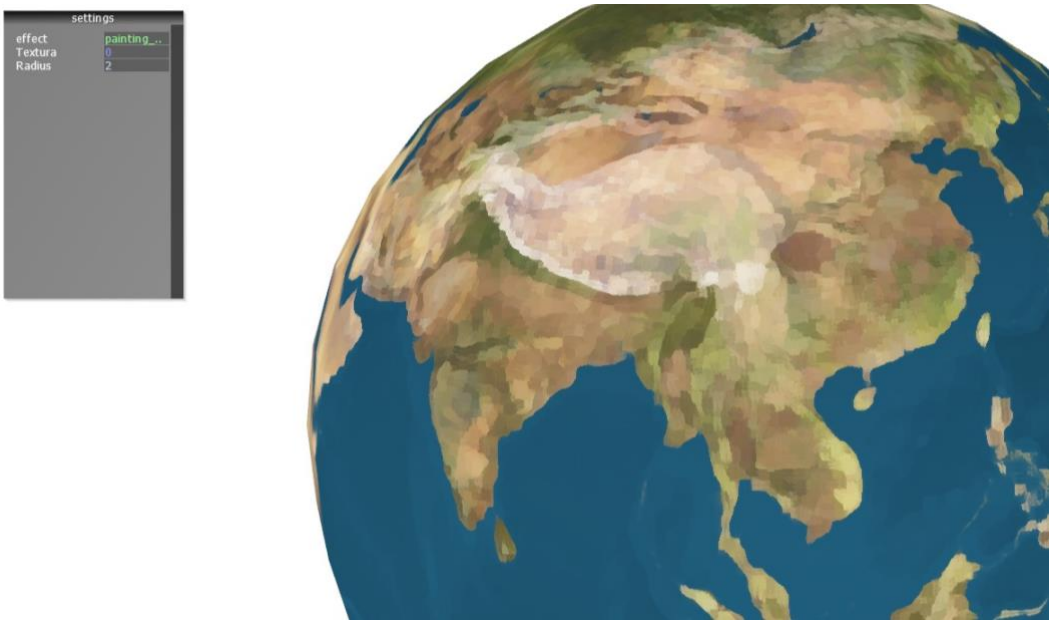


Figura 20 - Modelo com shader de painterly aplicada, de raio = 2.

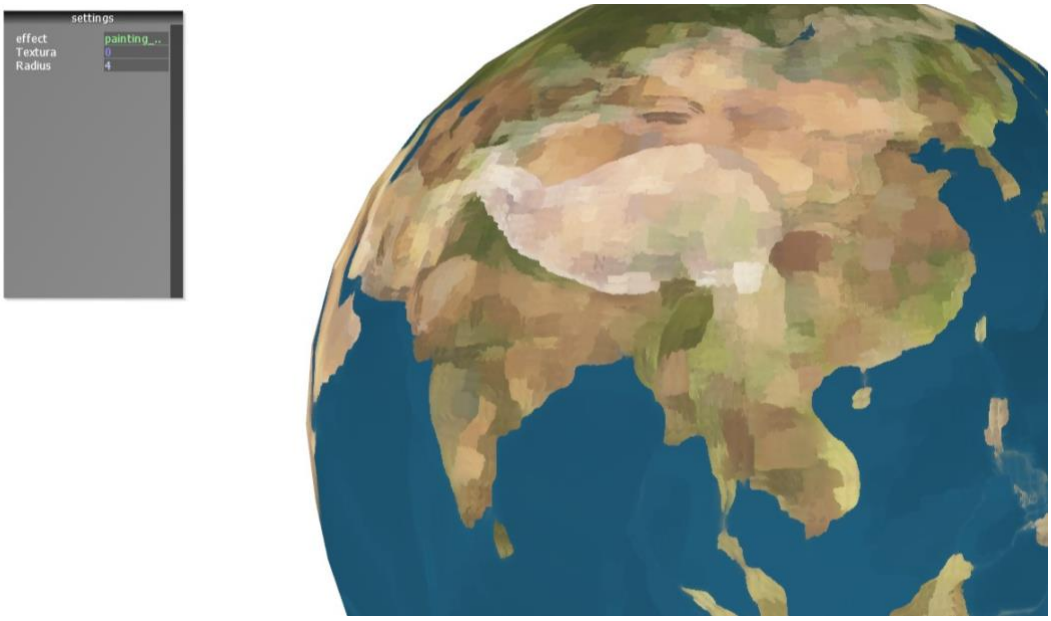


Figura 22 - Modelo com shader de painterly aplicada, de raio = 4.

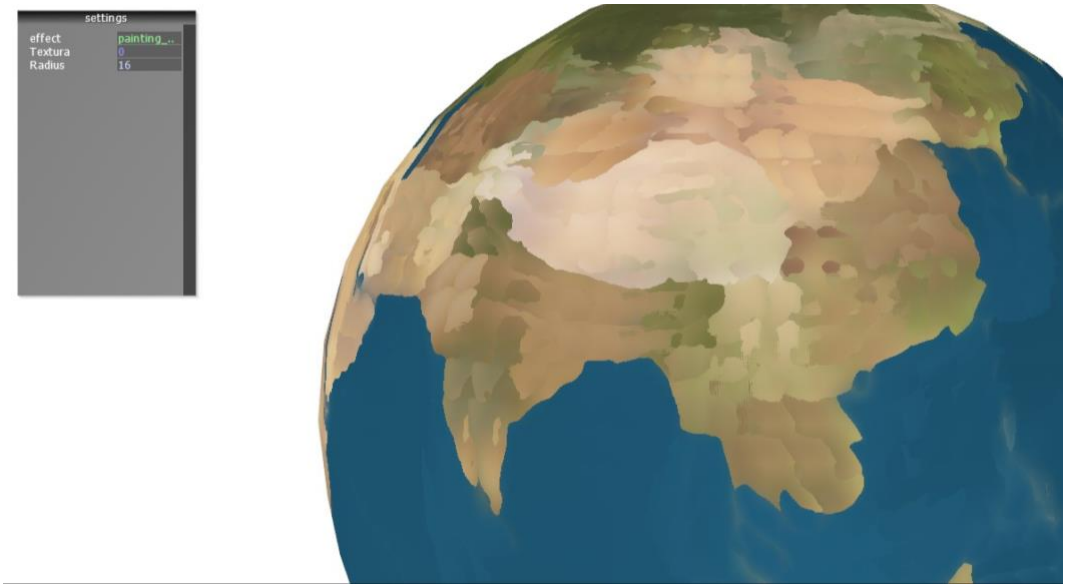


Figura 21 - Modelo com shader de painterly aplicada, de raio = 16.

