The House of Horrors

TÉCNICAS AVANÇADAS DE PROGRAMAÇÃO 3D

ADEMAR VALENTE
HENRIQUE AZEVEDO
JOSÉ LOURENÇO

LICENCIATURA EM ENGENHARIA DE JOGOS DIGITAIS

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

IPCA INSTITUTO POLITÉCNICO DO CÁVADO E DO AVE

THE HOUSE OF HORRORS

ÍNDICE

1. INTRODUÇÃO

O presente relatório foi desenvolvido para descrever a realização da componente prática da disciplina de Técnicas Avançadas de Programação 3D, do 3º ano da Licenciatura em Engenharia em Desenvolvimento de Jogos Digitais. Este trabalho é o resultado da colaboração entre os alunos Ademar Valente, Henrique Azevedo e José Lourenço, representa um dos elementos de avaliação prática da unidade curricular, e reflete o esforço conjunto e a aprendizagem adquirida no âmbito do curso.

O projeto em questão foi meticulosamente executado na plataforma Unity 3D, um ambiente robusto e versátil para o desenvolvimento de jogos. A essência deste trabalho prático reside na criação de uma série de materiais e shaders inovadores, concebidos especificamente para serem utilizados no Unity 3D, com o intuito de enriquecer a qualidade visual e a funcionalidade dos jogos desenvolvidos.

A equipa não só procura aplicar os conhecimentos teóricos adquiridos durante as aulas, mas também ir além disso. Através de uma pesquisa aprofundada, exploramos novas soluções e técnicas avançadas na criação de shaders, algumas das quais foram inspiradas e sugeridas pelo docente da disciplina. Este esforço conjunto resultou num trabalho que não só demonstra a aplicabilidade prática dos conceitos estudados, mas também reflete a capacidade de inovação e adaptação dos alunos ao enfrentarem desafios reais do mundo da programação de jogos.

1.1. O CONCEITO: UM CONJUNTO DE HORRORES

No início do nosso projeto, a docência propôs um desafio estimulante: escolher um tema que direcionasse o nosso trabalho prático. Após uma reflexão conjunta, decidimos que o nosso conjunto de efeitos visuais seria idealmente situado num ambiente de terror. Foi essa atmosfera sombria e arrepiante que escolhemos explorar.

Durante o processo criativo, dedicamo-nos a conceber ideias de efeitos que complementassem esse tema. O primeiro efeito que concebemos foi particularmente impactante para a escolha do título deste trabalho: um shader que simula uma casa a arder e a desintegrar-se como se fosse feita de papel. Este efeito não só captura a essência do terror, mas também evoca uma sensação de destruição efémera e assustadora.

THE HOUSE OF HORRORS

Foi este conceito inovador que inspirou o nome do nosso pacote de efeitos: The House of Horrors. Este nome reflete não só a natureza do nosso trabalho, mas também o ambiente imersivo que pretendemos criar com os nossos shaders, proporcionando aos desenvolvedores de jogos as ferramentas necessárias para envolver os jogadores numa experiência verdadeiramente horripilante.

A escolha do tema de terror não foi aleatória; reflete uma decisão consciente de mergulhar nas profundezas de um género que habitualmente cria uma experiência imersiva para o jogador. O terror, com suas nuances e complexidades, oferece um terreno fértil para a inovação em design de jogos e efeitos visuais.

1.2. OS SHADERS

Os shaders são o ponto central deste trabalho, bem como elementos fundamentais na criação de jogos com o Unity 3D; atuando como poderosas ferramentas que podem definir a aparência visual dos objetos num ambiente de jogo.

No contexto de um jogo, os shaders podem ser responsáveis por renderizar texturas, simular/modificar físicas de luz, criar efeitos visuais, ou otimizar o próprio desempenho de um jogo. Em suma, os shaders são cruciais para adicionar vida e movimento aos jogos dos dias de hoje.

Em The House of Horrors, os shaders são utilizados para evocar uma atmosfera de terror, com ilusões óticas que distorcem a realidade, ou efeitos que incutem atmosferas sombrias, bizarras e alternativas. Em várias vertentes dos jogos este tipo de efeitos são a espinha dorsal da experiência visual do jogo, transformando ambientes normais em cenários de pesadelo que desafiam a perceção e aumentam a tensão.

Através do Unity 3D, temos a liberdade de manipular esses shaders para criar experiências únicas, garantindo que cada elemento visual não só pareça impressionante, mas também contribua para a narrativa e jogabilidade do jogo. Os shaders são, portanto, uma expressão da arte técnica, permitindo que os desenvolvedores traduzam sua visão criativa em realidade virtual palpável.

1.3. ESTRUTURA DO RELATÓRIO

Este relatório foi estruturado para fornecer uma visão abrangente e detalhada do nosso trabalho prático na disciplina. Apresentamos assim a estrutura do relatório, que se desdobra em várias seções críticas:

- **Introdução:** Esta seção inicial estabelece o contexto do relatório, delineando os objetivos do trabalho prático e a relevância do tema escolhido. Ela serve como um prelúdio para as seções subsequentes, preparando o conteúdo técnico e criativo que se segue;
- **Descrição dos Shaders:** O cerne do relatório é dedicado aos shaders desenvolvidos. Subdividimos esta secção em **contextualização teórica e processo evolutivo** (onde se exploram os fundamentos teóricos dos shaders, descrevendo e discutindo sua importância, bem como o nosso próprio trabalho evoluiu a partir dele), **descrição técnica do código** (explicando estrutura e a lógica subjacente), funcionalidades de aplicabilidade adjacentes (scripts adicionais que suportam a aplicabilidade dos shaders no Unity 3D), e **ideias para aperfeiçoamento** (com reflexões sobre possíveis melhorias e expansões futuras de cada shader);
- **Conclusões:** Capítulo de reflexão sobre os objetivos alcançados, onde fazemos uma apreciação global do trabalho executado e identificamos processos e áreas que podem ser aprimorados em esforços futuros.
- **Bibliografia:** Com todas as fontes consultadas durante a pesquisa e desenvolvimento do trabalho, reconhecendo as contribuições teóricas e práticas que nos guiaram.

Cada seção foi elaborada com o intuito de não apenas documentar o nosso trabalho, mas também de servir como um testemunho na progressão ao nível de conhecimento teórico na disciplina; e na demonstração da sua aplicabilidade funcional.

2. OS SHADERS

2.1. SURFACE SHADER - DISSOLVE

Figura 1 - Dissolve Shader - Logo.

No coração do projeto "The House of Horrors", encontra-se uma casa abandonada, cujas paredes contam histórias de tempos esquecidos e segredos ocultos. Esta casa, outrora um lar cheio de vida, agora serve como o cenário principal para uma experiência visual arrepiante. Graças a um conjunto especial de efeitos visuais, a casa parece consumir-se em chamas, com as suas estruturas a desintegrar-se diante dos nossos olhos, evocando a sensação de um pesadelo que ganha vida.

