PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

Criando shaders de pós processamento no Unity Engine

Bernardo Hörner Lopes

Relatório de projeto final de graduação

CENTRO TÉCNICO CIENTÍFICO - CTC
DEPARTAMENTO DE INFORMÁTICA
Curso de graduação em Ciência da Computação

Rio de Janeiro, Junho de 2022

Bernardo Hörner Lopes

Criando shaders de pós processamento no Unity Engine

Relatório de Projeto Final, apresentado ao curso **Ciência da Computação** da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Waldemar Celes Filho

Rio de Janeiro
Junho de 2022.

Resumo

Hörner, Bernardo. Celes, Waldemar. **Criando shaders de pós processamento no Unity Engine**. Rio de Janeiro, 2022. 43 p. Projeto Final – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Shaders de pós-processamento são uma técnica de computação gráfica de baixo custo computacional usada para criar efeitos de tratamento de imagem, aplicados sobre uma cena. O objetivo deste projeto é usar o Unity Engine para criar *shaders* de pós-processamento de efeitos não-fotorrealistas,mantendo um baixo custo computacional. Escolhemos os efeitos de *toon shading*, *hatching* e *outline* para desenvolver, pois esses efeitos ajudam a compor estéticas estilizadas bastante atrativas. O *toon shader* faz objetos 3D parecerem desenhos chapados; o *hatching* simula em objetos 3D a técnica de sombreamento de desenho tradicional feita com riscos; o *outline* desenha um contorno nos objetos de uma cena. Os *shaders* produzidos neste projeto tiveram bons resultados, tendo um bom desempenho de renderização, provando que *shaders* de pós-processamento são uma ferramenta muito poderosa e de baixo custo computacional. Os shaders aqui desenvolvidos podem servir de base para criar efeitos similares, como foi feito com o *toon* e *hatching shaders*. O outline pode ser modificado para criar estilos diferentes de contornos.

Palavras-chave:

Pós-processamento; shader; toon; outline; hatching

Abstract

Hörner, Bernardo. Celes, Waldemar. **Creating post-processing shaders in Unity Engine**. Rio de Janeiro, 2022. 43 p. Final Project - Informatic Department, Pontifical Catholic University of Rio de Janeiro.

Post-processing shaders are a computer graphics technique with low computational cost used to create image treatment effects that are applied over a scene. This project's objective is to use the Unity Engine to create post-processing shaders of non-photorealistic effects that have a low computational cost. We chose to create the effects of toon shading, hatching and outline, because they help to concretize very appealing stylized aesthetics. The toon shader makes 3D objects seem flat drawings; hatching simulates for 3D objects the traditional drawing shading technique of hatching; outline draws a contour on a scene's objects. The shaders produced in this project had great results, having imperceptible impacts in the rendering's performance, proving that post-processing shaders are a very powerful, low computational cost tool. The shaders we've developed can serve as a base from which to create similar effects, as was done with the toon and hatching shaders, that share a lot of their code. There's also interesting options to be explored with the outline, like trying to create different styles of contouring.

Keywords:

Post-processing; shader; toon; outline; hatching

Sumário

1. Introdução

1.1. Motivação

Um jogo eletrônico é uma mídia bastante plural nas áreas de produção envolvidas em seu desenvolvimento. Porém, uma de suas mais notáveis características, que chama e prende a atenção de possíveis jogadores, é, sem dúvida, os *gráficos* do jogo, isto é, o seu visual. Essa área é bastante extensa por si só, tendo diversos aspectos e técnicas que tornam um produto visualmente atraente. Esse projeto visa explorar uma dessas técnicas: *Shaders*, mais especificamente, os de *pós-processamento*. Estes são efeitos de processamento de imagem que são aplicados sobre a imagem renderizada da cena, que é tratada como uma textura. *Shaders* de pós processamento possuem diversos usos para modificar a imagem da cena, como ajustar as cores, aplicar ruídos ou texturas sobre ela, entre vários outros. O próprio Unity Engine [1] já possui diversos efeitos clássicos desenvolvidos e prontos para o uso, como o *Ambient Occlusion*, *Bloom* e *Chromatic Aberration* [2]. *Shaders* de pós processamento, em comparação aos outros tipos de *shader*, costumam ser mais eficientes. Isto porque, diferentemente dos outros tipos de *shader*, não afetam objetos individualmente, somente tendo que processar e trabalhar sobre a textura da renderização da cena. Por conta disso, *shaders* de pós processamento não precisam lidar com objetos que não estão visíveis pela câmera, somente tendo que processar as cores dos pixels da textura da cena.

1.2. Objetivo

Nosso objetivo com este projeto é desenvolver *shaders* de pós processamento, mais especificamente, exploramos técnicas de renderização não foto-realista.

1.3. Contribuição

Nesse projeto nós buscamos trazer alguma melhoria ou inovação para *shaders* cujo conceito e implementações já existem.

Foram implementados 3 *shaders*:

1) ***Toon Shader***

   Cujas contribuições foram:

   a) Implementar o efeito em pós processamento, o que não é muito comum, trazendo os benefícios dessa técnica (eficiência, aplicando efeito sobre toda a cena).

b) Dar bastante flexibilidade para o usuário poder configurar com facilidade os parâmetros do efeito.

2) ***Hatching Shader***

Cujas contribuições foram iguais ao do *toon shader.*

3) ***Outline Shader***

Cuja contribuição foi ter maior precisão dos contornos para objetos distantes da câmera

2. Conceitos básicos

Nesta seção, explicamos algumas noções básicas de especificidades do Unity, necessárias para a compreensão da implementação das técnicas empregadas no restante do documento.

2.1 O que é e porque usamos o Unity

O Unity é um motor de jogo, isto é, um software utilizado para o desenvolvimento de jogos digitais, possuindo diversas funcionalidades úteis, voltadas para a criação de um jogo. Dentre elas, há o motor gráfico, que é “um software especializado em renderização (exibição) de cenas 2D e/ou 3D.” [3]. Optamos por usar o Unity pois é um dos motores de jogos mais utilizados e mais completos do mercado de desenvolvimento de jogos, sendo bastante relevante para profissionais da área de desenvolvimento de jogos atualmente. Além disso, o Unity permite que façamos iterações fáceis e rápidas para *shaders*, pois permite testar os efeitos em diversas cenas diferentes e ver seus resultados rapidamente após editarmos o código de um *shader*.

2.2 Ferramentas e elementos básicos do Unity

No Unity, para montarmos as cenas que compõem um jogo, com os *scripts* de código que serão executados e objetos visuais a serem renderizados, precisamos criar um asset do tipo *Scene* [4]. *Asset* (“recurso”, em português), é como são chamados arquivos e documentos utilizados em um projeto, podendo ser desde um *script* de código C# a um arquivo PNG de uma textura. *Scenes* são *assets* por onde um desenvolvedor trabalha com o conteúdo do jogo no Unity, contendo uma parte ou um jogo inteiro. Para montarmos as *Scenes*, nós utilizamos *GameObjects* e seus *Components*, que os compõem. *GameObject* [5] é a classe base para todas as entidades presentes em uma *Scene* do Unity. Para definir o comportamento de cada *GameObject* em uma cena, atrelamos a eles outros elementos, chamados *Components*. *Components* são classes que herdam da classe *Component* [6], o que as permite que sejam vinculadas a um *GameObject*. Na interface do Unity, há a janela chamada *Inspector* [7], que

permite que o usuário veja e edite propriedades e configurações para quase tudo no editor do Unity, incluindo os *GameObjects* e seus *Components*. Cada *GameObject* na *Scene* pertence a uma *Layer* (camada). As *Layers* são utilizadas para agrupar *GameObjects* e poder definir como outros elementos do jogo interagem com *GameObjects* pertencentes à essa *Layer*. Outra utilização é de simplesmente filtrar os *GameObjects* presentes em uma dada *Scene*. Por exemplo, é possível definir que *GameObjects* da *Layer* "A" não colidem com *GameObjects* da *Layer* "B", ou também fazer um *script* C# que encontra e deleta todos os *GameObjects* da *Layer* "B". A Fig. 1 ilustra esses três elementos básicos na interface do Unity.

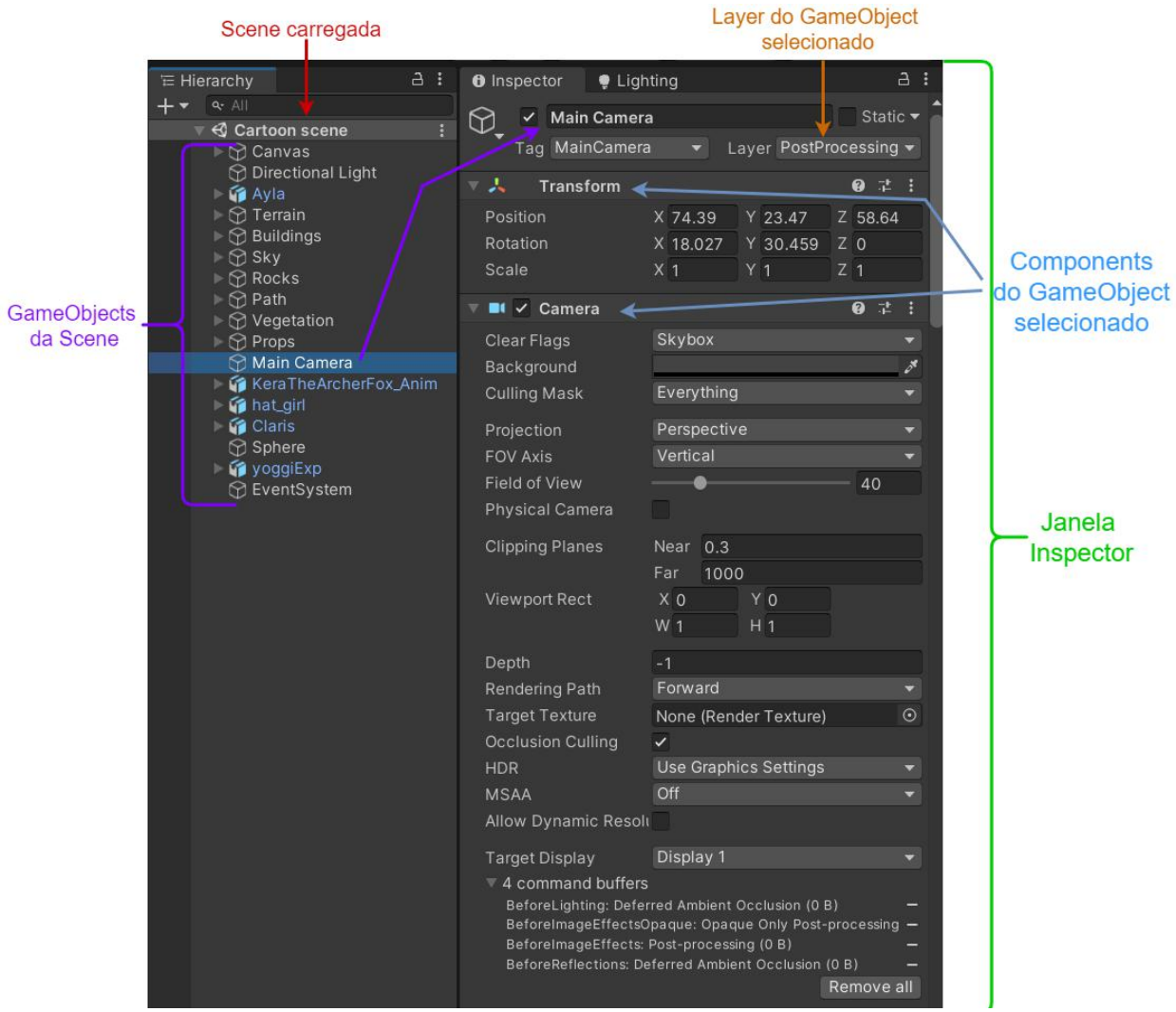

Fig. 1 - Um GameObject selecionado na Scene carregada, com a janela inspector mostrando as suas propriedades, Components acoplados ao GameObject e a Layer a qual o GameObject pertence.

2.3 *Render Pipelines* do Unity

Uma *render pipeline* é uma sequência de operações que pegam as informações de uma *Scene* e as processa para apresentá-las na tela do dispositivo que estiver rodando o jogo. Cada *render pipeline* possui diferentes capacidades e características de performance, cada uma sendo mais adequada para diferentes jogos, aplicações e plataformas. O Unity possui três *render pipelines* configuradas: a ***Built-in Render Pipeline* (BRP)**, ***Universal Render Pipeline* (URP)** e ***High Definition Render Pipeline* (HDRP)** [2]. Para esse projeto, escolhemos a BRP, pois é a mais antiga e mais estável das três.

2.4 O pacote de pós processamento do Unity

Na BRP, para criar um *shader* de pós processamento, utilizamos um pacote oficial do Unity. Pacotes são conjuntos de códigos e *assets* pré-feitos que podem ser baixados e instalados pela ferramenta chamada "*Package Manager*" do Unity. O pacote necessário é o *Post-Processing Stack* v2 [8], oficial do Unity. Tendo ele baixado, utilizaremos dois *Components* desse pacote: o "*Post-process Layer***"** e o "*Post-process Volume*" (também chamarei ocasionalmente os *Post-process Volumes* de "Volumes", para abreviar).

Post-process Layer

É o *Component* que, ao ser adicionado a um *GameObject* que possui uma câmera (*Component Camera*), habilita o uso de pós processamento para essa câmera. A Fig. 2 é um exemplo de um *Post Process Layer*. O *Post-process Layer* permite que o usuário ajuste várias configurações do pós-processamento para uma dada *Layer* e para a câmera a qual está atrelado. Essas configurações são divididas nas seguintes seções:

- **Configurações de *Volume Blending*:**

  Volume Blending, ou "Composição de volumes" em português, é uma forma de misturar efeitos de pós-processamento presentes em diferentes Post-process Volumes no Unity, tendo um resultado combinado de seus efeitos. Há mais configurações de *Volume Blending* nos Volumes, que descreveremos adiante.

  Suas configurações são:

  - ***Trigger***, que permite selecionar qual objeto que irá ativar os Volumes ao entrar em contato com o *Collider* presente no *GameObject* desse mesmo Volume. *Colliders* são *Components* que determinam uma área (se forem 2D) ou volume (se forem 3D) que detectam quando outros objetos entram em contato com eles.

- ***Layer***, que permite selecionar qual a *Layer* na qual os *Post-Process Volumes* estão, para que o *Post-process Layer* consiga buscá-los e acessá-los na *Scene*.