Como se pode verificar o impacto em performance é enorme visto que cada vez que aumentamos o raio dos *kernels*, está-se a aumentar o número de iterações de cada ciclo, além disso cada *kernel* está a percorrer RAIOxRAIO pixéis. Passou-se, então, de 2100 fps no rendering da imagem original para 1600 fps, 870 fps e por fim 107 fps para os respetivos raios de 2,4 e 16.

Os melhores resultados obtidos foram para raios entre 2-4, boa performance e visualmente simulam bem uma pintura a óleo.

- Efeito de pintura através do halftone/dot shader.

Para implementar esta *shader*, começou-se por criar uma grelha com círculos pretos igualmente espaçados entre si.

```
void main() {
   vec2 nearest = 2.0*fract(frequency * st) - 1.0;
   float dist = length(nearest);
   vec3 fragcolor = mix(ink_color, white_color, step(radius, dist));
   colorOut = vec4(fragcolor, 1.0);
}
```

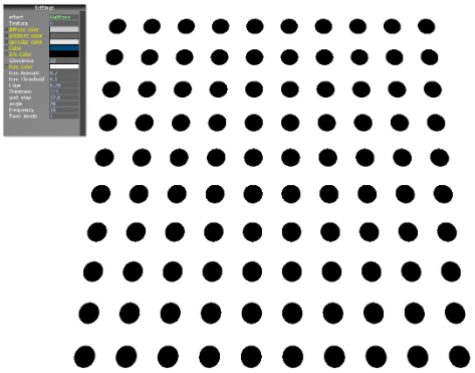


Figura 23 - Grelha utilizada na *shader*.

Uma vez que o olho humano consegue distinguir com maior facilidade linhas horizontais e verticais do que em outros ângulos, é uma prática comum rodar a grelha 45 graus.

```
vec2 st2 = mat2(0.707, -0.707, 0.707, 0.707) * st;
```

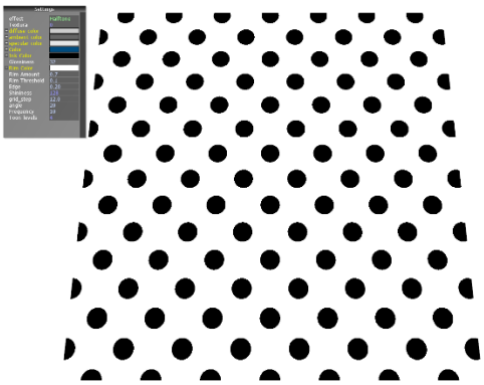


Figura 24 - Grelha após a rotação.

Com o objetivo de reduzir o *aliasing*, utilizou-se a função predefinida *smoothstep* e as derivadas automáticas *dFdx* e *dFdy*.

```
float aastep(float threshold, float value) {
        float afwidth = 0.7 * length(vec2(dFdx(value), dFdy(value)));
        return smoothstep(threshold-afwidth, threshold+afwidth, value);
}

void main {
        …
        vec3 fragcolor = mix(ink_color, white_color, aastep(radius, dist));
        colorOut = vec4(fragcolor, 1.0);
}
```

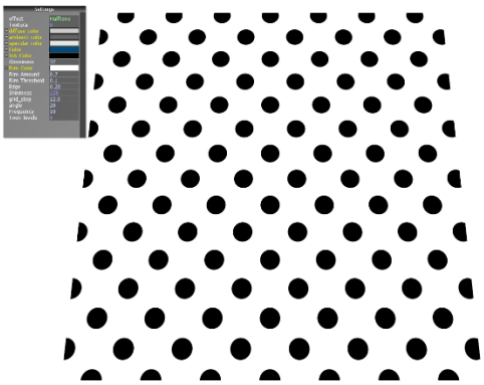


Figura 25 - Grelha após a suavização das bolas.

O passo seguinte foi variar o tamanho dos círculos pretos conforme a cor da textura.

```
vec3 texcolor = texture2D(image, st).rgb;

float radius = sqrt(1.0-texcolor.g);
```

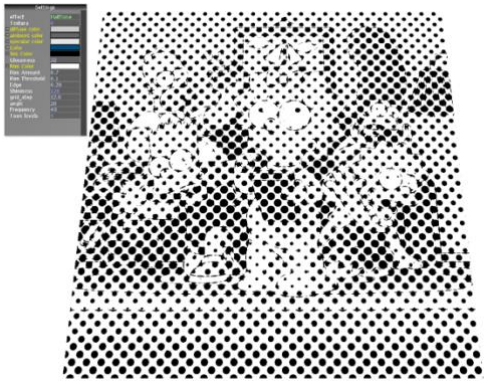


Figura 26 - Resultado da variação do tamanho dos círculos.

Uma vez que se quer simular uma pintura e os círculos não devem ser perfeitos, foi adicionado ruído ao valor do *threshold*, distorcendo as linhas do círculo.

```
float n = 0.1*snoise(st*200.0); // Fractal noise
n += 0.05*snoise(st*400.0);
n += 0.025*snoise(st*800.0);
vec3 fragcolor = mix(black, white, aastep(radius, dist+n));
```

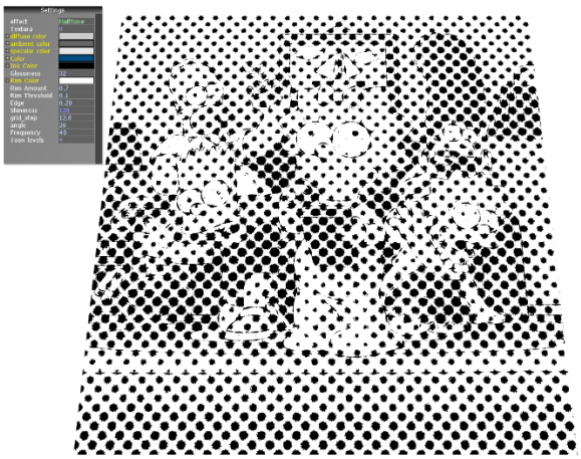


Figura 27 - Resultado da adição do ruído.