O primeiro efeito, aplicado à própria estrutura da casa, faz com que pareça estar a ser lentamente devorada pelo fogo, desaparecendo completamente. Este tornou-se o ponto de partida para toda a criação artística subsequente. Como complemento o segundo efeito é reservado para os vidros, que parecem derreter e distorcer-se. Este efeito complementa a visão da casa a arder, adicionando uma camada de realismo e profundidade à cena.

É importante notar que, embora este seja o único shader de superfície utilizado no projeto, foi uma escolha deliberada e em conformidade com as diretrizes estabelecidas pelo docente. Este shader não só foi o primeiro a ser desenvolvido, mas também se tornou um elemento indispensável, sem o qual o trabalho não teria o mesmo impacto visual. A sua inclusão não foi apenas uma necessidade técnica, mas também uma decisão que define o tom de "The House of Horrors".

2.1.1. CONTEXTUALIZAÇÃO TEÓRICA

Para o desenvolvimento destes shaders, foram utilizados princípios teóricos fundamentais de duas fontes educativas que serviram como guia durante o processo de desenvolvimento.

A primeira fonte, um tutorial de **Ronja**[1], explora a técnica de dissolução de texturas, onde se utiliza uma textura adicional para controlar o padrão de dissolução de um objeto. Este método começa com a adição de uma nova textura ao shader, que é usada para determinar onde e como o objeto irá desaparecer. A técnica emprega a função clip para descartar seletivamente partes do objeto com base no valor de dissolução, criando assim o efeito visual de um objeto que se dissolve de forma controlada.

A segunda fonte, um artigo de **Kyle Halladay**[2], revisita o efeito de shader de dissolução, com foco em elementos 3D. Este tutorial aborda a dissolução de um objeto com base numa textura de controle, utilizando uma textura de ruído suavizado para obter um efeito de dissolução com um bom contraste. O tutorial também detalha como adicionar cor aos limites do efeito de dissolução, enriquecendo o resultado visual e evitando que pareça apenas um efeito de shader genérico.

Ambas as fontes contribuem para a compreensão de como criar um efeito de dissolução convincente e visualmente atraente, que é essencial para a atmosfera de mistério e suspense do projeto.

Os shaders “Custom/DissolveWindow” e “Custom/DissolveHouse” compartilham semelhanças com os conceitos encontrados nos dois sites mencionados, mas também apresentam diferenças. Como vamos abaixo analisar de forma detalhada, utilizam uma textura de queima (_BurnMap) para controlar o efeito de dissolução, o que é uma técnica comum discutida nos tutoriais. Para além disso também empregam mapas de normais (_BumpMap) e mapas de rugosidade (_SpecGlossMap) para adicionar detalhes e realismo ao efeito de queima.

No entanto, o shader “Custom/DissolveWindow” usa a propriedade RenderType = "Fade" para criar transparência, enquanto o shader “Custom/DissolveHouse” usa RenderType="Opaque" para um efeito sólido. Além disso, o shader “Custom/DissolveHouse” inclui um efeito de piscar (blink) que não é abordado diretamente nos tutoriais, mas que adiciona uma camada adicional de dinamismo ao efeito de queima.

Desta forma, pudemos compreender e melhorar a documentação que foi explorada para a realização deste surface shader; e no final entregar o efeito pretendido.

2.1.2. DESCRIÇÃO TÉCNICA

Passada a fase de execução e análise teórica, passemos à dissecção do shader propriamente dito, bem como as suas funcionalidades, tentando sempre que possível associar visualmente o que cada parte faz por forma a retirar a máxima compreensão deste capítulo. Neste shader específico, para além da descrição seccional de cada um dos dois shaders utilizados (DissolveHouse e DissolveWindow), procuraremos no final descrever as diferenças e o porquê de utilizarmos os dois na casa.

- **"DissolveHouse"**

Properties: Esta secção define as propriedades que podem ser ajustadas no editor da Unity. Por exemplo, **_MainTex** é a textura principal, **_Glossiness** controla a escala do efeito queimado, **_SpecGlossMap** é a textura de rugosidade usada para o efeito queimado, **_BumpMap** é o mapa normal, e **_BurnMap** é a textura que define o modo de queimar.

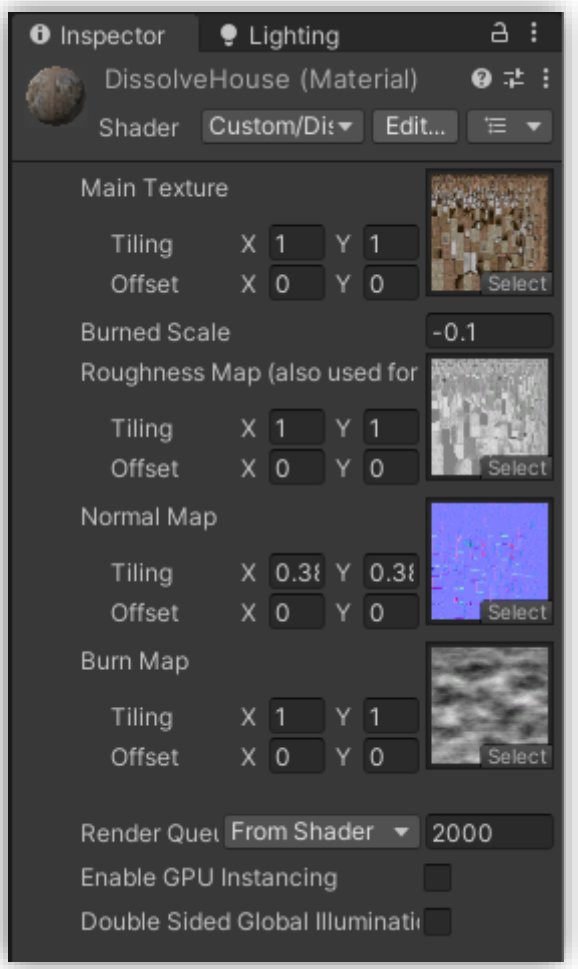


Figura 2 - DissolveHouse Inspector.

SubShader: Contém a lógica de renderização do shader. Define como a superfície do objeto deve aparecer. Inclui tags (de renderização definida como opaca), nível de detalhe (LOD – Level Of Details, que optimiza o desempenho), e o programa de shader (CGPROGRAM - HLSL).

Pragmas: Diretivas que controlam o processo de compilação do shader. Aqui, #pragma surface define a função surf como a função de superfície principal, e #pragma target especifica a versão do shader model a ser usada.

Sampler2D: Variáveis que armazenam as texturas a serem usadas pelo shader.

Input: Uma estrutura que contém as variáveis de entrada para a função de superfície. Aqui, armazenam-se as coordenadas UV para a textura principal.