- **Configurações de *Anti-aliasing*:**

  “O *anti-aliasing* (ou antisserrilhamento) é um método de redução de serrilhamento (também conhecido como aliasing), que é o efeito em forma de serra que se cria ao desenhar uma reta inclinada em um computador. Uma vez que a divisão mínima num monitor é de píxeis, surge o aparecimento dos "dentes" da serra ao longo da reta desenhada [9].” O efeito do anti-aliasing é apresentado na Fig. 3. Suas configurações são:

  - ***Mode*,** que permite a seleção de um modo de *anti-aliasing* ou de aplicar nenhum.
  - ***Stop NaN propagation***, que detecta valores de cores inválidos em pixels (valores infinitamente positivos e negativos ou que não tenham valor, conhecidos como valores NaN - *Not a Number*), substituindo eles pela cor preta para evitar artefatos de pós-processamento causados por valores inválidos.
  - ***Directly to Camera Target***, que pode ajudar a melhorar a performance do pós-processamento em aparelhos menos computacionalmente potentes.

- ***Toolkit***

  São apenas algumas ferramentas para auxiliar o debugging e o desenvolvimento dos efeitos de pós-processamento.

- ***Custom Effect Sorting***

  Permite configurar a ordem na qual os efeitos personalizados são renderizados. Isto pode ser bastante útil caso haja um efeito “A” que atrapalharia o resultado do efeito “B”, caso o A renderize antes que B. Com essa configuração, o usuário pode simplesmente fazer com que o efeito A renderize depois de B, evitando o problema.

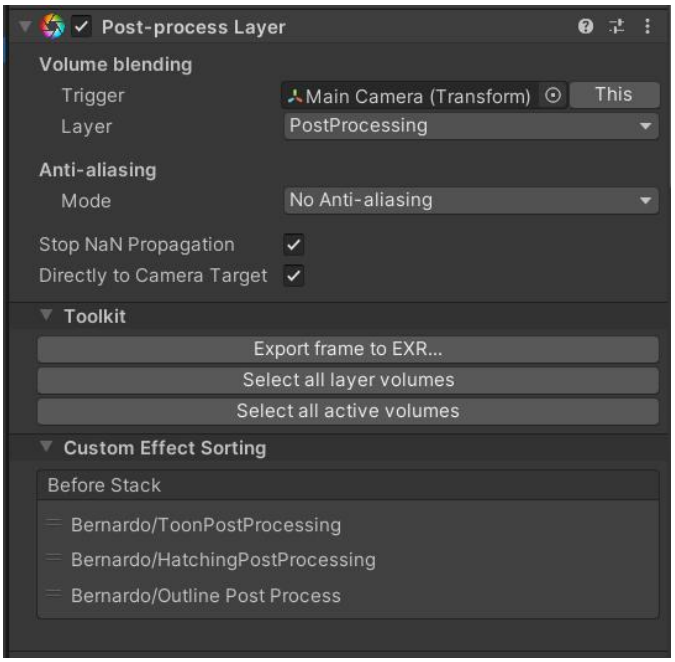

Fig. 2 - Um exemplo do Component "Post-process Layer" e suas configurações.

Fig. 3 [9] *- Imagem extraída da Wikipedia, ilustrando a diferença de uma imagem com e sem anti-aliasing*

Post-process Volume

Controla quais efeitos de pós-processamento estão afetando a câmera de sua *Layer*, além de controlar alguns parâmetros que afetam quando e como a *Post-process Layer* deve realizar a composição dos efeitos que estão afetando sua câmera. A Fig. 4 é um exemplo de um *Post-process Volume*. As configurações dele são:

- ***Is Global***, um valor que pode ser verdadeiro ou falso, indicando se o Volume é global, se verdadeiro; e local, se falso. Se for local, ele requer que o *Target* (definido na seção do *Post-process Layer*) esteja em contato com o *Collider* do Volume para ser ativo. Se for global, o efeito sempre estará ativo.

    - ***Blend Distance***, que determina a distância da superfície ao *Target*, a partir da qual o *Post-process Layer* pode iniciar o processo de composição dos efeitos. Este parâmetro só aparece quando *Is Global* é falso, pois requer que o Volume tenha um *Collider* sobre o qual se basear.
- ***Weight***, que ajusta a contribuição dos efeitos desse Volume à composição final de efeitos.
- ***Priority***, que define a ordem em que esse Volume é executado. Quanto mais alto o valor, mais prioritário é o Volume.
- ***Profile***, que permite definir um *Post Process Profile* a ser usado.
    - ***Post Process Profiles*** são *assets* que contém um conjunto de configurações específicas de *shaders* de pós processamento. São eles que definem quais os valores dos parâmetros a serem passados para um dado *shader* de pós processamento.

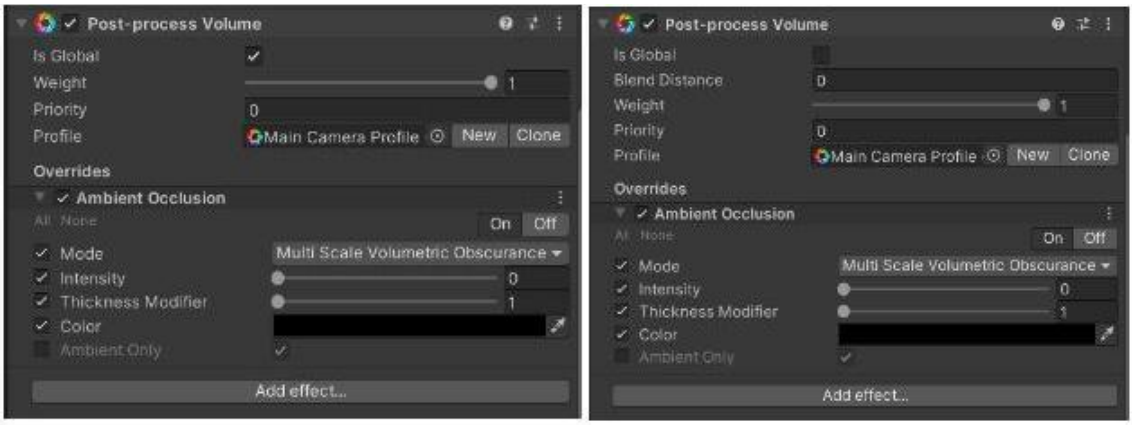

Fig. 4 - Um exemplo do "Post-process Volume" com um "Post-process Profile" com o efeito de Ambient Occlusion e suas configurações. À esquerda, o parâmetro Is Global é verdadeiro e na direita é falso, o que faz o parâmetro Blend Distance aparecer.

2.5 Criando um *shader* de pós-processamento na BRP

Cada *render pipeline* possui uma forma diferente de criar *shaders* de pós-processamento. Como optamos por utilizar a *Built in Render Pipeline* (BRP), as explicações aqui apresentadas terão alguns detalhes que tornam os códigos incompatíveis para outras *render pipelines*. Na BRP, são necessários 2 *assets*: O primeiro é um *script* escrito em C#, que irá expor as configurações e parâmetros do *shader*, a serem manipulados em um *Post Process Profile* e passar esses dados para o *shader*. O segundo é o que possui o código do *shader*, que é escrito em HLSL [10]. HLSL (*High-Level Shader Language*), é uma linguagem utilizada para escrever *shaders* no DirectX. O Unity consegue compilar o código

em HLSL para GLSL, Metal e outras APIs, portanto o código de *shader* não fica limitado ao DirectX [11].

O *script* C#

Esse script é composto de duas classes, que se referem a um único efeito de pós-processamento. Essas duas classes herdam das seguintes classes do Pacote de *Post Processing*:

1) ***PostProcessEffectSettings***

   A classe que herdar desta representará as variáveis configuráveis do *shader*, que ficam expostas para serem modificadas pelo usuário. Para declarar uma nova variável de configuração que pode ser editada pelo *Inspector*, é necessário utilizar as classes do tipo "*ParameterOverride*". Elas basicamente substituem outros tipos básicos e classes comuns, como int, float ou *Color*, sendo nomeadas de "Nome do tipo substituído"Parameter. Por exemplo, há o *IntParameter*, *FloatParameter* e o *ColorParameter*, entre vários outros. Essa classe ainda pode ter variáveis que não sejam *ParameterOverrides*, que inclusive podem ser passadas para o *shader*, com a única limitação de não ser possível editá-las pelo *Inspector*.

2) ***PostProcessingRenderer***

   A classe que herdar desta fará o papel de passar os valores da que herda de *PostProcessEffectSettings* para o código do *shader* via uma terceira classe, chamada "*PropertySheet*", além de também passar a textura da renderização da cena. A *PropertySheet*, em resumo, é uma classe que facilita a passagem de parâmetros, para um *shader*. Ela permite que o usuário adicione valores à ela via seu campo "*properties*", associando a um nome de uma variável presente no código do *shader*. Essa variável no *shader* irá então receber o valor definido no campo properties da *PropertySheet*.

3. Trabalhos relacionados

Nesta seção, apresentaremos alguns exemplos de implementações já existentes, para ilustrar o efeito que almejamos obter e comparar com os nossos resultados.

3.1 *Toon shader*

O *toon shader*, também conhecido como "*Cel Shading*", tem o objetivo de fazer objetos de computação gráfica 3D parecerem chapados, para imitar o estilo de histórias em quadrinhos e desenhos animados. O efeito pode ser obtido

reduzindo a quantidade de tons de cores na cena, fazendo com que, ao invés de haver um gradiente suave entre suas cores, haja uma divisão abrupta. De forma simplificada, os objetos são renderizados com linhas sólidas separando áreas de cores diferentes [12]. Este *shader* é comumente usado em conjunção com o outline *shader*, que desenha contornos em volta dos objetos, para enfatizar a estética “cartunizada”. Explicaremos com mais detalhes o que é o *outline shader* na Seção 3.3.

Exemplos de *toon shaders* já existentes:

Na Fig. 5, podemos ver a comparação do toon shading a uma iluminação difusa com especular, como a iluminação de Phong, que é uma forma clássica de iluminação fotorrealista. Note como as regiões com tons de cores diferentes no padrão fotorealista ficam com divisões distintas no *toon shader*.

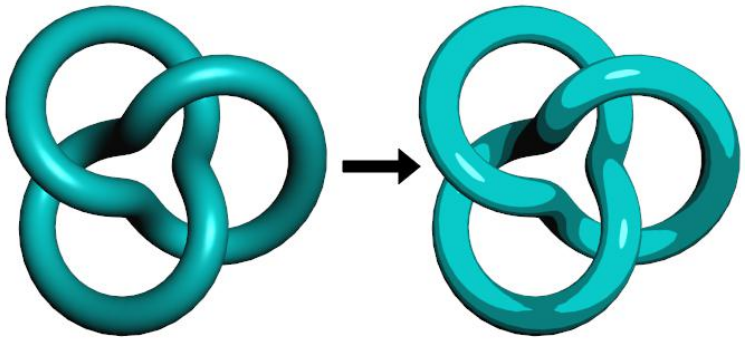

Fig. 5 - Dois torus entrelaçados. À esquerda, com uma iluminação difusa + especular padrão. À direita, com o toon shading aplicado. Extraído da página “Antialiased Cel Shading”, do site “The Little Grasshopper” [13].

Na Fig. 6, podemos ver o jogo “Ni No Kuni” e sua implementação de um *toon shader*. Diferentemente do exemplo da Fig. 5, que usou diversas escalas de tons, no exemplo da Fig. 6 foi utilizado somente dois tons para sombrear os personagens.

Fig. 6 - Captura de tela do jogo Ni No Kuni. Extraído do site “Let’s Play Archive” [14].

A Fig. 7 ilustra a diferença do *toon shading* para uma iluminação mais fotorrealista no jogo Legend of Zelda: Breath of the Wild. Na implementação do toon *shader* para esse jogo, os tons são majoritariamente divididos em somente dois segmentos. Porém, também foi adicionado um pequeno detalhe de iluminação especular em conjunto com o *toon shading*. Este é bem discreto, mas observe como na franja do cabelo do personagem há pequenos pontos iluminados.

Fig. 7 - Captua de tela do jogo Legend of Zelda: Breath of the Wild. À esquerda, o personagem está com o Toon Shader do jogo. À direita, o personagem está com uma iluminação difusa. Extraída do vídeo do YouTube “Zelda Breath of the Wild | Cel Shading Glitch | Punto del juego sin Cel Shading” [15].

3.2 *Hatching shader*

Hatching, hachura em português, "é uma técnica artística utilizada para criar efeitos de tons ou sombras a partir do desenho de linhas paralelas próximas" [16]. O hatching *shader* busca criar justamente esse efeito sobre objetos 3D, substituindo as cores da cena por um pequeno conjunto de texturas de hachuras, cada uma representando como seria uma hachura para um dado tom. Quanto mais escuro é um tom na cena original, mais linhas desenhadas tem a hachura. Chamaremos esse aspecto de "densidade" da hachura. Com isso, conseguimos simular como um desenhista aplicaria essa técnica sobre um desenho.

Exemplos de hatching *shaders* já existentes:

A Fig. 8 ilustra um *hatching shader* aplicado sobre o modelo de um navio. É uma mudança sutil, mas observe como no casco do navio, em seu meio, há uma hachura de menor densidade do que na parte mais para frente do casco.

Na imagem no canto inferior-direito, a região relativa à com menor densidade de hachura é a mais clara, que a de maior densidade de hachura. Observe também como os riscos da hachura estão sempre na mesma direção ou perpendiculares uns aos outros. Esta é uma abordagem do efeito, porém podemos realizá-lo com os riscos acompanhando as superfícies dos objetos, como podemos ver na Fig. 9. Observe como os riscos na barriga do pássaro acompanham a sua curvatura, não estando completamente paralelos uns aos outros.

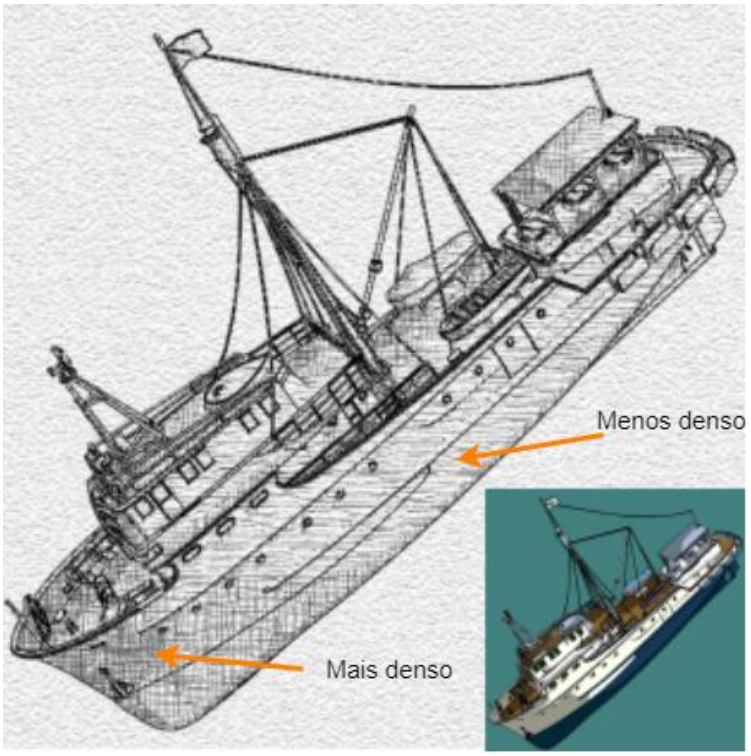

Fig. 8 - Modelo de um navio com um hatching shader aplicado, com os riscos sempre paralelos ou perpendiculares uns aos outros. No canto inferior-direito há o mesmo modelo, porém com um toon shader. Extraído do paper "Stylized Rendering Techniques For Scalable Real-Time 3D Animation" [17].