Tal como ter círculos perfeitos não era realista, a tinta também não deve ter uma cor homogénea. Por esse motivo foi reutilizado o ruído definido anteriormente para criar uma ligeira variação na cor de fundo e da cor preta.

```
vec3 white = vec3(n*0.5 + 0.98);
vec3 black = vec3(n + 0.1);
```

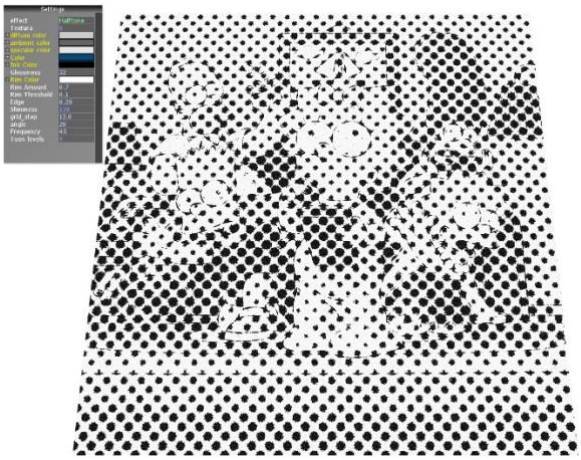


Figura 28 - Resultado da variação na cor de fundo e na cor preta.

As cores são reproduzidas através das cores primárias (CMYK). Os valores das 4 cores são então calculados, e é criado o padrão *halftone*. Os ângulos de rotação usados para as cores ciano, magenta, amarelo e preto são 15, 75, 0 e 45 graus respetivamente.

```
// Converte RGB para CMYK
vec4 cmyk;
cmyk.xyz = 1.0 - texcolor;
cmyk.w = min(cmyk.x, min(cmyk.y, cmyk.z));
cmyk.xyz -= cmyk.w;
…..
// Grelha para a cor preta
vec2 Kst = frequency*mat2(0.707, -0.707, 0.707, 0.707)*st;
vec2 Kuv = 2.0*fract(Kst)-1.0;
float k = aastep(0.0, sqrt(cmyk.w)-length(Kuv)+n);

// o mesmo para as outras cores
…..
```

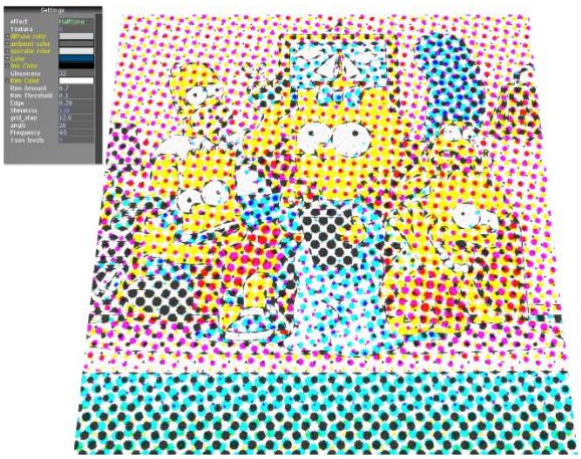


Figura 29 - Resultado da adição de côr à imagem.

Apesar do problema de *aliasing,* ao aproximar a câmara, estar resolvido, ainda é necessário resolver quando há um afastamento da câmara.

Como o objetivo do *halftoning* é que o resultado final se pareça com a imagem original quando visto à distância, foi decidido um ponto de transição onde a imagem original é usada em vez do padrão *halftone,* e executa um *smooth blend* entre os 2.

```
float afwidth = 2.0 * frequency * max(length(dFdx(st)), length(dFdy(st)));
float blend = smoothstep(0.7, 1.4, afwidth);
colorOut = vec4(mix(rgbscreen, texcolor, blend), 1.0);
```

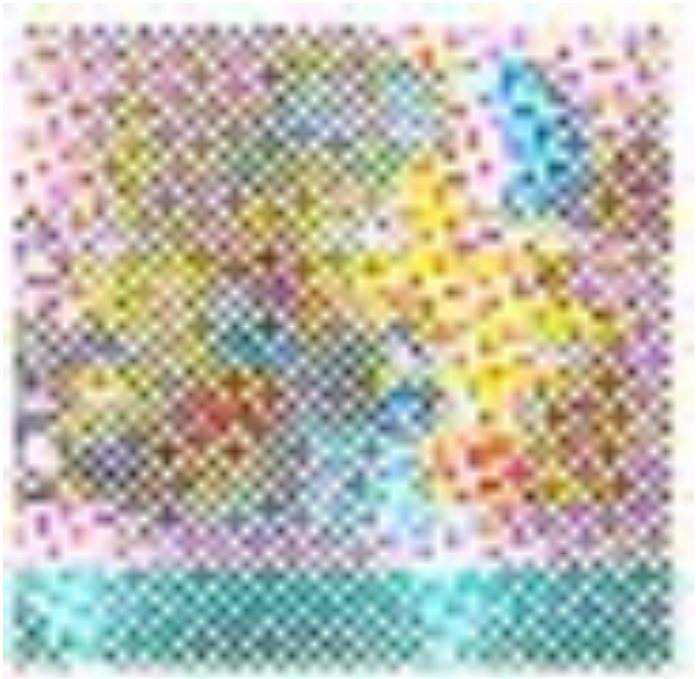


Figura 31 - Visto à distância sem transitar para a imagem original.

Figura 30 - Visto à distância após aplicar *smooth blend*.

Cartoon Rendering

- Toon Shader

1. Luz direcional

Como este *shader* vai apenas interagir com uma única luz direcional, não houve necessidade de se usar *shaders* de superfície. Primeiramente configurou-se o *shader* para receber dados de iluminação do *vertex shader*.

Para calcular a iluminação, usou-se um modelo de sombreamento comum chamado *Blinn-Phong* e aplicou-se alguns filtros adicionais para dar uma aparência de toon.

O primeiro passo é calcular a quantidade de luz recebida pela superfície da luz direcional principal. A quantidade de luz é proporcional à normal da superfície.

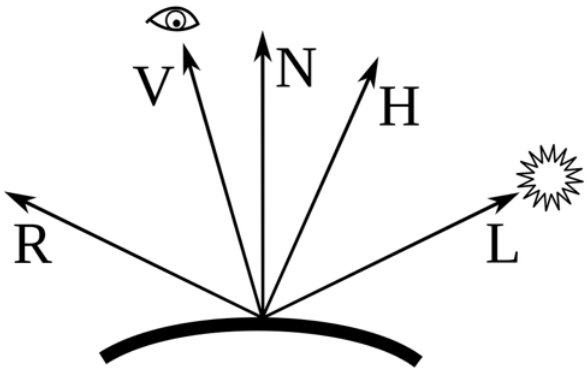


Figura 32 - Vetores *Blinn-Phong*, onde L é o vetor da fonte de luz e N é a normal com a superfície.

As normais são preenchidas automaticamente, enquanto os valores na v2f devem ser preenchidos manualmente no *vertex shader*. Também se quer transformar a normal do espaço do objeto para o espaço do mundo, pois a direção da luz é fornecida no espaço do mundo. Agora estando tudo no mundo normal, pode-se comparar a normal com a direção da luz usando o produto escalar.