Função surf: É a função que define como a superfície do objeto interage com a luz e como os efeitos visuais são aplicados. Aqui vários cálculos são realizados para criar o efeito visual desejado:

- Textura Principal: A textura principal é mostrada e usada como base para a cor do material;
- Mapa Normal: O mapa normal é descompactado para adicionar detalhes de relevo à superfície;
- Efeito de Queimar: Um efeito de incêndio é calculado usando a textura de queima, o mapa de rugosidade e o mapa normal;
- Efeito de Dissolução: Um ruído é gerado e usado para criar um efeito de dissolução, onde partes do material são removidas visualmente;
- Efeito de Piscar: Um efeito de piscar é adicionado à emissão do material, fazendo com que ele brilhe periodicamente;
- Deteção de Bordas: As extremidades do efeito de queimar são detetadas e realçadas com uma cor específica.
- Albedo e Emissão: A cor base (albedo) e a emissão são ajustadas com base nos cálculos anteriores para criar o efeito visual final;
- Suavidade: A suavidade da superfície é definida para controlar como a luz reflete no material;
- Transparência: O valor alpha é ajustado para criar transparência onde necessário.

FallBack: Especifica um shader de reserva para usar se este shader não for compatível com o hardware do usuário, garantindo que o material ainda tenha uma aparência aceitável.

- **“DissolveWindow” (diferenças)**

O shader “DissolveWindow” apresenta algumas diferenças funcionais em relação ao shader “DissolveHouse” anteriormente discutido:

RenderType: O “DissolveWindow” usa "RenderType"="Fade", o que significa que ele suporta transparência e pode ser usado para materiais que precisam de um efeito de desvanecimento ou transparência.

_Color Property: Este shader introduz uma nova propriedade chamada _Color, que permite controlar a cor do material diretamente no shader, multiplicando-a pela textura principal.

Alpha Transparency: A função surf neste shader define explicitamente o valor alpha para 0.66, o que indica que o material terá uma transparência consistente, ao contrário do shader anterior, onde o valor alpha era baseado na textura e no efeito de queima.

Inverted Normal Map:O shader inverte o mapa normal (o.Normal = -unpackNormal(...)), o que pode criar um efeito visual único, como se a luz estivesse interagindo com o material de uma maneira diferente.

Dissolve Effect: O efeito de dissolução é calculado de forma ligeiramente diferente, usando a função step (-_Glossiness, -noise). Isso pode resultar num padrão de dissolução alterado em comparação com o shader anterior.

Smoothness: A suavidade (o.Smoothness) é definida diretamente pelo mapa de rugosidade (specGloss), sem multiplicação adicional pelo valor de _Glossiness.

Essas diferenças alteram a aparência visual e o comportamento do material, oferecendo mais opções para personalizar e ajustar os efeitos visuais de acordo com as necessidades específicas de um projeto.

```
if (steppedNoise > outlineStep1 && steppedNoise > outlineStep2)
{
    // ...
    o.Alpha = c.a * 0.9; // Set the alpha to 90% of the original alpha
}
else if (steppedNoise > outlineStep2)
{
    // ...
    o.Alpha = c.a; // Set the alpha to the original alpha
}
else
{
    // ...
    o.Alpha = c.a; // Set the alpha to the original alpha
}
```

Figura 3 - valor alpha "DissolveHouse"

```
{
    // ...
    o.Alpha = 0.66; // Set the alpha to 0.66
    // ...
}
```

Figura 4 - Valor alpha "DissolveWindow"

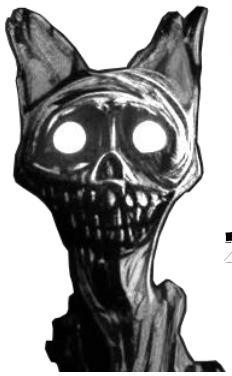
2.1.3. IDEIAS DE APERFEIÇOAMENTO

Tal como foi dito anteriormente, este foi o primeiro shader pensado para integrar este trabalho. Dessa forma, podemos dizer que o efeito que foi concretizado tem todas as funcionalidades que havíamos preconizado no início da sua construção. No entanto, existem sempre coisas a melhorar, e que podem complementar o efeito que atingimos e retirar ainda mais aproveitamento.

Um ponto a reter para a realização de futuros trabalhos é a **integração de partículas e transformação do shader num post-processing**, para realçar o efeito de dissolução. Integrar sistemas de partículas que simulem faíscas, fumo e cinzas que se desprendem do material à medida que ele se dissolve; e efeitos de pós-processamento como 'bloom' e 'glow' poderiam ser utilizados para enfatizar áreas de alta emissão, criando uma sensação mais intensa de calor e energia.

Outra melhoria é a integração de animações dinâmicas para dar vida ao efeito de dissolução, que alteram a textura de incêndio e de rugosidade ao longo do tempoEssas adições visuais não só melhorariam a imersão e o impacto visual dos efeitos, mas também adicionariam uma camada extra de realismo e detalhe, aumentando a qualidade da experiência visual para os usuários.

Figura 5 -Casa com Shaders aplicados.



2.2. TRIPLANAR SHADER - TERRAIN

Figura 6 - Triplanar Shader - Logo.

Todos (ou quase todos) os jogos precisam de um terreno, um solo para que se possa aplicar o desenrolar da ação de um jogo. Por isso mesmo, e no sentido de poder explorar um pouco mais um conceito de shader triplanar, que por falta de tempo não foi abordado em aula; decidimos elaborar um shader que pudesse de forma simples decorar um terreno, e mesclar vários componentes de solo sem grande dificuldade, e com um nível de realismo gratificante.

O shader “Custom/Triplanar/Terrain” é um shader de terreno triplanar que utiliza uma técnica de mapeamento de textura para aplicar texturas a um terreno de forma que elas sejam projetadas corretamente sem distorção visível, independentemente da complexidade da geometria do terreno.

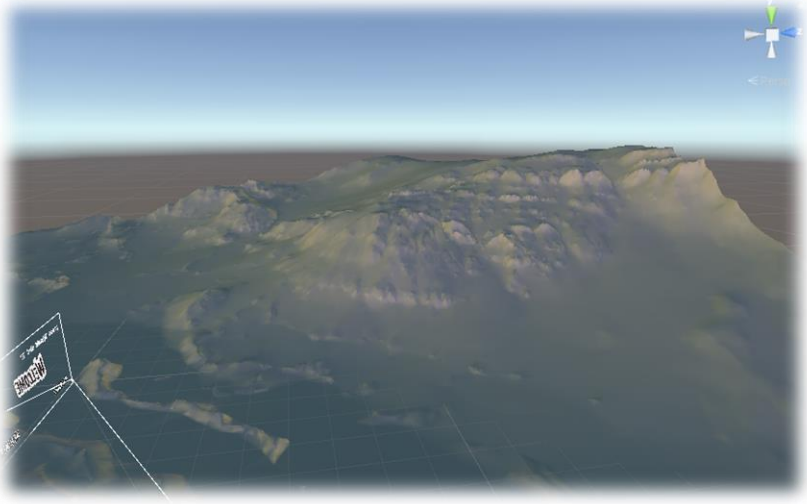
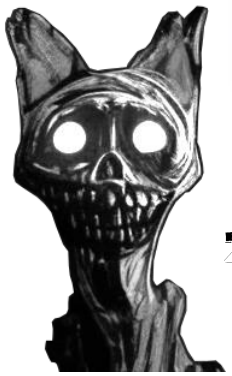


Figura 7 - Terrain da nossa Scene.