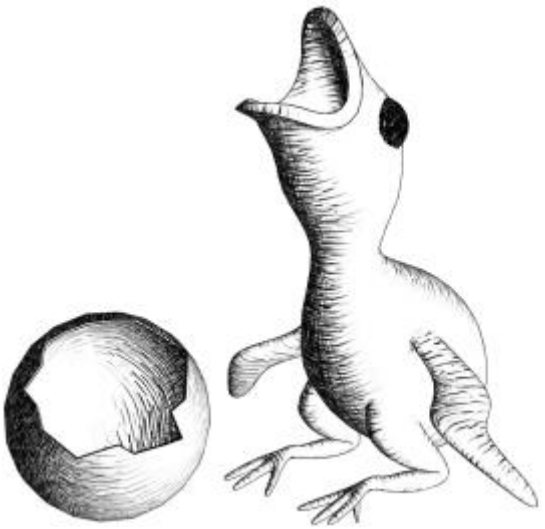

Fig. 9 - Modelo de um pássaro e um ovo com o hatching shader aplicado, com os riscos acompanhando a curvatura dos modelos. Extraído do paper "Real-Time Hatching" [18].

3.3 *Outline shader*

Outline, contorno em português, no contexto deste projeto, é uma técnica utilizada para criar contornos onde há uma descontinuidade da superfície de um objeto, buscando delimitar com mais clareza cada objeto da cena. Este *shader* é comumente empregado em conjunto com um *toon shader*, para evocar ainda mais o estilo de desenhos animados e histórias em quadrinhos, que muito comumente possuem contornos. Essa combinação também é bastante útil para ajudar a distinguir os objetos, que ficam com seus contornos menos definidos pela redução de detalhes de cores da cena.

Exemplos de *outline shaders* já existentes:

Na Fig. 10, podemos ver como a personagem possui um contorno preto desenhado nas descontinuidades de seu modelo. Observe como a há uma linha desenhada tanto na divisão entre sua manga e de sua mão, assim como da ponta do seu sapato em relação ao fundo da cena. Esta abordagem traz um contorno uniforme, que possui a mesma espessura ao longo de todo o objeto. Porém, como ilustrado na Fig. 11, há também a possibilidade de se fazer contornos mais estilizados, como feito no jogo Okami, que busca simular os traços imperfeitos de um pincel, que pode deixar o traço mais espesso ou fino organicamente.

Fig. 10 - Captura de tela do jogo Bomb Rush Cyberfunk mostrando uma personagem com um outline shader aplicado sobre seu modelo. Extraído do artigo "Bomb Rush Cyberfunk é um sucessor espiritual de Jet Set Radio", do site IGN Portugal [19].

Fig. 11 - Captura de tela do jogo Okami, ilustrando um outline mais estilizado para simular os traços disformes e um pincel [20].

4. *Shaders* criados

Nesta seção, descreveremos os *shaders* de pós processamento desenvolvidos neste projeto.

4.1 *Toon shader*

Esta seção explica qual foi a nossa abordagem para implementar o toon *shader* via pós-processamento.

Parte 1: Transformar as cores da cena do modelo RGB para HSV

Para conseguirmos identificar os diferentes tons da cena, vamos usar o valor do parâmetro "*Value*" do modelo de cores HSV. Para fazer isso, nós primeiro transformamos as cores da textura da visão da câmera do modelo RGB para o HSV. HSV é uma abreviação para *Hue*, *Saturation* e *Value*, que são os parâmetros que definem as cores nesse modelo. *Hue* é a matiz, *Saturation* é a saturação e *Value* é a luminosidade. Esse modelo simula como cores aparentam sob uma luz branca. O parâmetro *Value* determina quão intensa seria a luz que está atuando sobre o objeto, tendo um valor real que varia de 0 a 1. Uma luz bastante intensa mostra a cor da superfície mais clara, enquanto uma luz mais fraca escurece seu tom, em seu mínimo, tornando a cor preta [21]. A Fig. 12 ilustra como seria um gráfico para os parâmetros do HSV. Para realizar essa transformação, há uma função pronta, disponibilizada em um repositório público oficial do Unity [22]. O código é o seguinte:

```
    // Hue, Saturation, Value
    // Ranges:
    //  Hue [0.0, 1.0]
    //  Sat [0.0, 1.0]
    //  Lum [0.0, HALF_MAX]
    float3 RgbToHsv(float3 c)
    {
        const float4 K = float4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);
        float4 p = lerp(float4(c.bg, K.wz), float4(c.gb, K.xy), step(c.b, c.g));
        float4 q = lerp(float4(p.xyw, c.r), float4(c.r, p.yzx), step(p.x, c.r));
        float d = q.x - min(q.w, q.y);
        const float e = 1.0e-4;
        return float3(abs(q.z + (q.w - q.y) / (6.0 * d + e)), d / (q.x + e),
q.x);
    }
```

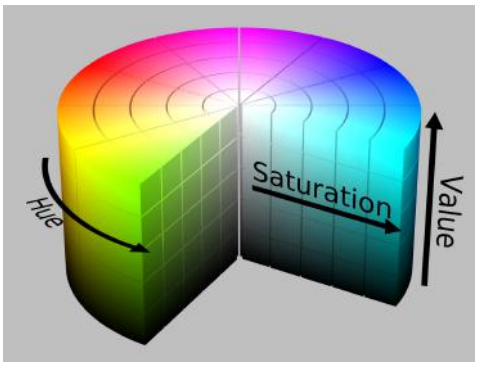

Fig. 12 [21] - *Uma representação do comportamento e influência sobre a cor final de cada parâmetro do modelo de cores HSV.*

Parte 2: Segmentar os tons da cena

Tendo as cores da cena no formato HSV, nosso próximo passo foi segmentar os tons da cena. Para definir a qual segmento cada *Value* pertence, definimos pontos de divisão entre cada segmento, que chamamos de *edges* (limites). Além dos *edges*, também definimos os valores que cada segmento irá representar e os chamamos de *steps* (degraus). Para decidir a qual *step* o *Value* de um dado pixel será atribuído, verificamos, em ordem crescente dos *edges*, se o *Value* é menor que a média entre um *edge* atual e o próximo. Se sim, significa que o *Value* pode estar tanto no *step* imediatamente anterior ou posterior ao *edge* atual. Se o *Value* não for menor que o valor de nenhuma dessas médias, significa que ele pertence ao último *step*. A *Fig. 13* ilustra como é a divisão dos tons para um exemplo de 4 *steps*, dado "sN" um *step* (N sendo um valor de 1 a 4) e "eN" sendo um *edge* (N sendo um valor de 1 a 4). O eixo horizontal é o *Value* original de um dado pixel e o vertical é o valor que é atribuído ao *Value* ao descobrir a qual *step* ele deve ser designado.

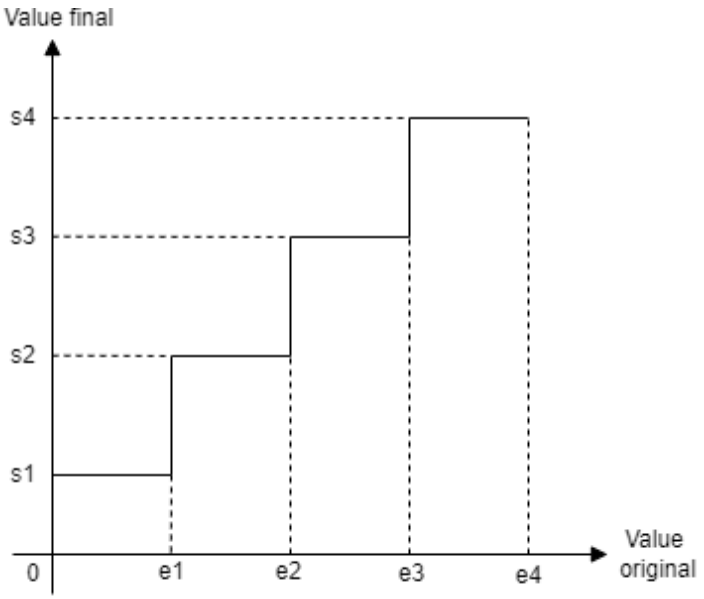

Fig. 13 - Diagrama da lógica de seleção de step do Toon Shader. Exemplo com 4 steps. “e1” ao “e4” são os edges e “s1” ao “s4” são os steps.

Agora, sabendo qual o edge devemos usar, nós precisamos aplicar um tratamento para reduzir o aliasing que pode ocorrer na transição entre tons. Para tal, utilizamos a função de HLSL chamada “smoothstep” [23]. A smoothstep interpola suavemente entre 0 e 1, dentro de um limite inferior e outro superior, para um dado valor, que no nosso caso definimos como o Value original. Se o valor for menor que o limite inferior, o resultado é 0. Se for maior que o limite superior, o resultado é 1. Quão menor o intervalo entre o limite inferior e superior, mais abrupta é a interpolação feita pela smoothstep. Para o efeito do toon shading, é justamente um intervalo bem pequeno de interpolação que queremos, pois assim mantemos a característica transição abrupta entre duas cores do efeito, tendo somente uma sutil transição suave, para evitar o aliasing. Como a Fig. 14 ilustra, para obtermos os valores do limite inferior e superior da smoothstep, nós subtraímos e adicionamos um valor pequeno do edge que selecionamos.

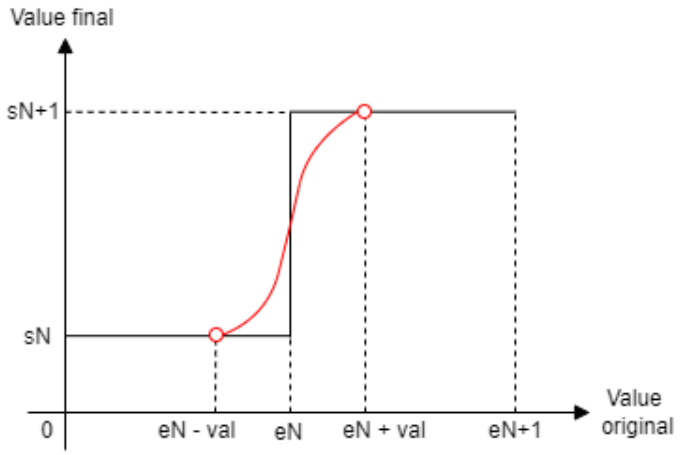

Fig. 14 - Generalização de como a smoothstep seria usada para um dado edge (eN), com o valor (val) a ser subtraído e somado ao edge para obter os limites inferior (eN - val) e superior (eN + val), respectivamente.

Com o resultado do smoothstep, podemos definir o *Value* final para o pixel: se for 0, é o valor do *step* anterior ao *edge*; se for 1, é o valor do *step* posterior ao *edge*; e se estiver entre os dois limites, será uma interpolação do valor do *step* anterior e o posterior ao *edge*.

Parte 3: Transformar a cor de volta do modelo HSV para o RGB

Por fim, nós atribuímos o valor obtido na última etapa ao *Value* da cor do pixel atual, que está em HSV, e traduzimos essa cor de volta para o modelo RGB, obtendo a cor final para o pixel, finalizando o algoritmo. Para realizar essa

transformação de volta, há uma função pronta, disponibilizada em um repositório público oficial do Unity [22]. O código é o seguinte:

```
float3 HsvToRgb(float3 c)
{
    const float4 K = float4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    float3 p = abs(frac(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * lerp(K.xxx, saturate(p - K.xxx), c.y);
}
```

Configurações do toon shader

As configurações disponíveis para o usuário manipular são as ilustradas na Fig. 15. Estes são: **1) *Transition Smoothness***, que define quão suave é a transição entre dois tons, com valores mais altos sendo uma transição mais suave, que é referente ao "val" da Fig. 14; **2) *Steps***, que define a quantidade de *steps* utilizados; **3) *Step 1~7 Brightness***, que define o *Value* a ser atribuído a cada *step*; **4) *Edge1~7***, que define os valores de cada *edge* utilizado;

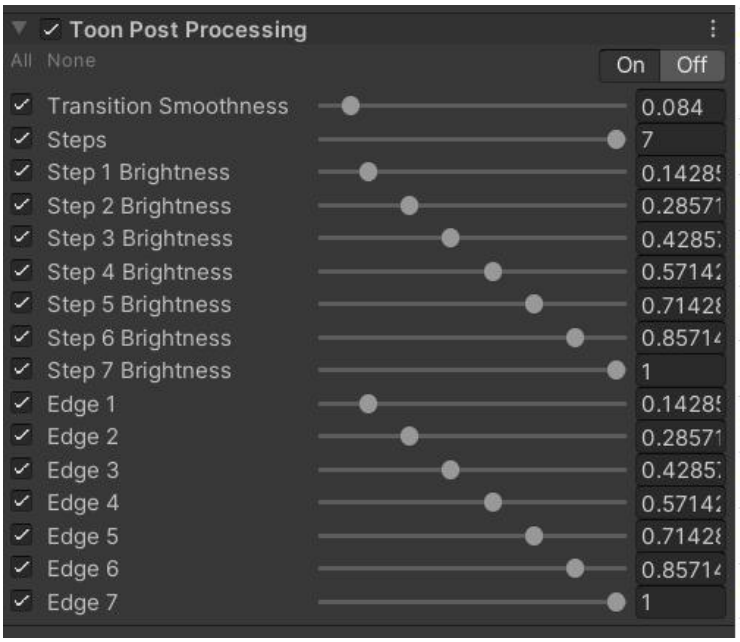

Fig. 15 - Configurações do PostProcessProfile do Toon Shader.

4.2 Hatching shader

Nesta seção, explicaremos sobre o conceito por trás da implementação do *hatching shader* de pós-processamento.

Parte 1: Transformar as cores da cena do modelo RGB para HSV

Esta etapa é idêntica à já explicada na "Parte 2" da seção 4.1.1, do *toon shader.*

Parte 2: Segmentar os tons da cena e aplicar a textura de hachura

Esta etapa é majoritariamente igual à do *toon shader*, exceto por uma questão: Ao invés de termos valores de 0 a 1 associados a cada *step*, que serão atribuídos ao *Value* final da cor, temos um conjunto de texturas de hachura, com cada uma das texturas atribuídas a um *step*. Esse conjunto forma um *Tonal Art Map*, isto é, uma sequência de imagens, cada uma atribuída a um certo tom de cor. As texturas que utilizamos estão ilustradas na Fig. 16. Cada textura representa um nível de densidade de hachura, com as mais densas sendo associadas aos *steps* mais baixos. Além dessas texturas, também utilizamos uma textura completamente branca para representar o menor nível de densidade, pois optamos por não desenhar traços para o *step* de *Value* mais elevado. Outro detalhe diferente do *toon shader* é que o *hatching* se beneficia de ter um intervalo um pouco maior para os limites do smoothstep, para deixar a transição entre as texturas mais suave e trazendo uma aparência mais orgânica ao efeito.