O produto escalar recebe dois vetores e retorna um único número. Quando os vetores são paralelos na mesma direção e são vetores unitários, esse número é 1. Caso sejam perpendiculares, o valor é 0.

```
vec3 normal = normalize(DataIn.normal);
float NdotL = dot(DataIn.l_dir, normal);
colorOut = color * NdotL;
```

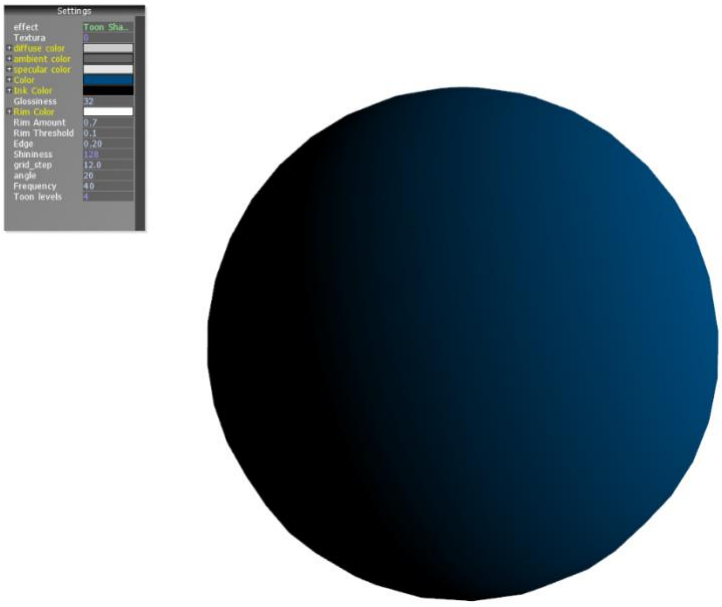


Figura 33 - Resultado do primeiro passo.

No *fragment shader*, vai se dividir a iluminação em duas partes: clara e escura, para ficar uma iluminação irrealista. Caso o valor do produto escalar for maior que 0, fica 1, caso contrário passa a 0.

```
float lightIntensity = NdotL > 0 ? 1 : 0;
colorOut = color * lightIntensity;
```

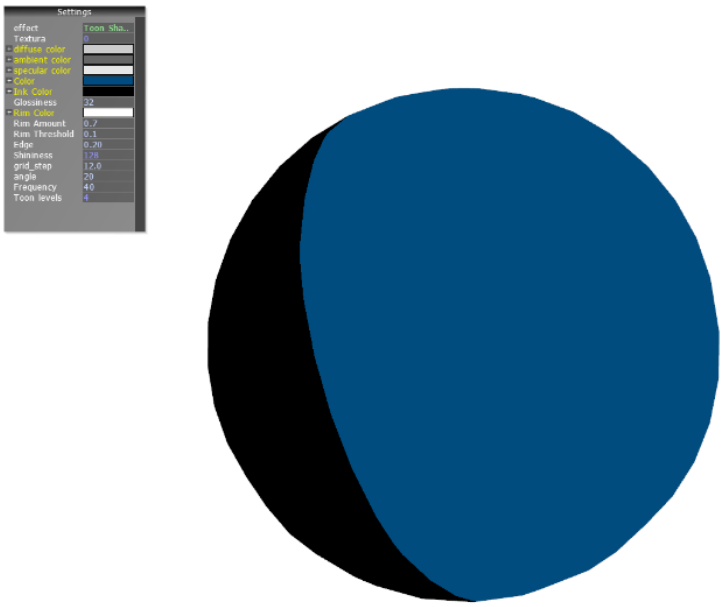


Figura 34 - Resultado da separação da luz em duas partes distintas.

2. Luz ambiente

A luz ambiente representa a luz que se reflete nas superfícies dos objetos e se espalha pela atmosfera. Adicionou-se, então, intensidade e cor à luz direcional.

```
colorOut = color * (ambient + lightIntensity);
```

Figura 35 - Resultado da adição de luz ambiente ao objeto.

Multiplicou-se o valor lightIntensity pela difusa existente e armazenou-se num vec4, para incluir a cor da luz no cálculo posterior.

```
vec4 light = lightIntensity * diffuse;
colorOut = color * (ambient + light);
```

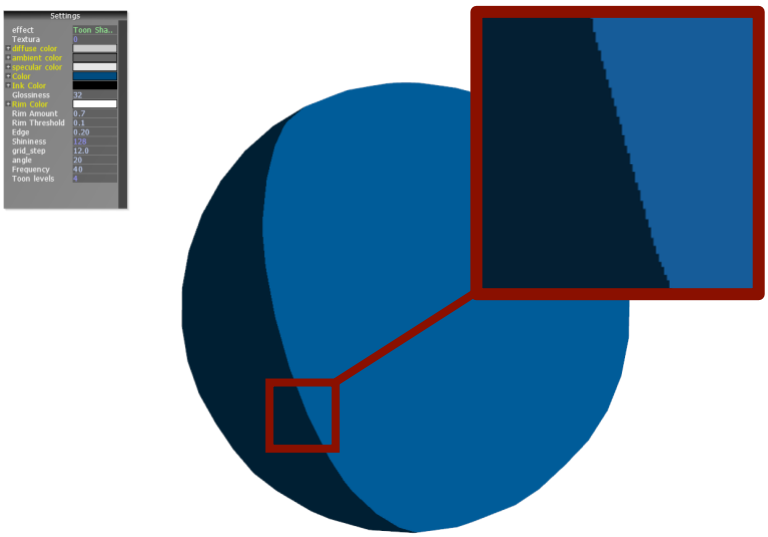


Figura 36 - Resultado da adição da multiplicação da difusa à intensidade de luz.

Passou-se à suavização da borda que separa a cor clara e a cor escura com a função *smoothstep* que utiliza três valores: um limite inferior, um superior e um valor esperado entre esses limites. Esta função não é linear: à medida que o valor se move de 0 a 0,5, ele acelera e conforme se move de 0,5 a 1 faz exatamente o processo contrário. Isso faz com que os valores sejam suavizados, como se pode ver na figura 37.

```
lightIntensity = smoothstep(0, 0.01, NdotL);
```

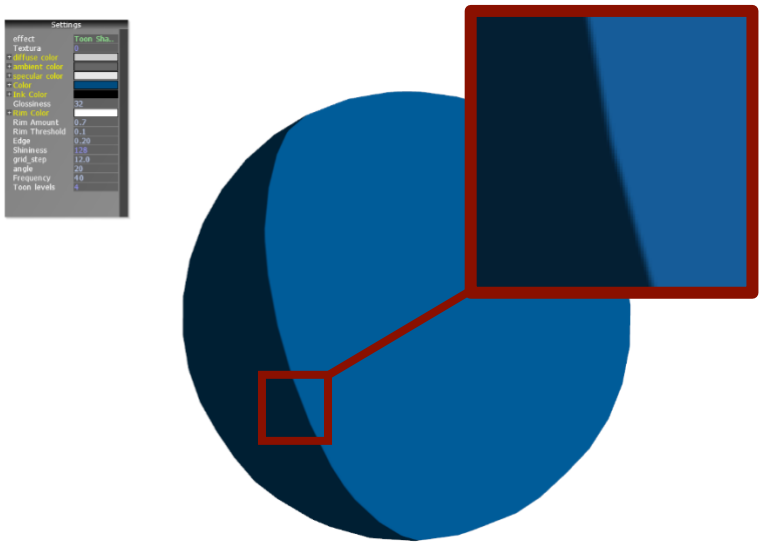


Figura 37 - Resultado da suavização da borda entre as duas cores.