2.2.1. CONTEXTUALIZAÇÃO TEÓRICA

Para procedermos à realização deste trabalho recorremos a um dos sites sugeridos pelo docente, o da **Catlike Coding**[3]. O tutorial sobre mapeamento triplanar no site da Catlike Coding descreve uma técnica avançada de texturização que não depende de coordenadas UV ou vetores tangentes. É particularmente útil para geometrias processuais ou de formas arbitrárias, como terrenos ou sistemas de cavernas, onde não é viável gerar coordenadas UV para um mapeamento de textura apropriado.

A técnica utiliza três projeções planares para mapear texturas num objeto, misturando-as com base na orientação da normal da superfície. Isso permite texturizar superfícies de todos os ângulos sem distorção, o que é ideal para terrenos irregulares.

O tutorial também aborda como adaptar shaders existentes para funcionar sem coordenadas UV, introduzindo uma abordagem genérica de superfície e ajustes condicionais no código do shader para lidar com a ausência de UV. Além disso, discute a importância de minimizar a quantidade de amostragem de textura por projeção para manter o desempenho regulável.

Este tutorial é parte de uma série que cobre técnicas de renderização mais complexas e especializadas, indo além dos shaders padrão do Unity.

E sim, o shader "Custom/Triplanar/Terrain" apresentado é uma versão simplificada do conceito de mapeamento triplanar. Enquanto o tutorial da Catlike Coding pode oferecer uma abordagem mais aprofundada e detalhada, com explicações sobre a remoção da dependência de coordenadas UV e tangentes, o nosso shader foca-se nos aspetos essenciais do mapeamento triplanar sem entrar em complexidades adicionais.

A simplicidade do shader pode ser benéfica em termos de facilidade de compreensão e eficiência de desempenho, tornando-o adequado para projetos que requerem uma solução rápida e direta para texturização de terrenos. No entanto, e futuramente para projetos mais avançados que necessitam de maior controle e detalhe, uma abordagem mais complexa como a descrita no tutorial da Catlike Coding pode ser benéfica.

2.2.2. DESCRIÇÃO TÉCNICA

Resumindo, neste shader o elemento fundamental é a utilização da técnica de mapeamento triplanar para aplicar texturas em um objeto 3D. Aqui está uma descrição técnica detalhada de suas partes:

Properties: Define as texturas e o fator de escala que podem ser ajustados no editor da Unity. Cada textura é mapeada em uma das três direções ortogonais do espaço do mundo, e o fator de escala ajusta o tamanho da projeção da textura.

SubShader: Contém a lógica de renderização do shader. As tags indicam que o shader é opaco. O pragma surface especifica que o shader usará o modelo de iluminação padrão do Unity.

Sampler2D e Scale: Variáveis que armazenam as texturas e o fator de escala. São usadas para amostrar as texturas com base na posição do mundo do objeto.

Input: Estrutura que passa informações para a função surf. Inclui coordenadas UV, a normal do mundo e a posição do mundo, que são cruciais para o cálculo do mapeamento triplanar.

Função surf: A função principal que calcula a aparência da superfície. Mostra as texturas com base na posição do mundo e na normal do mundo para aplicar o mapeamento triplanar. Mistura as texturas com base na orientação da normal para criar um efeito de texturização sem emendas. De uma forma mais detalhada podemos ver que dentro desta função shader faz o seguinte:

- Amostragem de Texturas: Usa a posição do mundo para mostrar as três texturas em diferentes orientações (YZ, XZ, XY) e aplica o fator de escala;
- Normalização da Normal: Calcula a contribuição de cada textura com base na normal do mundo, normalizando-a e dividindo-a pela soma de suas componentes para garantir uma mistura suave;
- Combinação de Cores: Mistura as cores das três texturas com base na normal do mundo para criar o efeito final no albedo e na emissão do material.

FallBack: Especifica um shader alternativo (“Diffuse”) para ser usado caso este shader não seja suportado pelo hardware do usuário.

Assim, e tal como foi dito anteriormente, achamos que criamos aqui um shader que é eficiente para criar terrenos realistas com texturas que se adaptam dinamicamente à geometria do terreno, evitando distorções comuns em mapeamentos UV tradicionais.

```
void surf (Input IN, inout SurfaceOutputStandard o) {

    // Sample the textures using the world position and scale
    fixed4 col1 = tex2D(_Texture2, IN.worldPos.yz * _Scale);
    fixed4 col2 = tex2D(_Texture1, IN.worldPos.xz * _Scale);
    fixed4 col3 = tex2D(_Texture3, IN.worldPos.xy * _Scale);

    // Normalize the world normal vector
    float3 vec = abs(IN.worldNormal);
    vec /= vec.x + vec.y + vec.z + 0.001f;

    // Combine the colors based on the normal vector
    fixed4 col = vec.x * col1 + vec.y * col2 + vec.z * col3;

    // Set the albedo and emission colors
    o.Albedo = col;
    o.Emission = col;
}
```

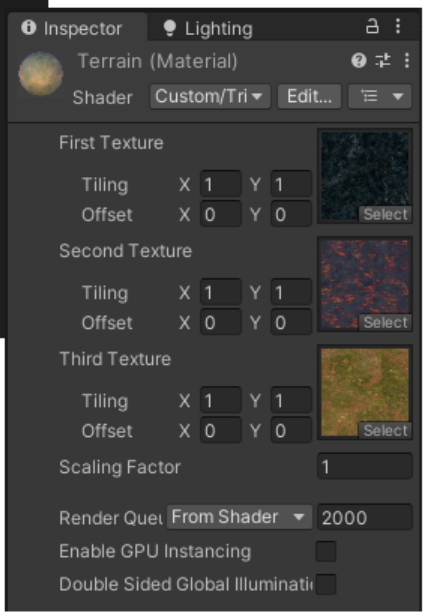


Figura 8 - void Surf do Shader e Inspector do Material.

2.2.3. IDEIAS DE APERFEIÇOAMENTO

Tal como já foi dito no desenvolvimento teórico deste shader especificamente, aqui o objetivo foi simplificar no sentido de adquirirmos conhecimentos introdutórios nesta área. Desta forma, temos a perfeita consciência de existirem várias maneiras de aprimorar a construção deste shader no sentido de o tornar mais eficaz face ao objetivo central de recriar terrenos texturizados com realismo em meshs irregulares.

Das inúmeras sugestões de melhoria que podíamos colocar aqui como o uso de técnicas de deslocamento do tipo parallax ou a implementação de sistemas de controle dinâmicos para a misturas das texturas triplanares; destacamos um: a **tesselação**.

Integrar mapas de altura para cada textura triplanar poderia adicionar uma dimensão extra de realismo ao terreno. Isso permitiria que o shader não apenas aplicasse texturas baseadas na orientação da normal, mas também ajustasse o relevo da superfície do terreno com base na altura especificada no mapa.

2.3. GRABPASS SHADER - MULTIPLE

Figura 9 - GrabPass Shader - Logo.