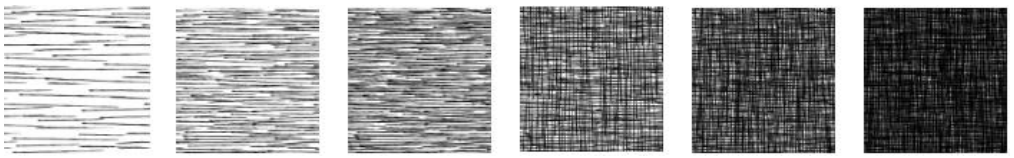

Fig. 16 - As texturas de hachuras utilizadas para o nosso hatching shader, ordenadas da esquerda para a direita da menor para a maior densidade. Elas foram extraídas de um repositório público no GitHub [24].

Parte 3: Aplicar cor base e, opcionalmente, as cores originais da cena

Para finalizar, temos dois tratamentos adicionais que disponibilizamos para o usuário: 1) Usar as cores originais da cena e aplicar a hachura por cima. 2) Cor base, que altera as partes que não possuem uma linha desenhada da hachura (isto é, entre os traços na textura e nas partes sem hachuras). Tendo esses últimos ajustes feitos, obtemos a cor final, atingindo o efeito de hachura que buscamos.

Configurações do *hatching shader*

As configurações disponíveis para o usuário manipular são as ilustradas na Fig. 17. Estes são: **1) *Transition Smoothness***, similar ao do *toon shader*, porém referente à transição entre duas texturas de hachura; **2) *Steps***, que define a quantidade de *steps* utilizados; **3) *Edge1~7***, que definem os valores de cada *edge* utilizado; **4) *Hatching 0~6***, que definem as texturas de hachura a serem aplicadas para cada *step*; **5) *Hatch Tiling Factor***, que determina o tamanho dos

riscos da hachura, com valores maiores deixando os riscos menores; **6) *Base Color***, que determina a cor do fundo; **7) *Use Original Color***, que define se devem ser usadas as cores originais da cena ou não.

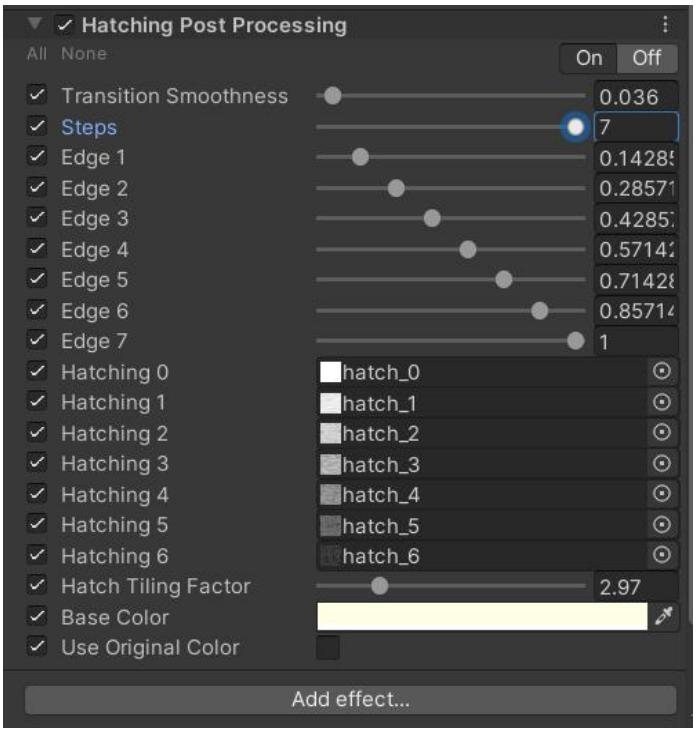

Fig. 17 - Configurações do PostProcessProfile do Hatching shader.

4.3 Outline shader

Esta seção explica qual foi a nossa abordagem para implementar o *outline shader* via pós-processamento. Para conseguirmos identificar as descontinuidades da cena, nós iremos utilizar duas informações da cena: **profundidade (distância) dos objetos em relação à câmera** e **diferença de normais das superfícies**. As normais de uma superfície são os vetores que ficam perpendiculares a cada vértice da superfície de um objeto. Eles representam a direção para qual cada vértice de uma superfície está virado. Implementamos essa técnica nos baseando em um tutorial do Erick Roystan [25]. Como esse efeito possui diversos passos razoavelmente complexos, para facilitar a compreensão, ilustramos o que cada passo está realizando em cima da cena ilustrada na Fig. 18.

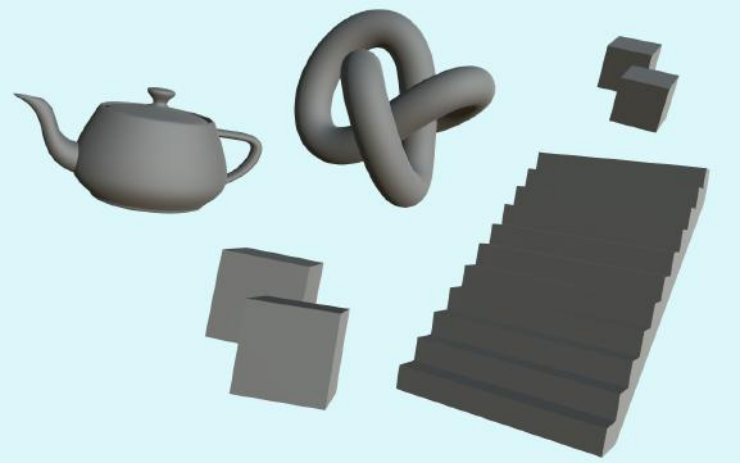

Fig. 18 - Cena sobre a qual ilustraremos os efeitos de cada passo para a criação do Outline Shader.

Parte 1: Obter a profundidade de cada pixel da tela

Para obtermos as informações de profundidade de cada pixel, consultamos a textura de profundidade. Essa é uma imagem especial que pode ser gerada pela câmera, que pinta cada pixel da tela de uma cor que varia de preto (mais próximo da câmera) a branco (mais distante da câmera). Ela é baseada no *depth buffer*, que éuma matriz bidimensional que possui dados referentes à profundidade do objeto mais próximo da câmera, para cada pixel da tela, obtendo uma imagem como a da Fig. 19. Note como os objetos mais próximos da câmera estão em um tom mais perto do branco e o par de cubos ao fundo está mais próximo do preto.

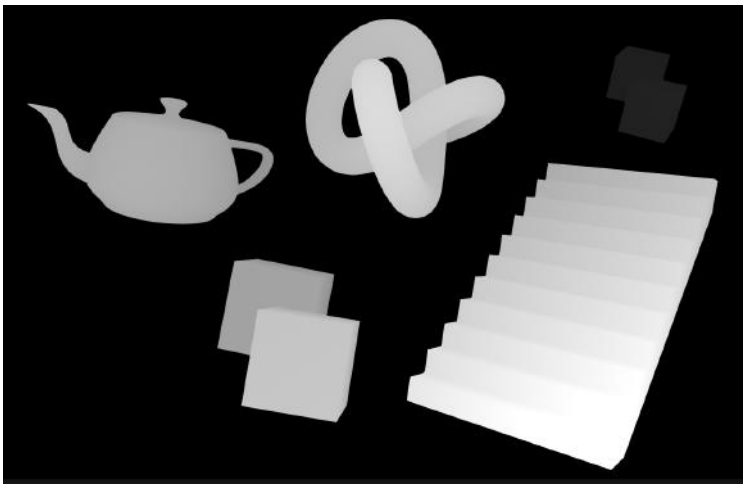

Fig. 19 - Cena da Fig. 18 com as cores substituídas pelas armazenadas na depth texture.

Parte 2: Obter as normais em *view space* dos objetos

Para obter as informações das normais dos objetos na cena, consultamos uma textura de normais em *view space*. *View space* é como é chamado a visão

da cena a partir da perspectiva da câmera. Nesse espaço, a posição e rotação dos objetos são modificadas em relação ao *object space* (coordenadas e rotações dos objetos em relação à origem da cena) para simular a mudança de perspectiva de uma câmera [26]. Essa é outra imagem especial com informações sobre a cena. Porém, diferentemente da textura de profundidade, que já é gerada pela câmera, nós precisamos gerar a textura de normais em *view space*. Para tal, utilizamos um shader e um *script* já disponibilizados pelo tutorial do Erick Roystan. O shader renderiza um objeto com os 3 canais de cores de sua superfície sendo substituídas pelas 3 coordenadas das normais em *view space*. O *script*, por sua vez, cria uma câmera secundária ao executarmos o programa, que irá acompanhar a câmera principal, para que as duas sempre estejam vendo da mesma perspectiva. Essa câmera secundária é usada para renderizar a cena utilizando esse *shader* dos vetores normais e gerar uma textura do resultado. Essa textura é então disponibilizada globalmente para quaisquer *shaders* que precisem usá-la. O resultado dessa operação é ilustrado na Fig. 20.

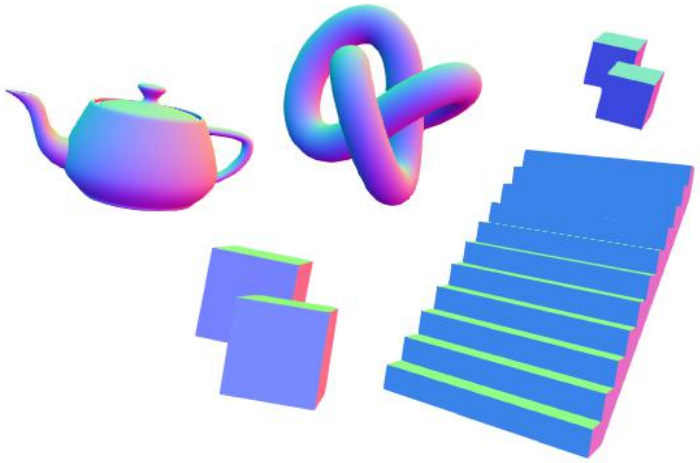

Fig. 20 - A cena da Fig. 18, com as cores dos pixels definidos com a cor da direção das normais em view space na posição de cada pixel.

Parte 3: Usando a *Roberts Cross* para detectar descontinuidades

Tendo agora as informações que precisávamos sobre a cena e seus objetos, podemos começar a calcular as descontinuidades. Para tal, nos baseamos em uma técnica de detecção de bordas em imagens, chamada "*Roberts Cross*" [27], que usaremos em conjunto com a textura de normais e de profundidade. O operador *Roberts Cross* usualmente recebe de entrada uma imagem preto e branca, processa os seus pixels e devolve uma outra imagem. Essa imagem devolvida tem as cores de seus pixels modificadas para uma escala de cinza, com as cores sendo mais brancas em regiões em que há uma

grande diferença de tonalidade. Normalmente uma diferença maior pode indicar uma descontinuidade. A Fig .21 ilustra o efeito aplicado pela *Roberts Cross*.

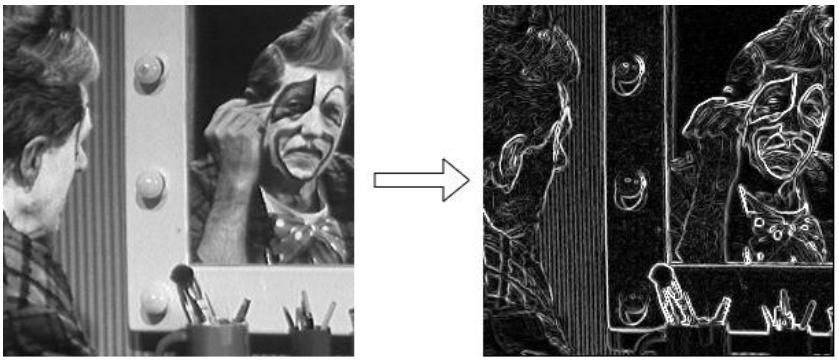

Fig. 21 - À esquerda temos a imagem original e à direita, temos a imagem após ser processada pela Roberts Cross [27]. O cálculo da *Roberts Cross* é feito da seguinte forma:

1. Para um dado pixel, formar dois pares de pixels diagonalmente adjacentes e tirar a diferença entre os valores dos pixels de cada par.
2. Para cada uma dessas diferenças, elevá-la ao quadrado e somá-las
3. Tirar a raiz quadrada da soma dos dois valores obtidos.

Este processo pode ser resumido na fórmula da Fig. 22.

Dado y, a cor de um dado pixel, i a posição no eixo x da imagem e j a posição no eixo y (a Fig. 23 ilustra os pixels participantes do cálculo):

$$z_{i,j} = \sqrt{(y_{i,j} - y_{i+1,j+1})^2 + (y_{i+1,j} - y_{i,j+1})^2}$$

Fig. 22 - Fórmula para calcular a Roberts Cross. Extraída da página sobre a Roberts Cross da Wikipedia [28].

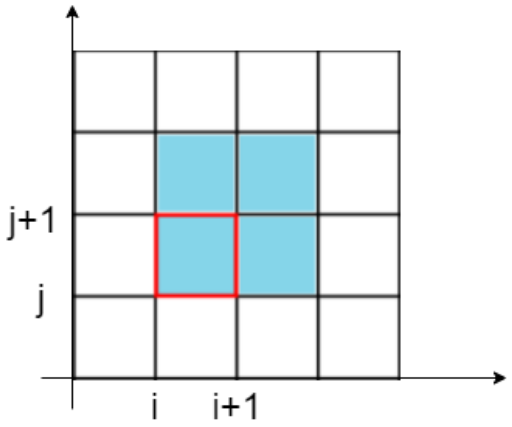

Fig. 23 - Representação do posicionamento dos pixels utilizados para calcular a Roberts Cross. O pixel contornado de vermelho é o pixel atualmente sendo analisado na imagem.

Para que possamos controlar a espessura do contorno, temos um parâmetro chamado *Scale* (escala). *Scale* altera quais são os pixels analisados pelo *Roberts Cross*, alterando a distância que os pixels analisados estão do pixel atual. Observe na Fig. 24 como a distância dos pixels não aumenta de uma forma constante, com pares de pixels sendo distanciados intercaladamente. Isso é porque dessa forma conseguimos garantir que a borda aumente um pixel de espessura por vez. Se não, para cada unidade que aumentássemos na escala, o contorno aumentaria a espessura em dobro.