3. Reflexão especular

A reflexão especular modela as reflexões individuais e distintas feitas por fontes de luz. Essa reflexão depende da vista, pois é afetada pelo ângulo de onde a superfície é observada.

Foi necessário então se calcular a direção da visão no mundo, no *vertex shader* e passá-la para o *fragment shader.*

Passou-se à implementação do componente especular de *Blinn-Phong*. Esse cálculo utiliza duas propriedades da superfície, uma cor especular que colora o reflexo e um brilho que controla o seu tamanho.

A força da reflexão especular é definida em *Blinn-Phon* como o produto escalar entre a normal da superfície e o meio vetor. O meio vetor é um vetor entre a direção de visualização e a fonte de luz, pode-se obter esse valor somando apenas esses dois vetores e normalizando o resultado final.

Controlou-se o tamanho da reflexão especular usando a função pow e multiplicou-se o ***NdotH*** pelo ***lightIntensity*** para garantir que a reflexão fosse desenhada apenas quando a superfície estivesse iluminada.

```
vec3 viewDir = normalize(vec3(DataIn.eye));
vec3 halfVector = normalize(DataIn.l_dir + viewDir);
float NdotH = dot(normal, halfVector);
```

```
float specularIntensity = pow(NdotH * lightIntensity, glossiness * glossiness);
colorOut = color * (ambient + light + specularIntensity);
```

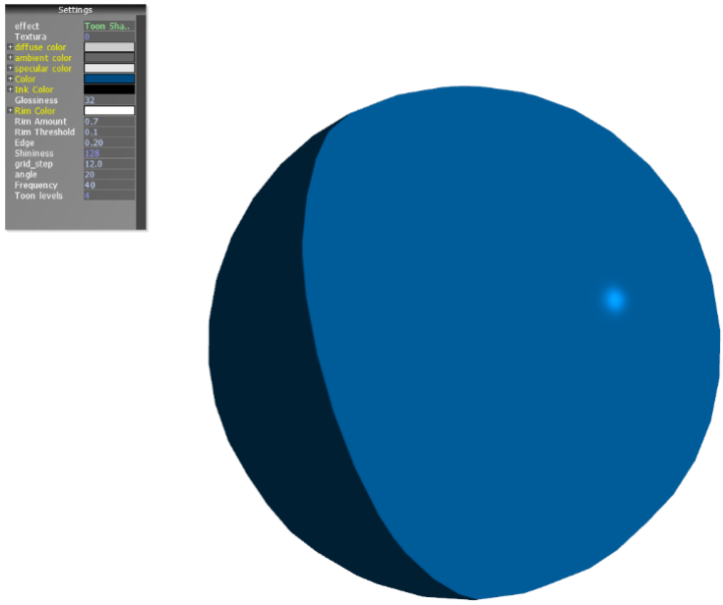


Figura 38 - Implementação do reflexo especular.

Mais uma vez, usou-se o *smoothstep* para tonificar a reflexão e multiplicar a saída final pelo Cor especular.

```
float specularIntensitySmooth = smoothstep(0.005, 0.01, specularIntensity);
vec4 specular = specularIntensitySmooth * specular;
colorOut = color * (ambient + light + specular);
```

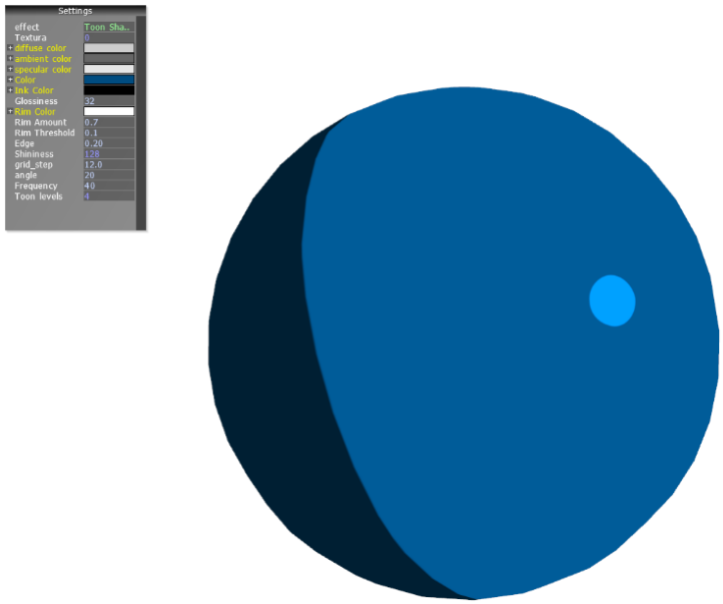


Figura 39 - Resultado da suavização do reflexo.

4. Iluminação da borda

A iluminação de borda é a adição de iluminação ao contorno de um objeto para simular a luz refletida ou a luz do fundo. É útil para *toon shaders* de forma a ajudar a silhueta do objeto a se destacar entre as superfícies sombreadas planas.