Seguimos na apresentação do nosso trabalho com aquele que é provavelmente o shader mais versátil que se podem encontrar no nosso trabalho por conter várias funcionalidades e propósitos. Este shader pode ser classificado como um shader de pós-processamento, sendo projetado para ser aplicado sobre o conteúdo já renderizado da tela (usando **GrabPass** para capturar a textura da mesma) e modificar essa imagem com vários efeitos visuais. Os principais efeitos que ele oferece são **Distortion**, onde é criado um efeito de distorção na imagem; **Pixelation** que aplica um efeito de pixelização; **Color Aberration**, onde se modifica a saturação e o brilho das cores da imagem; e **Noise** que acrescenta um efeito de ruído à imagem.

O propósito inicial deste shader era o que dar ao utilizador uma ferramenta versátil e interessante, com múltiplas funcionalidades que interagem entre si; o que consequentemente potencia este shader transformando-o num multiplicador de efeitos visuais que podem ser interessantes e adequar-se às mais variadas atmosferas de jogo.

Para além disso, e à medida que fomos acrescentando conteúdo ao shader, procuramos também fazer um exercício de execução de código, explorando as nossas capacidades para o fazer, e procurando verificar o limite para fazer um shader (coisa que, tal como tínhamos previsto, não verificamos aqui).

Com este shader, abrimos um leque de possibilidades criativas, permitindo aos desenvolvedores de jogos dar vida às suas visões artísticas com facilidade e flexibilidade, transformando simples imagens em experiências imersivas e memoráveis.

2.3.1. CONTEXTUALIZAÇÃO TEÓRICA

Tal como foi dito anteriormente na nota introdutória este shader foi-se construindo, foram sendo acrescentadas funcionalidades que achamos que seriam simples, compatíveis entre si, e que encaixariam nos mais variados processos relacionados com a construção dinâmica de um videojogo criado em Unity.

Desta forma foi também a nossa pesquisa para encontrar os efeitos que queriam adicionar ao nosso espelho mágico. De todos os locais em que pesquisamos este tipo de efeitos destacamos os dois principais impulsionadores para a criação deste shader.

Em primeiro lugar recorremos uma vez mais ao site **Ronja's tutorials**. O site é dedicado a tutoriais de shaders, com o objetivo de tornar a aprendizagem acessível a todos. Focado em shaders Unity com HLSL, o site oferece uma abordagem mais acessível do que as implementações diretas das APIs de shader, mantendo-se flexível. A criadora enfatiza que, embora os tutoriais sejam específicos para uma engine proprietária, as habilidades ensinadas são transferíveis para outros contextos. Para iniciantes, há uma série sobre os fundamentos absolutos dos shaders, explicando como criar um shader simples do zero.

Do leque de possibilidade que aqui temos disponibilizados obtemos duas fontes de inspiração para duas das funcionalidades do nosso shader: o **Perlin Noise** e (numa fase inicial) o **Vertex Displacement.**

O ruído Perlin é um tipo de ruído gradiente desenvolvido por Ken Perlin em 1983. É usado para criar texturas procedurais que parecem naturais e realistas em gráficos computacionais. Ao contrário do ruído aleatório puro, onde cada ponto tem um valor independente dos seus vizinhos, o ruído Perlin garante que pontos próximos tenham valores semelhantes, criando um padrão contínuo e coerente. Isso o torna ideal para simular texturas naturais como fogo, fumo, nuvens e superfícies orgânicas.[4]

Após termos feito o estudo do que encontramos neste tutorial, decidimos implementar algo ligeiramente diferente e original. As diferenças residem principalmente no método de geração do ruído (no site baseado em inclinações, no nosso shader através de uma função de seno para gerar o padrão de ruído); e na aplicação do ruído (mais amplo no tutorial, mais diretamente aplicado para distorcer a textura da tela da nossa parte).

Quanto ao elemento distorção presente no shader seguimos inicialmente o conceito do tutorial apresentado no site como Vertex (wobble) Displacement, tendo evoluído para uma abordagem diferente; visto que o efeito de distorção descrito no shader não se baseia em deslocamento de vértices.

Em vez disso, é um efeito de distorção de textura, onde as coordenadas UV da textura são alteradas para criar um efeito visual final na imagem renderizada. O deslocamento de vértice envolveria a alteração da posição dos vértices da malha em 3D, o que não é o caso aqui. Neste shader, a distorção é aplicada diretamente às coordenadas UV durante a etapa de fragmento (fragment shader), afetando assim a aparência da textura, mas não a forma geométrica do objeto.

Relativamente ao processo de construção dos outros dois efeitos, podemos de consciência tranquila afirmar que tivemos um parceiro na elaboração dos mesmos: **os motores de IA** existentes.[5,6]

Através de um conjunto de indicações e muita persistência nos pedidos, fomos construindo os dois efeitos e todas as suas funcionalidades. Assim, podemos afirmar que inicialmente construímos cada um deles, tendo depois acrescentado as funcionalidades pretendidas em cada um deles, como por exemplo a definição da resolução dos pixéis ou a montagem da animação relacionada com a variação das cores; tendo no final sido um apoio importante para poder harmonizar o código no shader e conseguir fazer com que as funcionalidades conseguissem interagir entre si.

Achamos que a experiência foi enriquecedora, e apesar da temática ser bastante controversa sobretudo na área do ensino, conseguimos ver que o uso deste tipo de ferramenta é por si só um impulsionador de estudo. A experiência mostra que os motores de IA utilizados erram com alguma frequência, o que nos obriga a compreender de forma aprofundada o que está a ser executado.

2.3.2. DESCRIÇÃO TÉCNICA

O nosso shader múltiplo encerra as seguintes propriedades, e subdivide-se nos seguintes setores:

Properties: Define as propriedades do shader que podem ser ajustadas no editor da Unity. Inclui intensidade da distorção, uso de pixelização, saturação e brilho das cores, e intensidade e escala do ruído.

SubShader: Contém as instruções de renderização, incluindo a ordem e o tipo de renderização (transparente neste caso).

GrabPass: Captura o conteúdo atual da tela e armazena-o em uma textura chamada _GrabTexture, que pode ser usada para aplicar efeitos de pós-processamento.

Pass: Define uma passagem de renderização individual dentro do SubShader.

ZWrite Off: Desativa a escrita no buffer de profundidade, o que é comum em operações de pós-processamento onde não queremos alterar a profundidade da cena.

Blend SrcAlpha OneMinusSrcAlpha: Define o modo de mistura para transparência baseada em alpha.

Pragmas: onde se especificam as **funções** vert e frag para shader de vértice e de fragmento.

PerlinNoise: A função que gera ruído Perlin, onde está contido o conjunto de operações matemáticas que criam o efeito de ruído visual.

v2f struct: Define a estrutura de dados que será passada do shader de vértice para o shader de fragmento.

Vertex Shader (vert): Processa cada vértice da malha e calcula a posição na tela e as coordenadas UV para texturização.