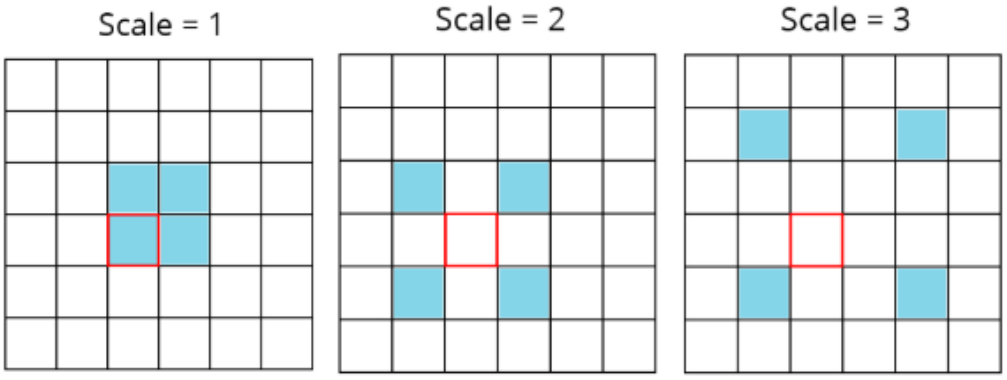

Fig. 24 - Posição dos pixels analisados, relativos ao pixel atual, para um Scale de valor 1 (esquerda), 2 (meio) e 3 (direita).

Parte 4: Detectar descontinuidades pela textura de profundidade e textura de normais

Para detectarmos as descontinuidades de uma cena, nós unimos a técnica da *Roberts Cross* com as informações da textura de profundidade e a de normais dos objetos na cena.

Descontinuidades na profundidade

Para detectarmos descontinuidades de profundidade utilizando a *Roberts Cross* é bastante simples. Como a textura de profundidade, por ter seus pixels com cores em uma escala de cinza, representada com um intervalo de 0 a 1 (preto ao branco), seus valores já são escalares. Portanto, podemos calcular a *Roberts Cross* como descrito na Parte 3 desta seção (4.3.1), utilizando somente um dos canais das cores dos pixels da textura (já que todos os canais são iguais). Com o resultado da *Roberts Cross* calculado, temos que aplicar um ajuste. Note como na Fig. 25 há descontinuidades mais acinzentadas e outras mais brancas. Essas que são mais acinzentadas são pontos de descontinuidade mais fracos, e que não necessariamente representam um ponto em que queremos desenhar um contorno. Para que só tenhamos descontinuidades mais fortes, criamos um limite mínimo para os valores de descontinuidade que

resultam da aplicação da *Roberts Cross*. Valores iguais ou abaixo do limite tornam-se 0 e acima do limite tornam-se 1. Chamaremos esse limite de “limite de profundidade”.

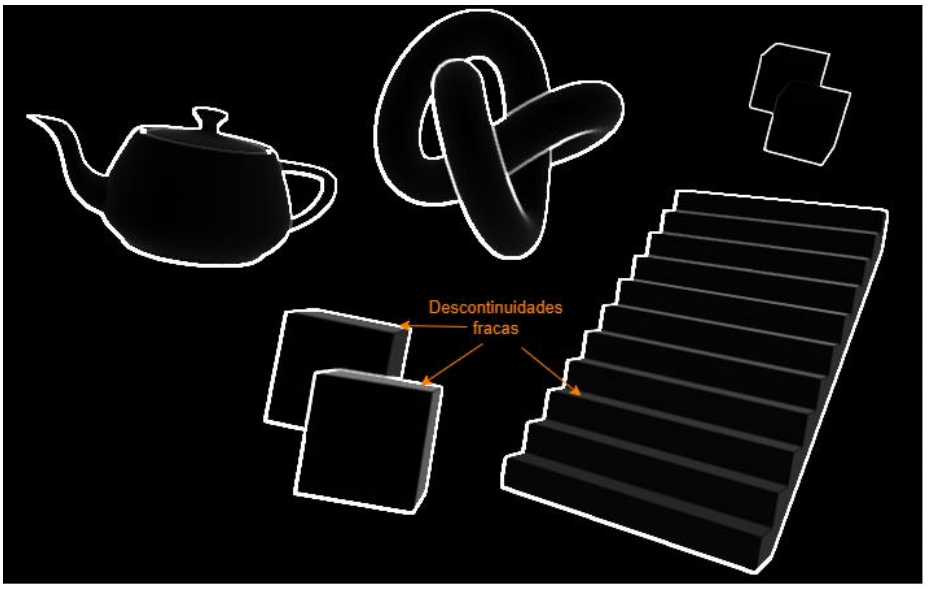

Fig. 25 - Descontinuidades pela textura de profundidade

Porém, há um problema que surge com esse ajuste: o limite que definimos é uma constante, enquanto o valor registrado no *depth buffer* é não-linear. Essa não-linearidade implica que os valores da textura de profundidade aumentam mais rápido quão mais perto os objetos estão da câmera. Por conta disso, objetos, que por exemplo, estejam separados um metro entre si, terão uma diferença de profundidade muito maior caso estejam bem próximos à câmera e bem menor caso estejam distantes dela. Observe na Fig. 26 como são detectadas descontinuidades entre o par 1, que está mais próximo da câmera, que não são detectadas entre o par 2.

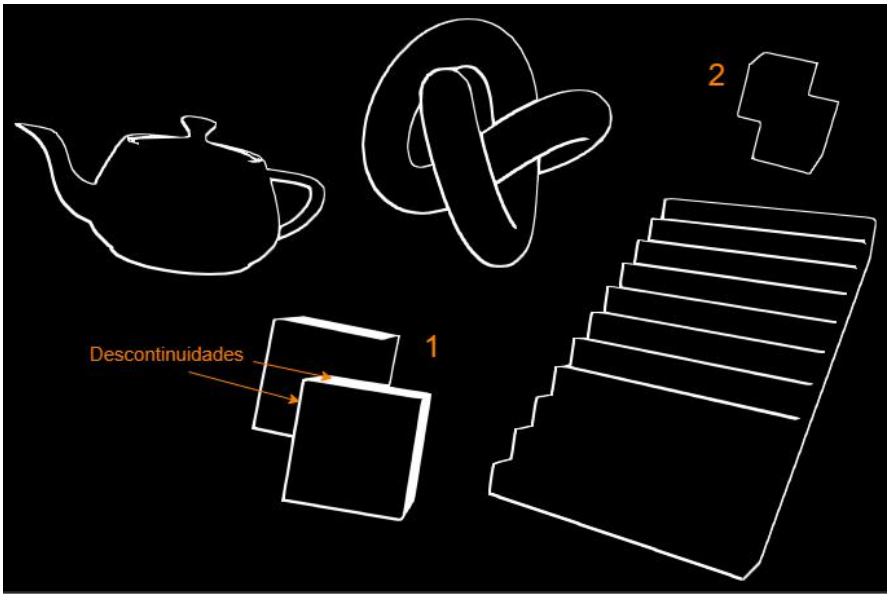

Fig. 26 - O par de cubos 1 se encontra mais próximo da câmera e o par 2 se encontra mais distante dela.

Para tratarmos esse problema basta multiplicar o limite de profundidade por qualquer um dos pixels selecionados para a checagem da *Roberts Cross*. Observe a Fig. 27 e note como agora todas as descontinuidades detectadas no par 1 são detectadas também pelo par 2. Foi necessário também aumentar o limite, pois os objetos que estavam próximos da câmera passaram a detectar descontinuidades onde não haviam.

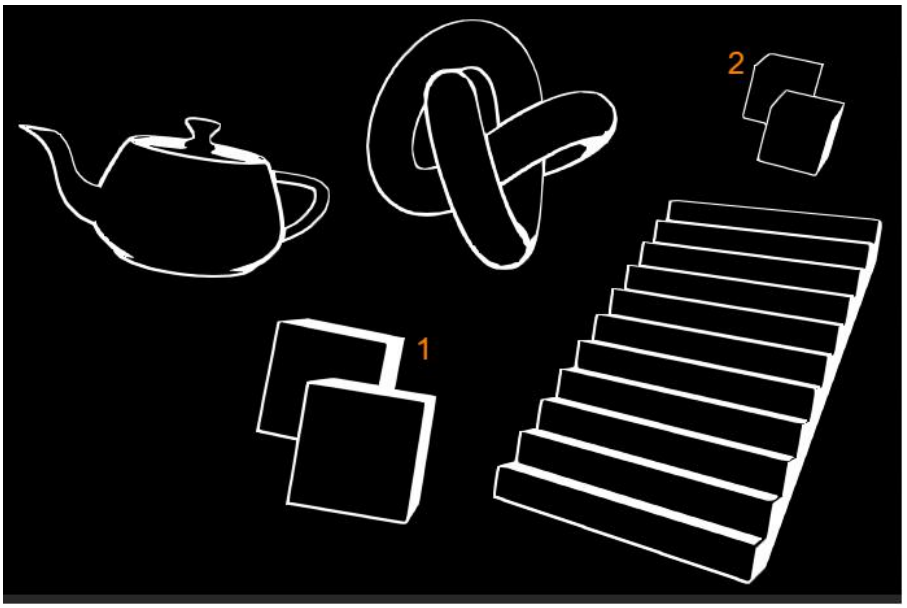

Fig. 27 - A cena da Fig. 26, após aplicar a correção para a não-linearidade da informação de profundidade.

Descontinuidades nas normais dos objetos

Para detectarmos descontinuidades entre normais de uma superfície, não é uma solução tão direta. Como precisamos que o valor seja um escalar, não podemos trabalhar com a forma original das cores dos pixel, porque eles são vetores do $\mathbb{R}3$. Para transformá-los em um escalar e, ao mesmo tempo, aplicarmos a potência ao quadrado e a raiz quadrada sobre o resultado final, como é feito no cálculo da *Roberts Cross*, podemos usar o produto interno dos vetores, que é calculado justamente utilizando essas operações. Após calcular as descontinuidades, também aplicamos um novo limite, como fizemos para as descontinuidades por profundidade. A Fig. 28 ilustra as descontinuidades detectadas pela textura de normais.

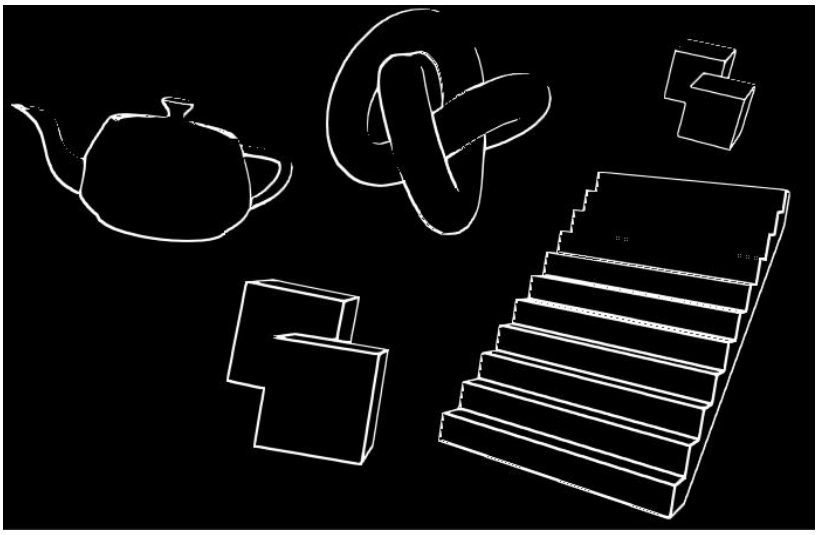

Fig. 28 - Descontinuidades pela textura das normais

Tendo calculado as descontinuidades pela profundidade e pelas normais dos objetos, agora precisamos unir os resultados. Isso porque, como podemos ver comparando as Fig .27 e Fig. 28 há algumas descontinuidades que cada método não consegue detectar individualmente. Por exemplo, na Fig. 27 podemos notar que no par de cubos 2, não são detectadas as descontinuidades em algumas arestas, que são detectadas na Fig. 28. Ao unir os dois resultados, conseguimos detectar o máximo de descontinuidades, como podemos ver na Fig.29.

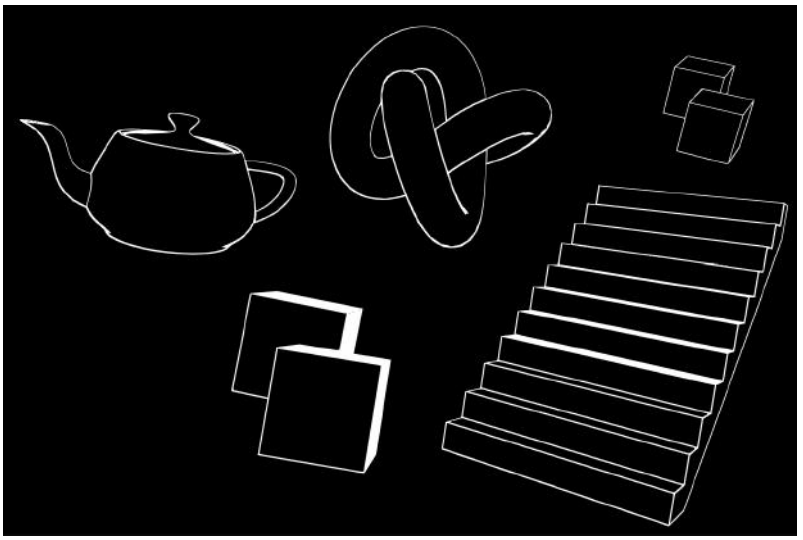

Fig. 29 - Resultado combinado da detecção por profundidade e normais.

Parte 5: Corrigindo artefatos visuais

Para finalizarmos a detecção de contornos, só precisamos corrigir um problema que causa artefatos visuais, como a Fig. 30 apresenta. Artefatos é como são comumente chamados erros visuais em computação gráfica. O problema ocorre quando temos partes de uma superfície que possuem uma

angulação muito grande em relação à câmera. Isto porque quão mais íngreme está a superfície, maior é a diferença de profundidade entre dois pixels adjacentes. Isto faz o *shader* detectar descontinuidades onde ele não deveria. Para resolvermos o problema, precisamos ajustar o limite de profundidade baseando-nos na normal da superfície relativa à câmera. Nosso objetivo é fazer com que o limite mude dinamicamente, sendo maior para superfícies mais íngremes em relação à câmera e menor para superfícies menos íngremes. Para fazermos isso, precisamos de dois dados: 1) a normal de cada superfície (o que já temos pela *view normals*); 2) *View direction*, isto é, a direção da câmera para a superfície. Como já temos as normais, só precisamos adquirir a view direction, o que é um processo um pouco complicado.

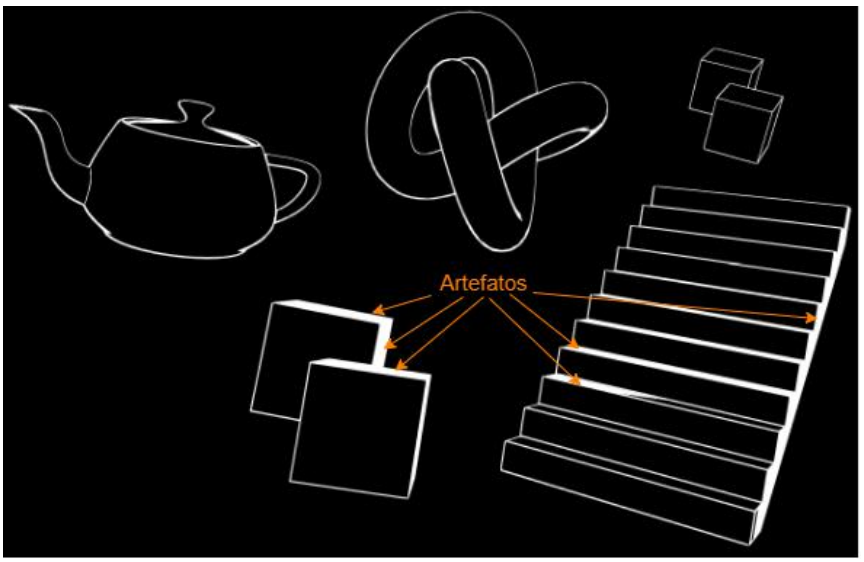

Fig. 30 - Artefatos visuais causados por uma inclinação grande das superfícies a partir da perspectiva da câmera.