Para calcular, inverteu-se o produto escalar da direção normal e da vista.

```
vec4 rimDot = 1 - vec4(dot(viewDir, normal));
colorOut = color * (ambient + light + specular + rimDot);
```

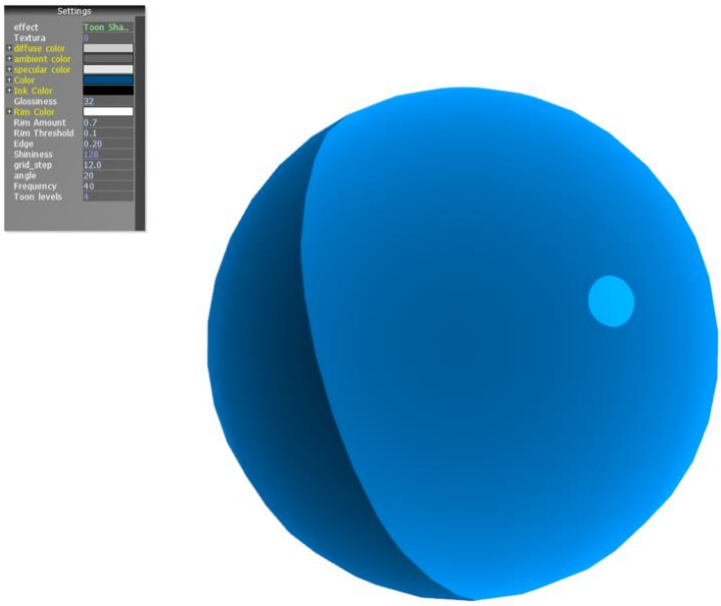


Figura 40 - Resultado da adição de borda.

Mais uma vez, se utilizou o ***smoothstep*** para criar o efeito de *toon*, suavizando a borda.

```
float rimIntensity = smoothstep rimAmount - 0.01, rimAmount + 0.01, float(rimDot));
vec4 rim = rimIntensity * rimColor;
colorOut = color * (ambient + light + specular + rim);
```

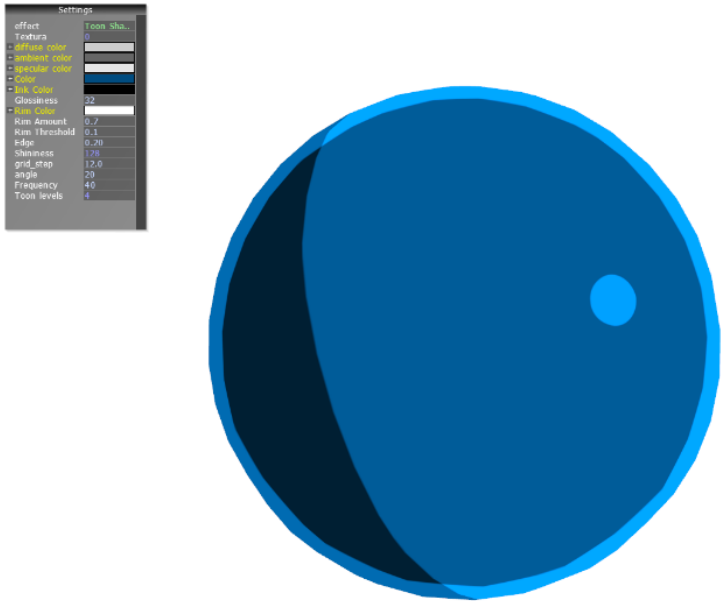


Figura 41 - Resultado da suavização da borda.

O aro desenhado à volta do objeto tende a se parecer mais com um contorno do que com um efeito de iluminação. Modificou-se para aparecer apenas nas superfícies iluminadas do objeto.

```
rimIntensity = float(rimDot) * NdotL;
rimIntensity = smoothstep(rimAmount - 0.01, rimAmount + 0.01, rimIntensity);
vec4 rim = rimIntensity * rimColor;
colorOut = color * (ambient + light + specular + rim);
```

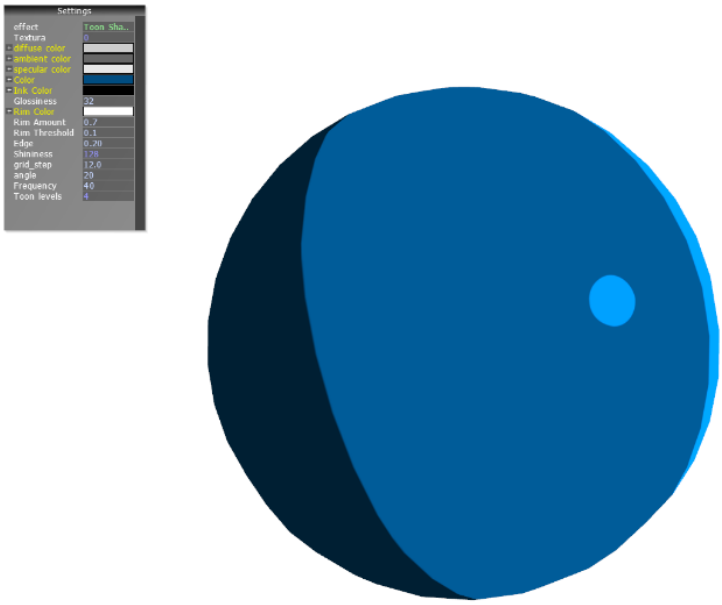


Figura 42 - Resultado da modificação do aro.

Isso melhora bastante o aro, mas seria útil poder controlar a distância que a borda se estende ao longo da superfície Iluminada, então, usou-se a função pow para escalar a borda.

```
rimIntensity = float(rimDot) * pow(NdotL, rimThreshold);
rimIntensity = smoothstep(rimAmount - 0.01, rimAmount + 0.01, rimIntensity);
rim = rimIntensity * rimColor;
colorOut = color * (ambient + light + specular + rim);
```

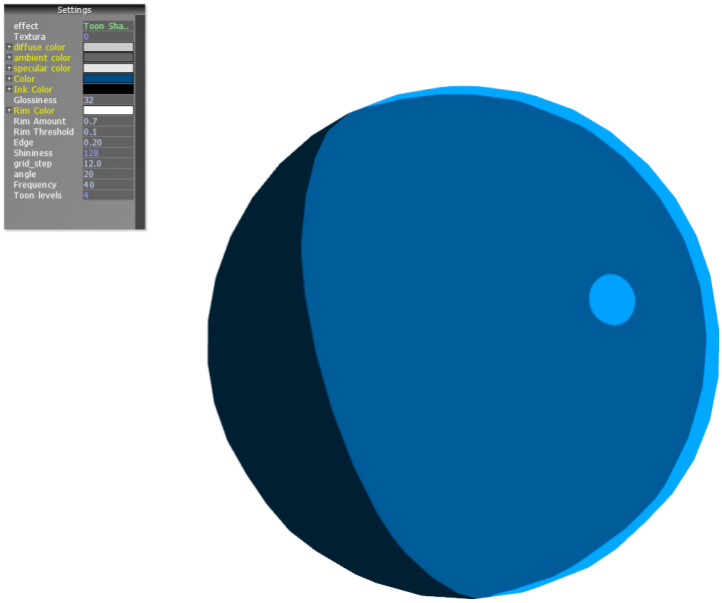


Figura 43 - Resultado final da renderização da toon shader.