Fragment Shader (frag): Processa cada pixel da imagem final e aplica os efeitos de distorção, pixelização, efeito de cor e ruído com base nas propriedades definidas:

- **Distortion:** Usa a função seno para criar um efeito de ondulação na imagem.
- **Pixelation:** Ajusta as coordenadas UV para criar um efeito de pixelização.
- **Color Effect:** Modifica a saturação e o brilho das cores e aplica um padrão de cor dinâmico.
- **Noise:** Adiciona ruído Perlin à imagem para criar um efeito visual de “granulado”.

FallBack: Especifica um shader alternativo para usar se este shader não for suportado pelo hardware do usuário.

Cada seção trabalha em conjunto para permitir que os desenvolvedores ajustem visualmente a cena renderizada, adicionando efeitos visuais complexos e interessantes do qual o utilizador tem controlo.

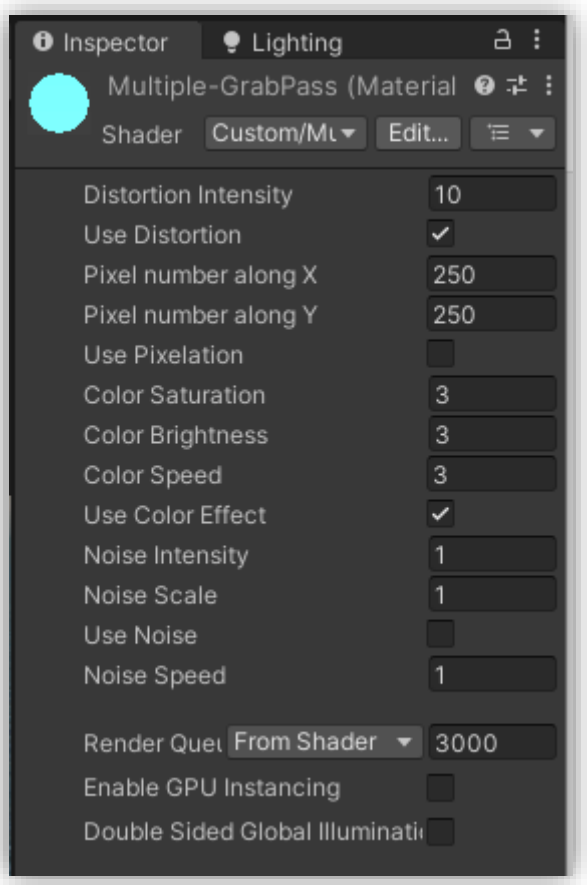


Figura 10 - Multiple-GrabPass Material Inspector.

2.3.3. IDEIAS DE APERFEIÇOAMENTO

A elaboração deste shader teria sempre uma ideia de melhoria a registar: acrescentar mais efeitos aos 4 que já existem. Nesse sentido, e por forma a dar continuidade a um dos métodos implementados para a execução deste shader, perguntamos ao Copilot duas sugestões de melhorias para uma execução futura no código que o compõe (sem recorrer a sugestões de melhoria relacionadas com efeitos exteriores (como a introdução de partículas). A resposta foi:

"Para aperfeiçoar o shader apresentado, sem recorrer a partículas, aqui estão duas sugestões que podem enriquecer os efeitos visuais e a funcionalidade geral:

Efeito de Desfoque de Movimento (Motion Blur): Pode-se adicionar um efeito de desfoque de movimento para simular a sensação de velocidade e movimento. Isso pode ser feito calculando a diferença entre as posições anteriores e atuais dos objetos na cena e aplicando um desfoque na direção do movimento. Isso adicionaria uma camada extra de realismo, especialmente em cenas de ação rápida.

```
// Adicionar propriedades para controlar o desfoque de movimento
_MotionBlurIntensity ("Motion Blur Intensity", Range(0, 1)) = 0.5
[Toggle] _UseMotionBlur ("Use Motion Blur", float) = 0

// No fragment shader
if (_UseMotionBlur > 0.5) {
    // Cálculo simplificado do efeito de desfoque de movimento
    float2 motionVector = ...; // Vetor de movimento baseado na diferença de posição
    half4 blurredColor = tex2D(_GrabTexture, grabUV + motionVector * _MotionBlurIntensity);
    baseColor = lerp(baseColor, blurredColor, _MotionBlurIntensity);
}
```

Figura 11 - Sugestão Copilot - MotionBlur.

Efeito de Vignette: Um efeito de vignette pode ser adicionado para concentrar a atenção do jogador no centro da tela, criando um efeito de sombreamento ou desfoque nas bordas. Isso pode ser particularmente útil para criar uma atmosfera mais imersiva ou para enfatizar momentos dramáticos no jogo.

```
// Adicionar propriedades para controlar o efeito de vignette
_VignetteIntensity ("Vignette Intensity", Range(0, 1)) = 0.5
[Toggle] _UseVignette ("Use Vignette", float) = 0

// No fragment shader
if (_UseVignette > 0.5) {
    // Cálculo simplificado do efeito de vignette
    float2 center = float2(0.5, 0.5);
    float distance = distance(grabUV, center);
    float vignette = smoothstep(0.5, 1.0, distance) * _VignetteIntensity;
    baseColor.rgb *= (1 - vignette);
}
```

Figura 12 - Sugestão Copilot - Vignette.

Estas são apenas duas das muitas maneiras de aprimorar um shader para criar efeitos visuais mais dinâmicos e envolventes. A chave é experimentar e ajustar os parâmetros para se adequar à estética desejada do jogo."

Posto isto, e concluindo este shaders, chegamos mais uma vez à conclusão de que o uso de motores de IA não impede o estudo, por outro lado impulsiona-o. As sugestões dadas são dois efeitos que já tínhamos incluído no nosso trabalho em shaders que apresentaremos neste relatório mais à frente.

2.4. GEOMETRY SHADER - EXTRUDE

Figura 13 - Extrude Shader - Logo.

Este shader proporciona um efeito de extrusão de geometria em tempo real, o que permite criar uma versão "crescida" dos objetos, deslocando seus vértices ao longo das normais das faces.

Para o efeito, este shader utiliza um geometry shader para gerar novos triângulos necessários para a extrusão, incluindo as faces da frente e de trás, bem como as faces laterais. Isso garante que a extrusão seja visualmente completa e consistente.

As vantagens de utilizar este shader em contextos de construção de videojogos incluem a criação de efeitos visuais atraentes, tornando objetos estáticos mais interessantes e visualmente ricos. A capacidade de animar a extrusão em tempo real permite criar efeitos de pulso ou de crescimento que podem responder a eventos no jogo, como power-ups, aumento de nível ou outras interações do jogador. Com várias propriedades

ajustáveis, como fator de crescimento, velocidade e intervalos mínimos e máximos, o nosso shader de extrusão oferece um alto grau de customização, permitindo ajustar facilmente o efeito para diferentes necessidades e estilos visuais.

Como o shader é aplicado no nível da geometria, ele pode ser usado em qualquer objeto 3D na cena, tornando-o versátil para vários tipos de modelos, desde simples objetos de cenário até personagens complexos. Além disso, o shader processa normais e gera novas faces que interagem corretamente com os sistemas de iluminação e sombreamento da Unity, mantendo uma aparência coesa e integrada com o resto da cena.