Calculando a view direction

Como as nossas *view normals* estão em *view space*, precisamos que a view direction também esteja no *view space*. Felizmente, como esse *shader* é feito em *screen space* (ou seja, trabalha com os dados renderizados na tela), podemos calcular a *view direction* em *clip space*, a partir da posição do vértice. *Clip space* é o espaço utilizado para determinar quais vértices da cena estão visíveis pela câmera, verificando se suas coordenadas estão dentro de um certo intervalo. Se um vértice não estiver visível, ele é *clipped* (cortado), não sendo passado para a tela. Uma posição em *clip space* varia da coordenada (-1, 1), no topo-esquerda, até (1,1), no fundo-direita. Essas posições podem ser interpretadas como a *view direction* da câmera para cada pixel da tela, enquanto no *clip space*. Porém, como nossas normais estão em *view space*, precisamos

transformar essa posição do *clip space* para *view space*. Para tal, precisamos multiplicar a posição pelo inverso da matriz de transformação chamada de *projection matrix*. A *projection matrix* é usada para transformar uma posição do view space para o *clip space*. Por isso vamos usar a **matriz inversa** dela, para transformar do *clip space* para *view space*. Com isso, temos a nossa *view direction* em *view space*.

Agora que temos a view direction, podemos calcular o nosso fator que irá multiplicar o limite de profundidade para corrigir os artefatos. Para tal, calculamos o inverso do produto interno (1 - produto interno) entre a *view normal* e a *view direction*. Com isso, quanto mais íngreme está uma superfície em relação à câmera, maior será o multiplicador do limite de profundidade, aumentando a diferença de profundidade necessária para detectar uma descontinuidade.

Por fim, só resta juntar as cores originais da cena com a cor do contorno (que é uma configuração disponível para o usuário) para finalizar a composição. Para tal, utilizamos um *alpha blending* em modo normal. Isto é, utilizamos a cor do contorno ao invés da cor original da cena, ou realizamos uma mistura com a cor original caso a cor definida pelo usuário possua algum alpha que não seja 1. Com isso, temos o resultado final na Fig. 31.

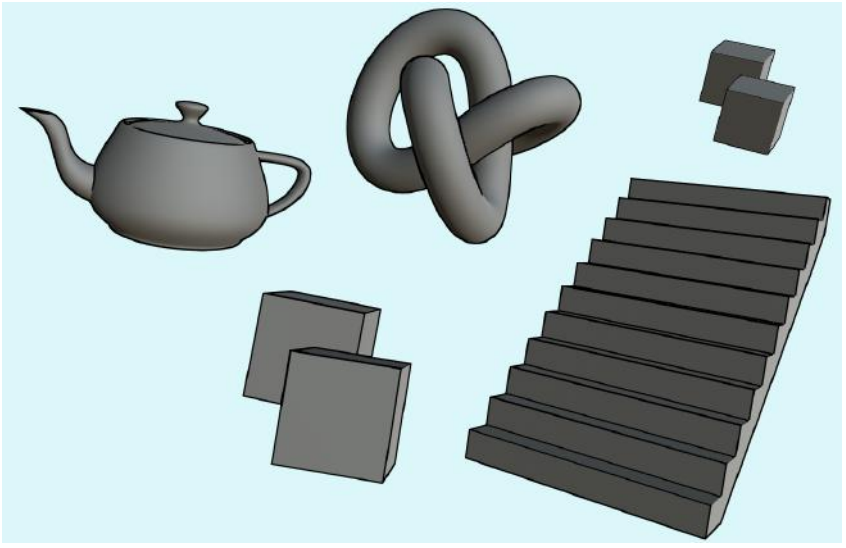

Fig. 31 - Resultado final do outline shader

Esta implementação foi extraída do tutorial do Erick Roystan, porém nós aprimoramos o resultado com uma alteração que visa resolver um problema que notamos: Objetos muito afastados da câmera acabavam por ser cobertos mais do que deviam pelo contorno, pois esse não estava mudando a escala conforme a distância da câmera, se mantendo de um tamanho fixo independentemente de quão afastados estavam dela. Nossa proposta é dividir o *Scale* original

(espessura do contorno) pela distância do objeto à câmera, obtida pegando a textura de profundidade, e transformando ela em linear usando a função do Unity *LinearEyeDepth [29]*. Uma observação é que essa função faz os valores de distância mais próximos da câmera começarem em 0 ao invés de 1, com o valor crescendo conforme os objetos ficam mais distantes. Isso inverte a lógica da textura de profundidade, de que objetos mais próximos da câmera têm um valor maior que os mais distantes. Dessa forma, objetos mais distantes terão a profundidade como valores maiores, fazendo seus contornos ficarem menores que os objetos mais próximos, pois os mais distantes estarão sendo divididos por valores maiores. Para evitar que isso afete a espessura dos contornos de objetos próximos da câmera, que não precisam desse ajuste, aplicamos um limite de distância. Se a câmera estiver mais próxima que a distância do limite, essa correção não é aplicada, usando a *Scale* original. Observe na Fig. 32 como o contorno do efeito original começa a quebrar e sobrepor muito os modelos. Com a minha proposta, o contorno se mantém em um tamanho consistente, independentemente da distância. O usuário pode manipular a distância em que essa mudança de escala começa a ter efeito, para evitar que o contorno fique pequeno quando a cämera estiver próxima.

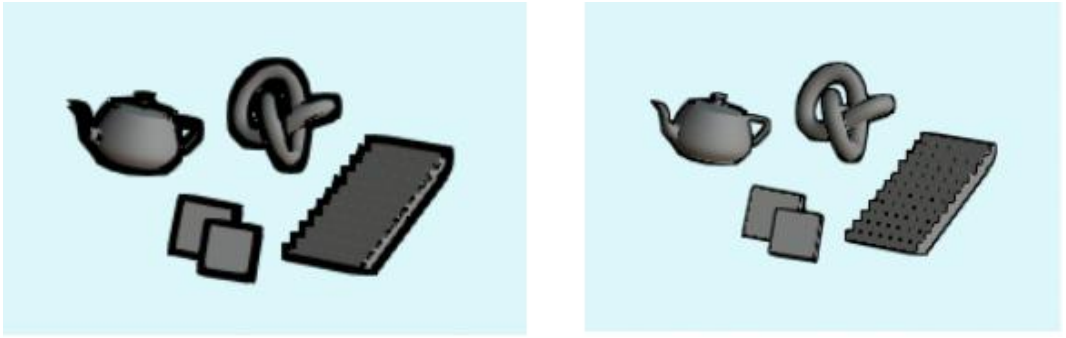

Fig. 32 - À esquerda é o efeito original. À direita é o efeito após a nossa correção proposta.

Configurações do *outline shader*

As configurações disponíveis para o usuário manipular são as ilustradas na Fig. 33. Estes são: **1) *Scale***, que determina o número de pixels do contorno; **2) *Depth Threshold***, que determina a limite de profundidade; **3) *Normal Threshold***, que determina o limite de das normais; **4) *Depth Normal Threshold***, que determina o mínimo da inclinação da superfície em relação à câmera para aplicar o ajuste para os artefatos causados pela alta inclinação de algumas superfícies; **5) *Depth Normal Threshold Scale***, que multiplica o limite do Depth Normal Threshold, dando um pouco mais de controle a essa configuração; **6) *Color***, que determina a cor do contorno; **7) *Distance To Use Normal Scale***, que

determina quão perto da câmera um objeto deve estar para não aplicar a correção de espessura do contorno para distâncias maiores da câmera.

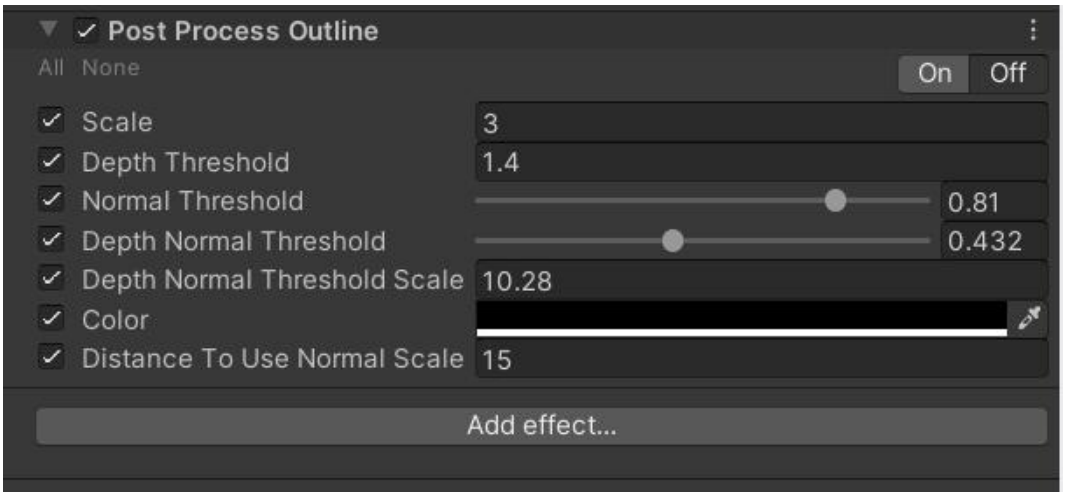

Fig. 33 - PostProcessProfile do outline shader

5. Resultados e validação

Nesta seção apresentaremos os resultados dos *shaders* desenvolvidos. Isto é, apresentaremos exemplos deles aplicados sobre algumas cenas e mostraremos como é o seu desempenho em uma cena bastante pesada em processamento. É melhor que esta cena seja pesada para que qualquer alteração na taxa de quadros por segundo (FPS) seja mais perceptível. A cena que utilizaremos para os testes de desempenho é a da Fig. 34, que já pode-se notar pela média de FPS (campo “Average” na imagem) já está bem baixo. Utilizaremos a cena da Fig. 35 de base para a apresentação visual dos efeitos. A chamaremos simplesmente de “cena de teste”. Todos os shaders desenvolvidos neste projeto podem ser adquiridos gratuitamente no seguinte repositório do GitHub: https://github.com/SirBernhart/Unity-post-processing-shaders.

Fig. 34 - Cena para testes de performance.

Fig. 35 - Cena de teste base para apresentação dos efeitos

5.1 Toon Shader

O toon shader funcionou muito bem, como ilustra a Fig. 36, que consegue replicar quase identicamente shaders já existentes. A Fig. 37 ilustra a cena de teste com um toon shading de 2 steps aplicado, que acreditamos que foi o melhor resultado para essa cena. Porém, temos a opção de ajustar as configurações do shader para quantos steps acharmos mais adequado. A Fig. 38 mostra a cena de teste com 2, 3, 4 e 5 steps, para ilustrar as outras possibilidades.

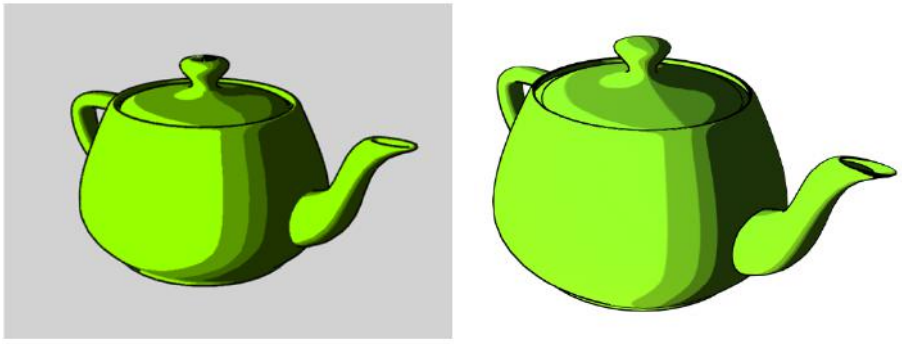

Fig. 36 - À esquerda, um exemplo de toon shader com outline sobre o clássico Utah Teapot. Imagem extraída da página "Cel Shading" da Wikipedia.[30] À direita, o mesmo modelo utilizando o Toon Shader e o Outline desenvolvidos neste projeto.

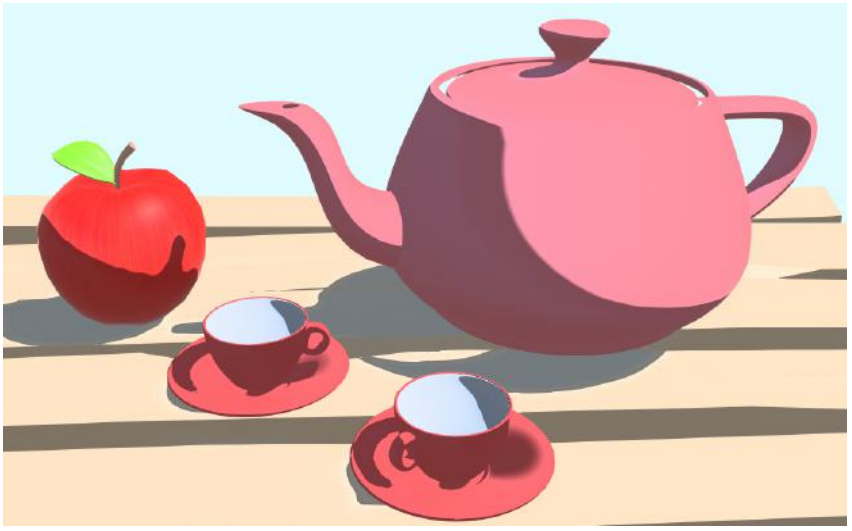

Fig. 37 - Cena da Fig. 34 com o toon shader de 2 steps aplicado

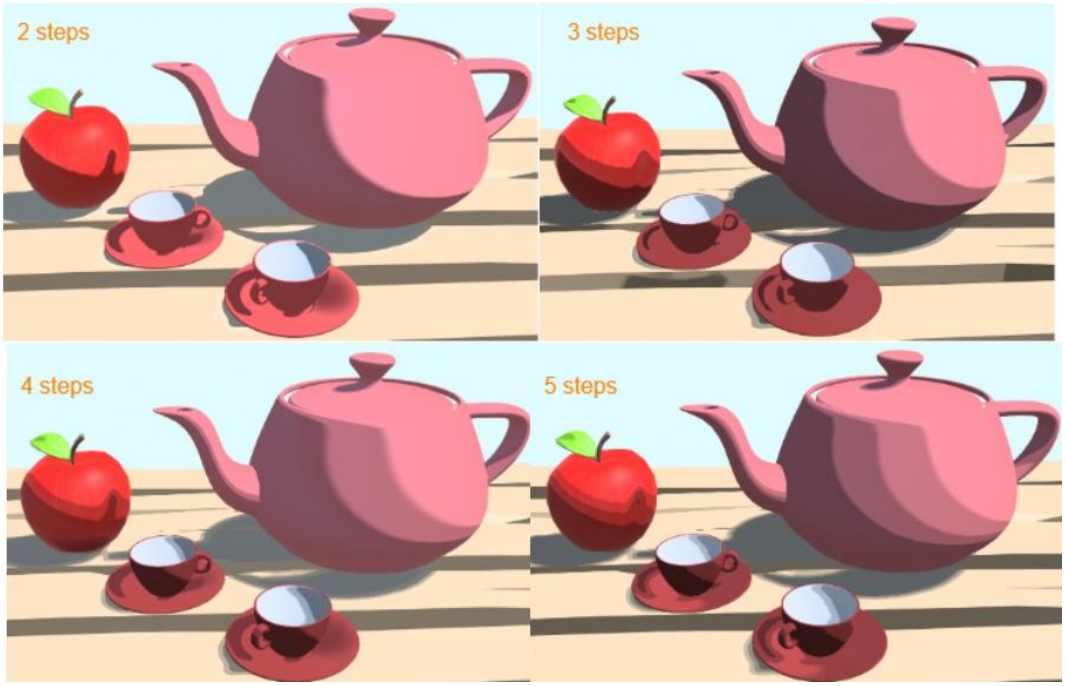

Fig. 38 - Cena da Fig. 34 com toon shader de 2, 3, 4 e 5 steps.