- Toon Shading 2

Esta é uma versão alternativa de pós-processamento para criar um efeito de *cartoon* com iluminação "flat".

Uma iluminação convencional cria um gradiente de cores suave, criando um efeito de realismo, neste caso o objetivo é criar o efeito oposto, uma iluminação com um gradiente com menor número de cores possíveis.

O processo para criar este efeito é o seguinte:

1. Escolhe-se um número de zonas (*shades*) que vamos utilizar;
2. Calcula-se a intensidade de luz;
3. Para cada zona vai se verificar a intensidade de luz;
4. Todos os pixels de uma zona ficam com a mesma cor dependendo da sua posição relativa à luz e à intensidade que recebem de luz.
5. Intensidades de maior valor, significam zonas com cores mais claras;
6. A iluminação é agora *"flat"* e o número de cores do gradiente é igual ao número de zonas escolhido.

Figura 44 - Resultado do efeito *toon shader* no objeto *teapot*.

Este *shader* é um bom efeito para simular desenhos animados num ambiente 3D, normalmente depois da aplicação desta técnica aplica-se também uma para realçar os contornos para dar um efeito ainda mais semelhante ao de um desenho. O custo computacional é relativamente menor visto que o número de zonas escolhido para criar o efeito de uma iluminação feita à mão e com gradiente com poucas cores é baixo.

- Toonify

Este *shader* converte um modelo com textura para uma versão *cartoon* semelhante ao *toon shading*.

O algoritmo começa por converter a cor de cada pixel, em RGB para HSV. Na conversão, a cor é mapeada para o conjunto pré-definido de valores guardados nos arrays *HueLevels*, *SatLevels* e *ValLevels*. Após o mapeamento, as cores dos pixéis voltam a ser convertidos para RGB. É verificado para cada pixel se pertence a uma aresta, ou seja, se o pixel tem uma grande diferença de intensidade em relação aos pixéis vizinhos.

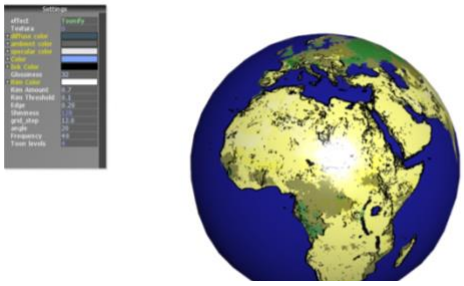


Figura 45 - Resultado da renderização com o efeito *toonify*.

Se um pixel pertencer a uma aresta/borda, então a cor original é ignorada e passa a ter a cor preta. Resultado na figura 45.

- Silhouette Shader

O *Silhouette shader* cria um efeito como se as bordas do objeto estivessem contornadas. Foi relativamente fácil de se implementar, simplesmente foi necessário calcular o produto escalar de dois vetores e caso esse valor estivesse entre o intervalo definido (0.3 a -0.3), o pixel era pintado a preto.

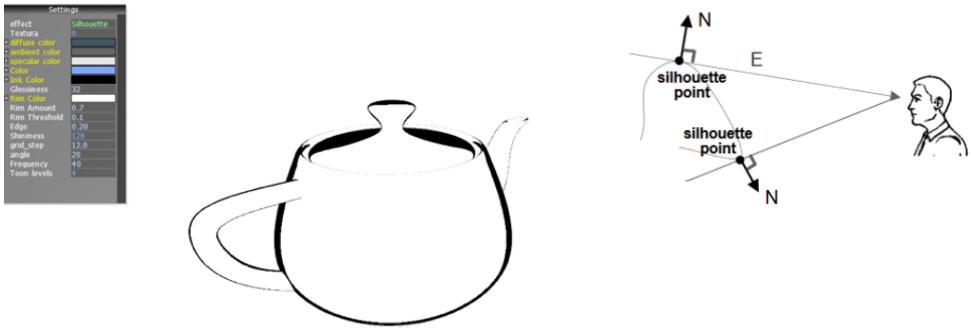


Figura 46 - Resultado da renderização com o Silhouette Shader.

- Halftone/Dot Shader

O algoritmo implementado para esta shader foi o seguinte:

1. Calculou-se o valor difuso produzido pela iluminação da luz. Colocou-se na variável df, a intensidade da luz.
2. Criou-se a função rotate, para rodar a grelha, de modo a criar a aparência desejada.

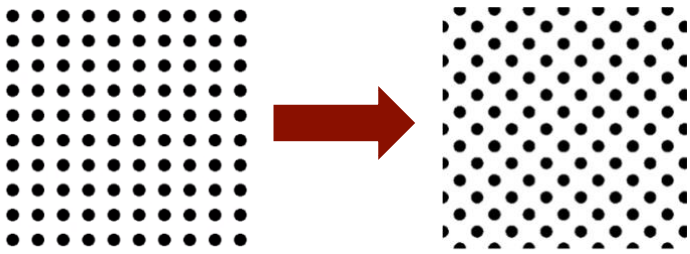


Figura 47 - Rotação da grelha.

3. Criou-se uma função que retorna um valor entre 1 e 0. Este valor indica se o pixel está dentro do círculo (ponto preto).

```
float circulo(vec2 pixel, vec2 centro, float raio) {
return 1.0 - smoothstep(raio - 1.0, raio + 1.0, length(pixel - centro));
```

Esta função vai ser usada para calcular a cor da superfície, caso o pixel esteja dentro do círculo, a variável *cor_final* passa a ser preta, caso contrário mantém a cor inicial ou vai escurecendo gradualmente conforme a intensidade da luz.

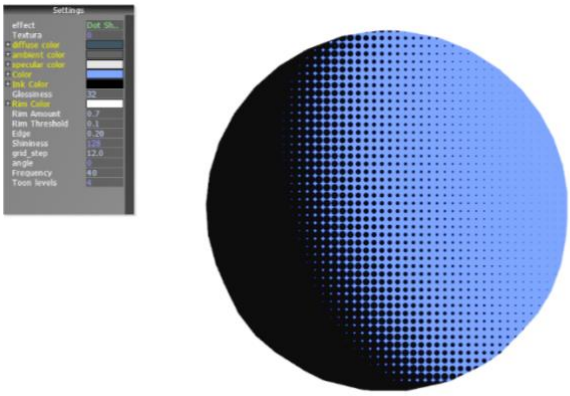


Figura 48 - Resultado da renderização com a Halftone shader.

É de salientar, que é possível se alterar o ângulo e o tamanho da grelha durante a renderização.