2.4.1. CONTEXTUALIZAÇÃO TEÓRICA

O nosso shader foi elaborado a partir da visualização do tutorial presente no canal **"Ned Makes Games"** no YouTube.[7] Este canal oferece uma série de tutoriais detalhados e informativos sobre desenvolvimento de jogos, incluindo a criação de shaders avançados em Unity. Através do tutorial específico de extrusão de geometria, foi possível desenvolver um shader que proporciona um efeito dinâmico e personalizável, permitindo ajustar a quantidade de extrusão em tempo real com base em variáveis como fator de crescimento, velocidade e intervalos mínimos e máximos.

O shader utiliza um geometry shader para gerar novos triângulos e aplicar texturas aos objetos extrudidos, garantindo uma integração coesa com os sistemas de iluminação e sombreamento da Unity. A experiência compartilhada pelo canal foi fundamental para a criação deste shader.

No entanto, o shader desenvolvido é mais simples. Encontramos ainda na fase inicial da curva de aprendizagem. Esta simplicidade reflete nosso estado atual de compreensão e prática com shaders em Unity, servindo como uma base sólida sobre a qual podemos construir e aprimorar habilidades mais avançadas no futuro. À medida que progredimos na nossa jornada de aprendizagem, esperamos desenvolver shaders ainda mais complexos e integrados, aproveitando ao máximo os recursos oferecidos por tutoriais especializados como este.

Durante o desenvolvimento do shader de extrusão em Unity, passamos por várias fases e enfrentamos diferentes desafios. Encontrar o cálculo das normais e a geração de novas faces foi um passo particularmente desafiador.

No final encontrar propriedades que fossem significativas para dar funcionalidades foi um passo de debate interno e nem sempre consensual entre nós, pela dificuldade em encontrar o que realmente era importante na implementação deste shader.

2.4.2. DESCRIÇÃO TÉCNICA

A segmentação das funcionalidades técnicas deste Geometry shader pode ser feita da seguinte forma:

Properties: Define as propriedades do shader, como textura principal (_MainTex), fatores de crescimento (_Factor, _MinFactor, _MaxFactor), controle de crescimento (_Toggle), e velocidade (_Speed).

SubShader: Contém as tags e configurações do subshader, como desativação do culling para renderizar todas as faces dos triângulos.

Pass: Define as etapas de processamento do shader, e inclui os **Pragmas:** vertex, geometry e fragment shaders.

Estruturas de Dados: a v2g, estrutura que armazena dados de vértice enviados do vertex shader para o geometry shader; e a g2f que armazena dados enviados do geometry shader para o fragment shader.

Vertex Shader: Transforma os vértices de entrada e envia dados para o geometry shader, incluindo a posição do vértice, coordenadas de textura e normais.

Geometry Shader: A função geom é crucial, pois calcula as normais das faces, determina o fator de crescimento, e gera novos vértices e triângulos para representar as faces extrudidas. O uso de funções auxiliares facilita a organização do código e a geração de geometria complexa. Este shader oferece controle dinâmico sobre a extrusão, permitindo criar efeitos visuais impressionantes e interativos em jogos.

Fragment Shader: Aplica a textura principal (_MainTex) usando as coordenadas UV interpoladas e multiplica pela cor do vértice.

2.4.3. IDEIAS DE APERFEIÇOAMENTO

Para aperfeiçoar o shader de extrusão criado, tanto ao nível de construção de código quanto ao nível de melhoria de efeitos visuais, existem diversas abordagens a serem consideradas. Dividir a lógica em funções menores e mais específicas, como separar o cálculo de normais das faces em uma função dedicada, pode tornar o código mais claro e eficiente.

No que diz respeito à melhoria de efeitos visuais, adicionar efeitos de animação, como pulsação ou ondulação, controlados por parâmetros uniformes ou mapas de textura, pode enriquecer a dinâmica visual do shader. Outra medida poderia ser a implementação de suporte para transparência e blending, e assim criar efeitos de materiais semitransparentes, como vidro. Variedades de fatores de crescimento, controlados por ruído procedural ou texturas, poderiam adicionar diversidade aos efeitos visuais. Fazer com que a extrusão reaja a outros objetos ou forças no ambiente, como colisões ou campos de força, proporcionaria maior interatividade.

Melhorar as sombras projetadas pelas partes extrudidas também aumentaria o realismo e a integração com a iluminação da cena.

Aperfeiçoar este shader em termos de construção de código e efeitos visuais pode transformá-lo num shader funcional e uma ferramenta poderosa e flexível, capaz de criar efeitos visuais ricos e dinâmicos em videojogos. Esses aprimoramentos não só melhoram a eficiência do desenvolvimento, como também o fariam sair da fase embrionária em que se encontra e o entregamos para avaliação.

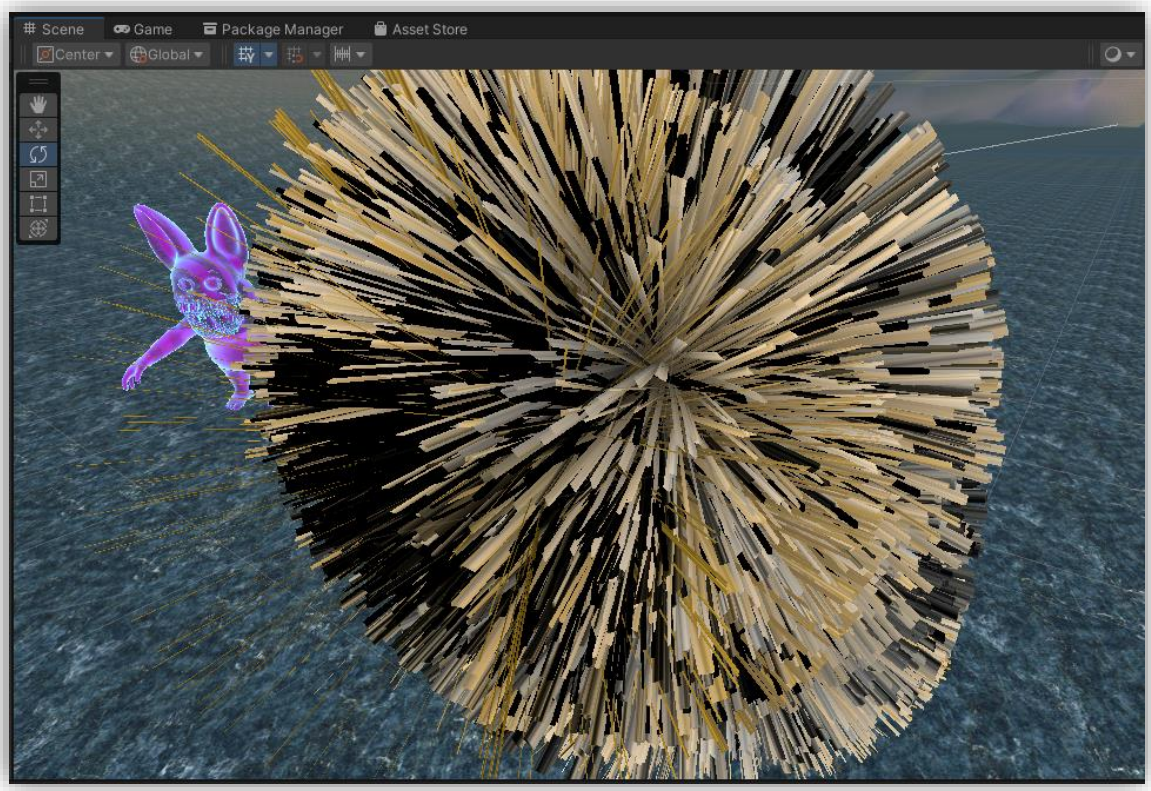


Figura 14 - Extrude Object.