O Toon Shader funcionou muito bem para cenas que já têm uma estética mais estilizada, isto é, com texturas simples e com cores bem distintas, por ter uma variação de tons mais uniforme, muitas vezes só tendo transições entre tons por conta das sombras. A Fig. 39 ilustra uma cena já com objetos com uma estética estilizada, o que encaixou perfeitamente com o toon shader, com as setas apontando para as maiores diferenças na personagem. Observe como os tons se tornaram bem mais uniformes no cabelo e roupa da personagem, assim como o jarro na mesa e o chão.

Fig. 39 - Uma cena estilizada, mais voltada para um estilo cartum. A de cima é a cena com o toon shader desativado e a de baixo é com o Toon shader ativo, com 4 steps.

Porém, para cenas mais realistas, com texturas muito detalhadas e com cores muitas vezes menos saturadas, o resultado não é ideal, ficando bastante ruidoso e com pouco contraste, como apresentado na Fig. 40. Nesse tipo de cena, sequer é possível perceber as características faixas de tonalidade bem delimitadas, como vemos no bule da Fig. 36. Em conclusão, esse é um efeito que reduz os detalhes de uma cena em prol de atingir uma estética específica.

Portanto, ele não é muito indicado para cenas realistas detalhadas, podendo até ser considerado contra intuitivo sua aplicação nelas, pois se emprega bastante esforço para se obter o realismo. Por isso é mais comum aplicá-lo em cenas estilizadas, que já possuem pouco detalhe.

Fig. 40 - Uma cena realista, com texturas detalhadas e cores pouco saturadas. A de cima está com o toon shader desativado e a de baixo está com ele ativado, com 3 steps.

Teste de performance:

Como podemos notar na Fig. 41, o toon shader na cena de teste de performance não trouxe nenhum impacto de desempenho, mantendo-se na

média de 13 FPS. Esse resultado era esperado. Como trabalhamos com *shaders* de pós-processamento, o custo computacional independe da complexidade da cena. O mesmo pode ser dito para os nossos *shaders* de *outline* e *hatching*.

Fig. 41 - Cena de teste de performance com toon shader

5.2 Hatching Shader

O hatching shader teve ótimos resultados, dando uma ilusão convincente de que os objetos da cena realmente sejam desenhos feitos à mão, como a Fig. 42 ilustra. Em geral, o hatching shader funciona melhor para cenas bastante simples, com texturas pouco detalhadas, como o da Fig. 42 e 43. Além disso, diferentemente do toon shader, o hatching shader é um efeito para ser usado com parcimônia, pois adiciona muita informação para uma cena, podendo ser muito cansativo para os olhos.

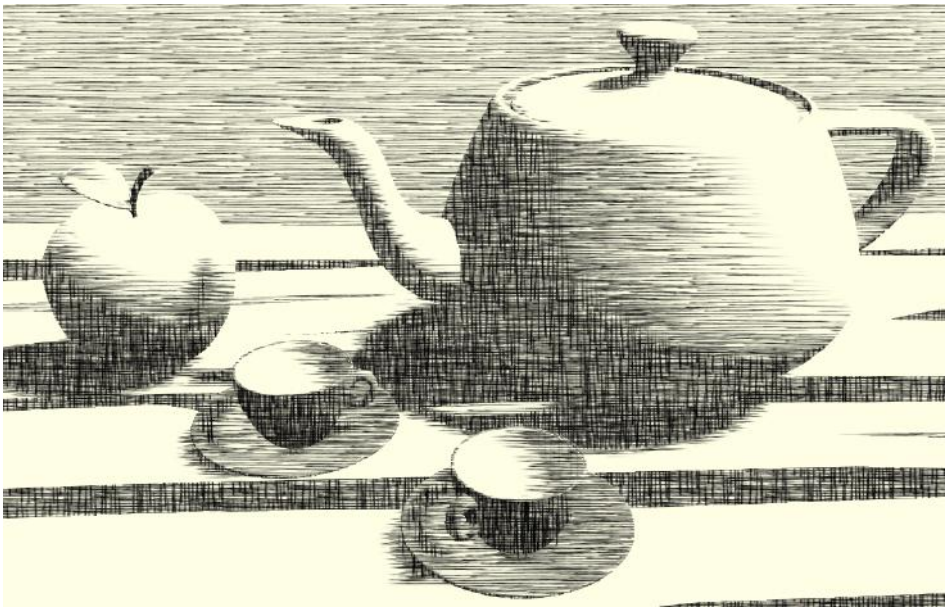

Fig. 42 - Cena de teste com o hatching shade aplicado sobre ela, com 7 steps, para ter uma transição o mais suave possível entre as texturas de hachura.

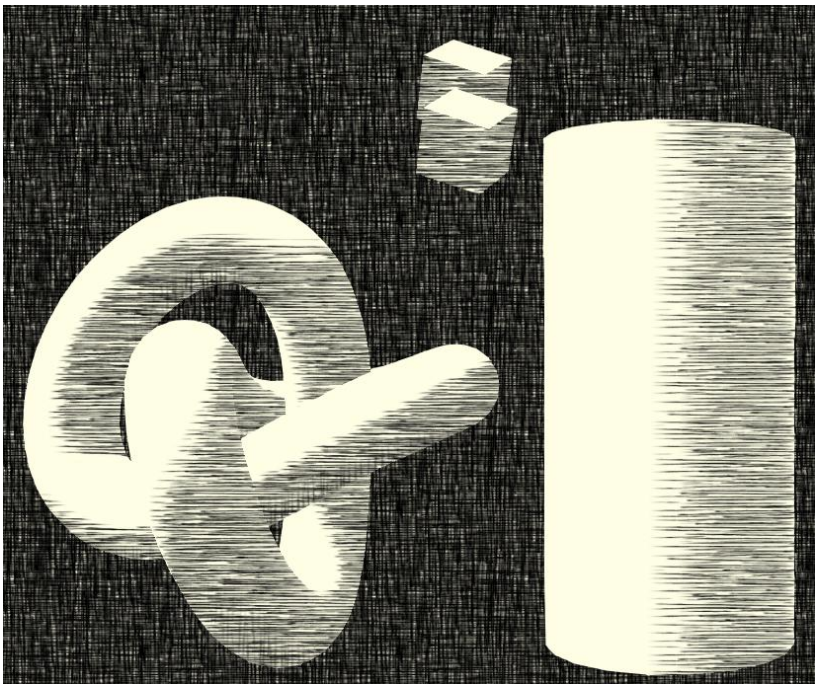

Fig. 43 - Cena com objetos simples, com apenas cores chapadas.

Um outro detalhe a se considerar é que o hatching, para cenas realistas com muitos detalhes, funciona melhor se os objetos forem observados de longe, com o intuito de mostrá-los de uma forma sem detalhes. Pois desta forma, não se espera observar os pequenos detalhes que são perdidos com a aplicação do hatching. Observe como na Fig. 44, que está observando a cena perto, é razoavelmente difícil distinguir os objetos. Porém na Fig. 45, que tem a câmera mais distante, fica mais fácil perceber o formato geral dos objetos da cena. Caso o usuário queira, também há a opção de utilizar as cores originais da cena no hatching shader, que se obterá um resultado como o da Fig. 46.

Fig. 44 - Cena realista, com texturas detalhadas e tons próximos, tendo a câmera bem próxima aos objetos da cena, com o hatching shader aplicado.

Fig. 45 - Cena realista, com a câmera distante dos objetos da cena.

Fig. 46 - Hatching shader sobre a cena de teste, utilizando as cores originais da cena.

Teste de performance:

Como podemos notar na Fig. 47, o hatching shader na cena de teste de performance não trouxe nenhum impacto de desempenho, mantendo-se na média de 13 FPS.

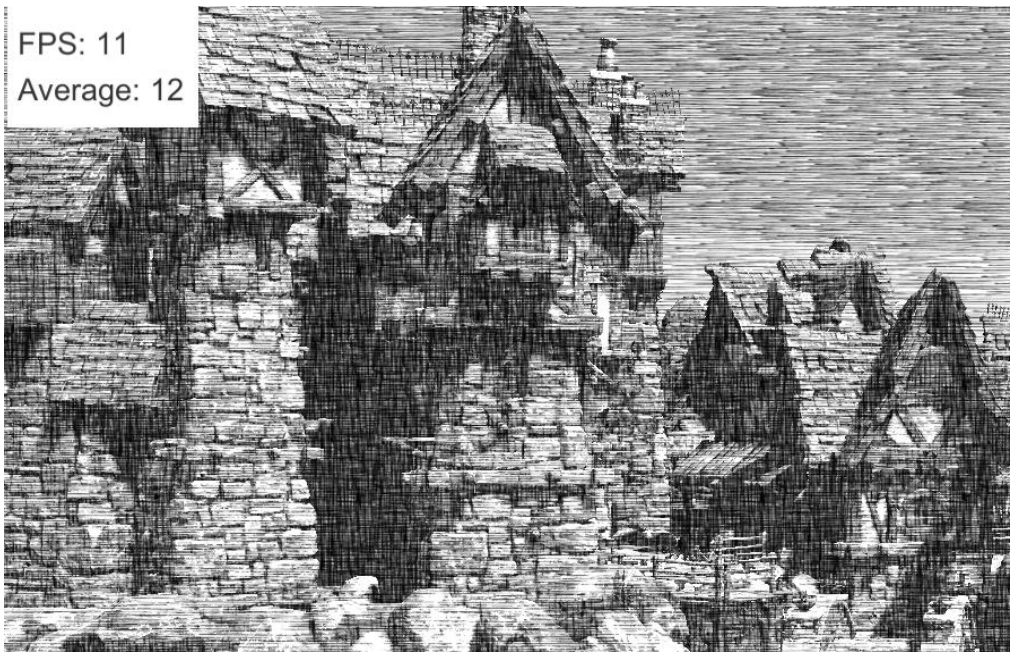

Fig. 47 - Cena de teste de performance com hatching shader

5.3 Outline Shader

O outline shader teve resultados bastante satisfatórios, como a Fig. 48 mostra. Ele tem uma afinidade natural com cenas mais estilizadas, completando um visual bastante cartunesco ao ser aplicado nelas, como ilustrado na Fig. 49.

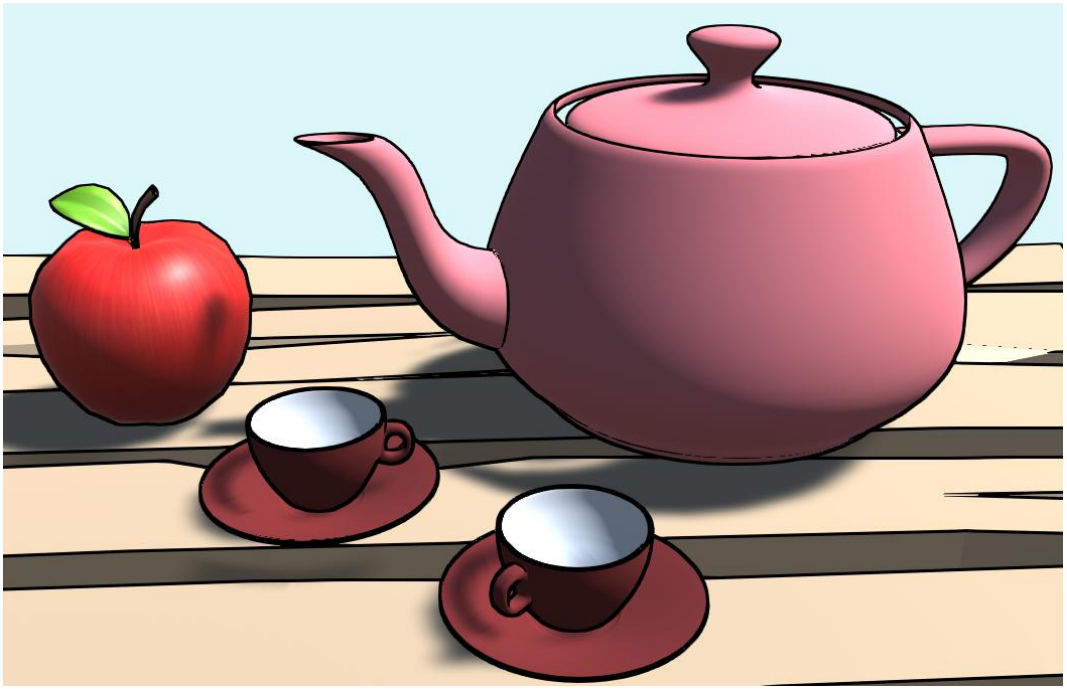

Fig. 48 - Cena de teste com o outline shader.

Fig. 49 - Cena estilizada. À esquerda, com o outline shader desativado. À direita, com o outline shader ativado.

Apesar de ele se encaixar muito bem com cenas mais estilizadas, ele também faz um bom par para uma cena mais realista, dando um detalhe sutil para alguns lugares, que evoca um estilo que remete ao de histórias em quadrinhos com o traço mais realista. Observe o contorno nas pedras e a cerca na imagem de baixo da Fig. 50.

Fig. 50 - Cena realista. A imagem de cima está com o outline shader desativado. A imagem de baixo está com o outline shader ativo.

Teste de performance:

Como podemos notar na Fig. 51, o toon shader na cena de teste de performance não trouxe nenhum impacto de desempenho, mantendo-se na média de 13 FPS.