Technical Illustration

- Gooch Shader

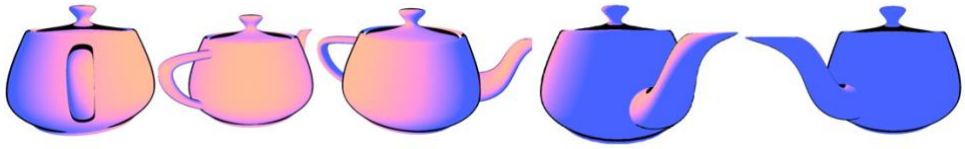
Como em outros *shaders* ou técnicas utilizadas para ilustrar alguma propriedade física, este *shader* é uma boa escolha para mostrar o comportamento da temperatura em vários pontos de um objeto dependendo da sua posição relativa à luz.

Este efeito é criado da seguinte forma:

- Além da cor original escolheu-se também outras duas, uma cor quente e uma cor fria
- Tal como acontece na natureza, as cores quentes (refletem ondas com maior frequência) são os tons de vermelho e laranja e as cores frias são tons de azul e roxo.

- Em vez de usar só a cor original usamos uma combinação das três cores
- Quando mais próximo um ponto está da luz, mais a cor usada é quente e quando mais afastado, mais a cor fica azulada.

Exemplo de vários lados de um objeto com este efeito:



Esta *shader* é bom exemplo se o objetivo for ilustrar mudanças de temperatura causadas por alguma fonte de energia (ex: lareira, lâmpada, sol). Tem pouco custo computacional já que a única mudança é feita em pós processamento, ao combinar as três cores dependendo da intensidade de luz e o código é relativamente simples de se implementar.

Outras Shaders

- Pixelado

A *shader* pixelado é muito utilizada também no meio dos filmes de animação e videojogos como censura de imagem. A técnica de pós processamento para criar um efeito de redução de resolução:

- Escolheu-se um tamanho 'pz' que vai ser utilizado para separar a imagem em quadrados, cada quadrado com pz*pz;
- Quanto maior o tamanho escolhido maior será o efeito de redução de resolução;
- Em vez de mapear linearmente as texturas para o modelo, vai-se mapear cada texel na posição x,y para todo a zona que esta dentro do mesmo quadrado, ou seja para cada x,y mapeamos a cor em x+pz e y+pz;
- A cor escolhida é sempre a referente ao pixel do lado esquerdo inferior;

- Quadrados maiores fazem com que a mesma cor seja mapeada para mais pixels criando um efeito redução de resolução, no entanto a resolução não se altera.

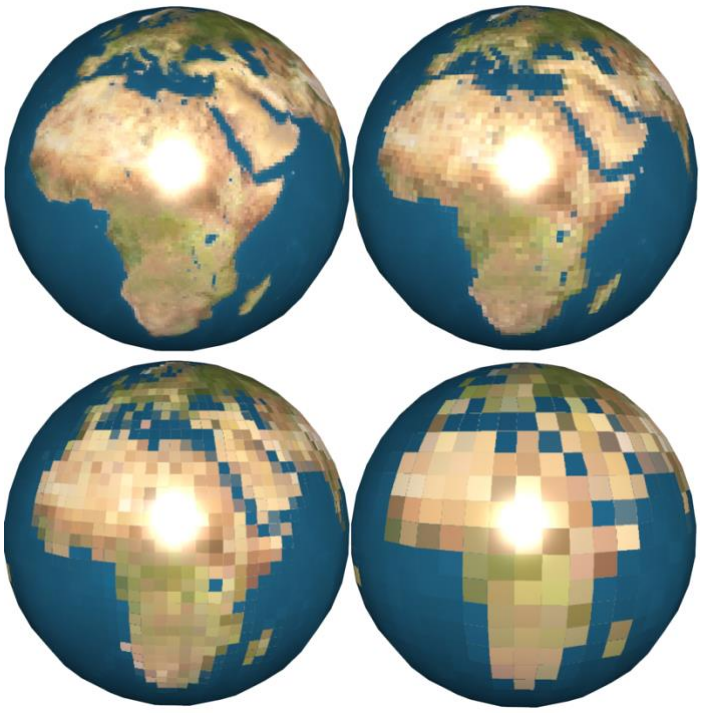


Figura 49 - Exemplo de um modelo com tamanhos de pz 2,4,8 e 32, respetivamente.

A *shader pixelated* é relativamente fácil e rápida de se implementar.

A técnica utilizada tem uma variante em que se faz *down-sampling* da textura em si e usa-se depois um tipo de mapeamento à escolha, sendo essa alternativa ainda mais rápida. Tem um bom desempenho, no entanto só se deve utilizar quando se quer aplicar o efeito para toda a imagem, caso contrário, se o desejado for aplicar só a zonas específicas do modelo, é necessário utilizar outro tipo de *shader* para o mesmo efeito.

Conclusão, observações e trabalhos futuros

Existem inúmeras técnicas que caem neste campo de renderização, na sua grande maioria as várias técnicas permitem grande expressividade artística e abstração da realidade com custo computacional reduzido. Contrariamente, se se comparar com técnicas que visam aumentar o realismo dos modelos e recriar o que se observa na natureza, transpondo isso para um ambiente virtual e discreto, os custos computacionais vão ser muito mais elevados.

Existem ainda muitas outras técnicas que não foram implementadas e sendo NPR um tema que oferece a liberdade e criatividade, existe uma variedade ilimitada de outras *shaders* que poderão surgir no futuro já que ao contrário de tentar atingir e se aproximar ao máximo do realismo, e realidade é só uma, em NPR só se está limitado à nossa imaginação.

No geral as técnicas utilizadas tinham bom desempenho, no entanto foram aplicadas a cenas estáticas, como trabalho futuro poderíamos fazer os mesmos testes a cenas dinâmicas, e comparar o desempenho, e melhorá-lo caso necessário.

Referências

[1] C. d. Wikipédia, "Cel shading," 2019. [Online]. Available: https://pt.wikipedia.org/w/index.php?title=Cel_shading&oldid=56526927.

[2] [Online]. Available: https://www.mrl.nyu.edu/publications/npr-course1999/npr99.pdf.

[3] [Online]. Available: http://www.r3f.com/en/products/npr/index.shtml.

[4] [Online]. Available: https://www.mybluprint.com/article/cross-hatching-techniques.

[5] [Online]. Available: https://danielilett.com/2019-05-18-tut1-6-smo-painting/.

[6] [Online]. Available: http://www.shaderslab.com/demo-63---oil-painting.html.

[7] [Online]. Available: https://www.mathsisfun.com/data/standard-deviation.html.

[8] [Online]. Available: https://www.raywenderlich.com/100-unreal-engine-4-paint-filter-tutorial#toc-anchor-003.

[9] W. contributors, "Kuwahara filter," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Kuwahara_filter.