Fig. 51 - Cena de teste de performance com outline shader

5.4 Junções dos efeitos

Agora que vimos os efeitos individualmente, vamos fazer algumas combinações com eles, para obtermos alguns resultados interessantes.

Outline + Toon

Esta é a combinação clássica, muitas vezes considerada somente como toon shading completo, pois o outline completa a estética de desenhos animados, histórias em quadrinhos e cartuns que o toon shading sozinho tenta emular. A Fig. 52 ilustra a combinação dos dois efeitos.

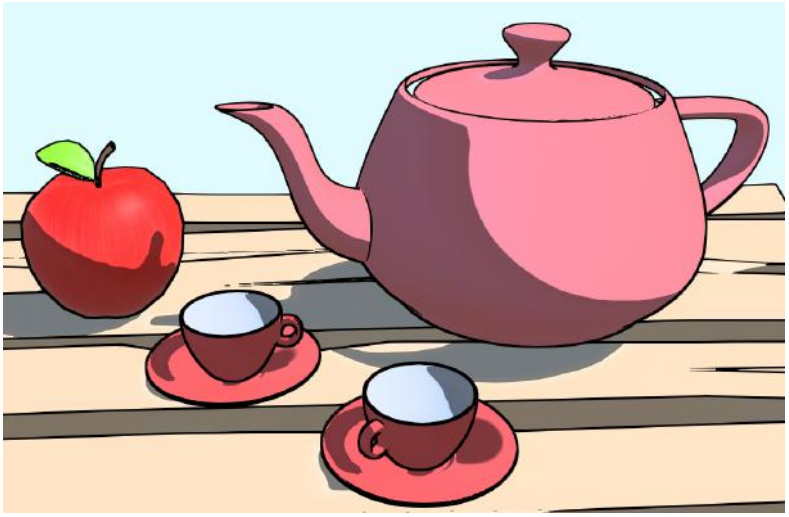

Fig. 52 - Cena de teste com toon shader e outline

Outline + hatching

Um problema do hatching shader por si só é que ele muitas vezes ele não deixa as bordas dos objetos muito bem definidas. Isso pode acontecer por conta da hachura aplicada a um objeto ser a mesma que a de um objeto atrás ou do fundo da cena, sendo difícil de distinguir onde um começa e outro acaba. O outline resolve esse problema, pois seu propósito é justamente de delimitar as bordas dos objetos da cena, dando a nitidez que o hatching precisava. Compare a Fig. 53 com a Fig. 42 para perceber quão mais nítida fica o hatching acompanhado do outline.

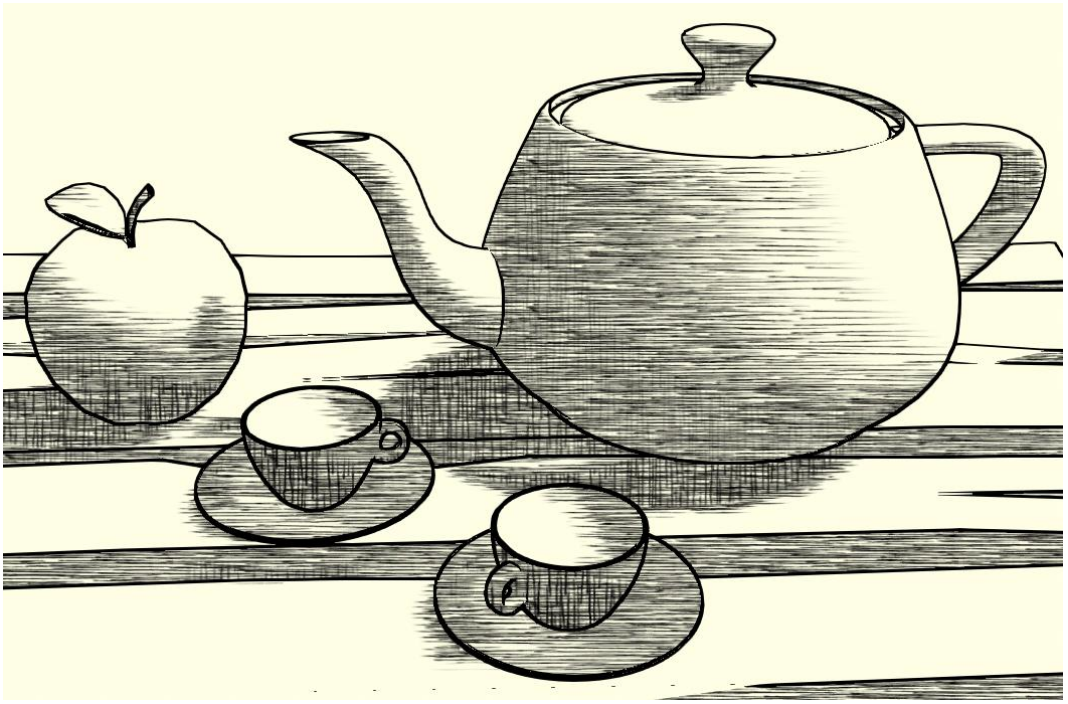

Fig. 53 - Cena de teste com hatching e outline aplicados.

Outline + Hatching + Toon com 2 steps

Esta combinação traz um efeito que remete a um estilo de hachura mais estilizada, similar ao que se encontra em alguns mangás japoneses, como o da Fig. 54. O resultado se encontra na Fig. 55.

Fig. 54 - Exemplo de hachura utilizada em mangás japoneses. Extraído do fórum "Skull Knight.net" [31].

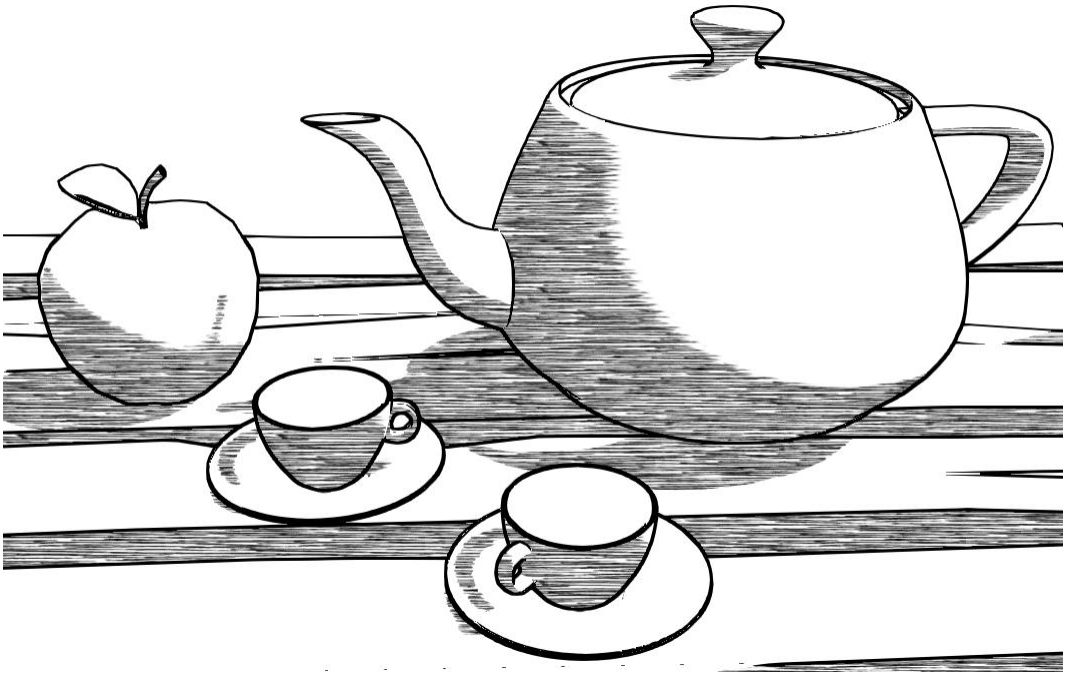

Fig. 55 - Cena de teste com toon shading com 2 steps, hatching e outline aplicados.

6. Conclusão

Os resultados obtidos dos *shaders* atenderam às nossas expectativas, tanto no efeito produzido em sí quanto no baixo impacto para a performance da renderização da cena. A implementação do *toon shader* pela interpretação das cores da cena no modelo HSV funcionou muito bem, sendo facilmente configurável para adaptar-se melhor a qualquer cena. O *hatching* também teve um ótimo resultado, sua implementação sendo um caso interessante, em que pudemos reutilizar grande parte do código feito para o *toon shader*. O *outline shader* também teve um bom resultado, sendo capaz de detectar bem descontinuidades, ainda com a capacidade de ajustar a espessura de seu contorno com a nossa contribuição para corrigi-los quando vistos de longe.

Para trabalhos futuros em cima dos shaders desenvolvidos neste projeto, vemos como uma possibilidade estudar a viabilidade de fazer com que as hachuras do *hatching shader* acompanhem a superfície dos objetos, como na Fig. 9. Outra possibilidade é a de ter um contorno de espessura irregular para o *outline shader*, como na Fig. 11, ou até aplicando algum tipo de textura ao invés de uma simples cor. O código compartilhado do *hatching* e do *toon shader* também abre possibilidades de criar efeitos com um conceito similar de identificar os tons da cena e modificá-los.

7. Referências

[1] UNITY. **A principal plataforma de criação de conteúdo em tempo real do mundo**. https://unity.com/pt. Acesso em: 20/06/2022.

[2] UNITY. **Post-processing and full-screen effects.** https://docs.unity3d.com/Manual/PostProcessingOverview.html. Acesso em: 20/06/2022.

[3] WIKIPEDIA. **Motor gráfico**. https://pt.wikipedia.org/wiki/Motor_gr%C3%A1fico. Acesso em: 14/06/2022.

[4] UNITY. **Scenes**. https://docs.unity3d.com/Manual/CreatingScenes.html. Acesso em: 15/06/2022.

[5] UNITY. **GameObject**. https://docs.unity3d.com/ScriptReference/GameObject.html. Acesso em: 15/06/2022.

[6] UNITY. **Component**. https://docs.unity3d.com/ScriptReference/Component.html. Acesso em: 15/06/2022.

[7] UNITY. **The Inspector window**. https://docs.unity3d.com/Manual/UsingTheInspector.html. Acesso em: 15/06/2022.

[8] UNITY. **Post Processing Stack v2 overview**. https://docs.unity3d.com/Packages/com.unity.postprocessing@3.2/manual/index.html. Acesso em: 20/06/2022.

[9] WIKIPEDIA. **Anti-aliasing**. https://pt.wikipedia.org/wiki/Anti-aliasing. Acessado em: 15/06/2022.

[10] MICROSOFT. **HLSL**. https://docs.microsoft.com/pt-br/windows/win32/direct3dhlsl/dx-graphics-hlsl. Acesso em: 15/06/2022.

[11] UNITY. **Writing Custom Effects**. https://docs.unity3d.com/Packages/com.unity.postprocessing@3.0/manual/Writing-Custom-Effects.html. Acesso em: 15/06/2022.

[12] Akenine-Möller, T., Haines, E., Hoffman, N., Pesce, A., Iwanicki, M., Hillaire, S. **Real-Time Rendering**. 4ª edição. A K Peters/CRC Press, p. 652-653, 2018.

[13] Rideout, P. **Antialiased Cel Shading**. https://prideout.net/blog/old/blog/index.html@p=22.html. Acesso em: 19/06/2022.

[14] LET'S PLAY ARCHIVE. **Ni no Kuni: Part 2**. https://lparchive.org/Ni-no-Kuni/Update%2002/. Acesso em: 19/06/2022.

[15] ElAnalystaDeBits. **Zelda Breath of the Wild | Cel Shading Glitch | Punto del juego sin Cel Shading**. https://www.youtube.com/watch?v=jq4RptJkeH4&ab_channel=ElAnalistaDeBits. Acesso em: 19/06/2022.

[16] WIKIPEDIA. **Hachura**. https://pt.wikipedia.org/wiki/Hachura. Acesso em: 12/06/2022.

[17] Lake, A., Marshall, C. Harris, M†. Blackstein, M. **Stylized Rendering Techniques For Scalable Real-Time 3D Animation**. http://www.markmark.net/npar/npar2000_lake_et_al.pdf. Acesso em: 19/06/2022.

[18] Praun, E., Hoppe, H., Webb, M., Finkelstein, A. **Real-Time Hatching**. https://hhoppe.com/hatching.pdf. Acesso em: 19/06/2022.

[19] Ferreira, P. **Bomb Rush Cyberfunk é um sucessor espiritual de Jet Set Radio**. https://pt.ign.com/games/89201/news/bomb-rush-cyberfunk-e-um-sucessor-espiritual-de-jet-set-radio. Acesso em: 19/06/2022.

[20] Foca, R. **Análise: Mesmo 12 anos depois, Okami HD é um destaque que merece atenção**. https://jogazera.com.br/analise-okami-hd/. Acesso em: 19/06/2022.

[21] WIKIPEDIA. **HSL and HSV**. https://en.wikipedia.org/wiki/HSL_and_HSV. Acesso em: 16/06/2022.

[22] UNITY TECHNOLOGIES. **FPSSample/Color.hlsl**. https://github.com/Unity-Technologies/FPSSample/blob/master/Packages/com.unity.postprocessing/PostProcessing/Shaders/Colors.hlsl. Acesso em: 20/06/2022.

[23] MICROSOFT. **Smoothstep**. https://docs.microsoft.com/pt-br/windows/win32/direct3dhlsl/dx-graphics-hlsl-smoothstep. Acesso em: 16/06/2022.

[24] candycat1992. **NPR_Lab**. https://github.com/candycat1992/NPR_Lab/tree/master/Assets/Textures/Hatch. Acesso em: 19/06/2022.

[25] ROYSTAN, E. **Outline Shader**. https://roystan.net/articles/outline-shader.html. Acesso em: 17/06/2022.

[26] VRIES, J. **Learn OpenGL: Coordinate Systems**. https://learnopengl.com/Getting-started/Coordinate-Systems. Acesso em: 18/06/2022.

[27] HIPR2. **Roberts Cross Edge Detector**. https://homepages.inf.ed.ac.uk/rbf/HIPR2/roberts.htm. Acesso em: 17/06/2022.

[28] WIKIPEDIA. **Roberts cross**. https://en.wikipedia.org/wiki/Roberts_cross. Acesso em: 17/06/2022.

[29] Golus, B. **Unity Forums: DecodeDepthNormal/Linear01Depth/LinearEyeDepth explanations**. https://forum.unity.com/threads/decodedepthnormal-linear01depth-lineareyedepth-explanations.608452/#post-4070806. Acesso em 20/06/2022.

[30] WIKIPEDIA. **Cel Shading**. https://en.wikipedia.org/wiki/Cel_shading. Acesso em: 19/06/2022.

[31] Skull Knight.net. **New Digital Art Techniques in Berserk**. https://www.skullknight.net/forum/index.php?threads/new-digital-art-techniques-in-berserk.15089/. Acesso em: 20/06/2022