2.5. GEOMETRY SHADER - TRIANGLES

Figura 15 - Triangles Shader - Logo.

O nosso shader "Unlit/Geometry/Geometry-Triangles" pertence à categoria de shaders não iluminados (unlit) e utiliza a geometria para processamento. Este shader é projetado para trabalhar com os triângulos individuais de uma mesh, permitindo manipulações complexas de vértices e arestas antes da rasterização.

Um **Geometry shader** é um tipo de shader que opera entre o shader de vértice e o shader de fragmento no pipeline gráfico. Ele recebe primitivas geométricas (como pontos, linhas ou triângulos) do shader de vértice e tem a capacidade de gerar novas primitivas ou modificar as existentes. Isso permite a criação de efeitos complexos que não seriam possíveis apenas com o shader de vértice, como a expansão de geometria, a criação de formas complexas a partir de primitivas simples, e a manipulação de características como normal e posição dos vértices.

2.5.1. CONTEXTUALIZAÇÃO TEÓRICA

O Geometry shader pode ser usado para várias operações visuais que podem acrescentar bastante a performance de um videojogo. São elas:

- **Amplificar a geometria:** Gerar mais geometria do a que foi fornecida pelo vertex shader.

- **Modificar a geometria:** Alterar a forma das primitivas recebidas.
- **Descartar a geometria:** Não passar certas primitivas para o próximo estágio do pipeline.
- **Criar efeitos específicos:** São exemplos explosões de partículas, cabelo ou pelo e relva dispostos de forma dinâmica, ou a implementar técnicas como renderização em camadas.

Em suma, o geometry shader oferece uma camada adicional de controle sobre a geometria de uma cena, permitindo transformações e efeitos que podem melhorar significativamente a qualidade visual de uma aplicação gráfica.

O nosso apoio para a construção deste shader foi, uma vez mais, os tutoriais do site Catlike Coding, mais especificamente o tutorial **"Flat and Wireframe Shading"** que aborda técnicas avançadas de flat shading e a visualização de meshs em wireframe. O tutorial ensina como usar derivadas no espaço da tela para encontrar as normais dos triângulos e como realizar o mesmo processo através de um geometry shader.

Aqui podemos retirar uma explicação de como expor os triângulos de uma malha para criar um efeito de sombreamento plano, que resultam em meshs com uma aparência facetada. Este tutorial revelou-se uma de aprendizagem. Conseguimos compreender que o domínio destas técnicas pode adicionar uma camada extra de estilo artístico aos nossos jogos.

Comparando a nossa abordagem com os conceitos teóricos apreendidos, podemos dizer que conseguimos entregar parte do que vimos no tutorial. Como semelhanças podemos dizer que conseguimos de fato executar a manipulação de uma mesh com a aplicação deste shader, e utilizar técnicas de sombreamento que alteram a aparência visual do objeto.

No entanto, o tutorial do Catlike Coding é mais complexo, abordando técnicas como sombreamento plano e wireframe, enquanto o shader que fizemos é mais simples, focando-se apenas na renderização básica de triângulos.

De forma resumida, o nosso shader "Unlit/Geometry/Geometry-Triangles" é uma implementação mais direta e menos complexa, adequada para renderização básica de triângulos com iluminação simples. Conseguimos servir o propósito a que nos propusemos e responder ao desafio de renderizar usando esta técnica.

2.5.2. DESCRIÇÃO TÉCNICA

De forma o mais detalhada possível tentaremos explicar o nosso shader, dando especial ênfase à função geom, que é usada com geometry shader. Desta forma, o nosso shader é constituído por:

Properties: Define as propriedades do shader que podem ser ajustadas no editor da Unity. Aqui, temos _Color, que define a cor do shader, e _MainTex, que é a textura principal aplicada à malha.

SubShader: Contém as instruções de renderização, incluindo a ordem de renderização (Queue), o tipo de renderização (RenderType) e o modo de iluminação (LightMode).

Pass: Define uma passagem de renderização individual dentro do **SubShader**. É aqui que o código do shader é realmente implementado (CGPROGRAM).

#include "UnityCG.cginc": Inclui um arquivo de código comum necessário para muitos shaders no Unity, como transformações de vértices e cálculos de iluminação. Por exemplo, ele contém funções para transformar pontos do espaço do objeto para o espaço da câmara ou para o espaço de recorte, bem como funções para calcular a direção da visão no espaço do mundo ou no espaço do objeto. [8]

Pragmas: com #pragma vertex vert, que especifica que a função vert será usada como o shader de vértice; #pragma geometry geom, que especifica que a função geom será usada como o geometry shader; e #pragma fragment frag, que especifica que a função frag será usada como o shader de fragmento.

Vertex Shader (vert): Processa cada vértice da mesh, calculando a posição transformada e as coordenadas UV.

Geometry Shader (geom): Este é o coração do shader e onde a magia acontece. O geometry shader recebe primitivas (neste caso, triângulos) do shader de vértice e pode modificar a geometria ou criar novas primitivas. Neste shader específico:

- [maxvertexcount(3)]: Define o número máximo de vértices que o geometry shader pode emitir por primitiva, que é 3 para um triângulo.
- void geom(triangle v2g IN3, inout TriangleStream<g2f> triStream): A função geom é chamada para cada triângulo da malha. Ela recebe um array de três vértices (IN) e um stream de triângulos para saída (triStream).

- //Compute the normal: Calcula a normal do triângulo usando o produto cruzado dos vetores formados pelos vértices do triângulo.
- //Compute diffuse light: Calcula a luz difusa baseada na direção da luz e na normal do triângulo.
- //Compute barycentric uv: Calcula as coordenadas UV barycentric, que são usadas para texturização.
- //Append vertices: Adiciona os vértices processados ao stream de saída.

Fragment Shader (frag): Processa cada pixel da imagem final, aplicando a textura e a cor baseada na intensidade da luz calculada pelo geometry shader.

Fallback "Diffuse": Especifica um shader alternativo para usar se este shader não for suportado pelo hardware do usuário.

No contexto deste shader, o geometry shader é usado para calcular a iluminação por vértice e para ajustar a geometria antes de ser rastreada pelo fragment shader; resultando num efeito de mosaico, "pixelizando" a imagem em triângulos.

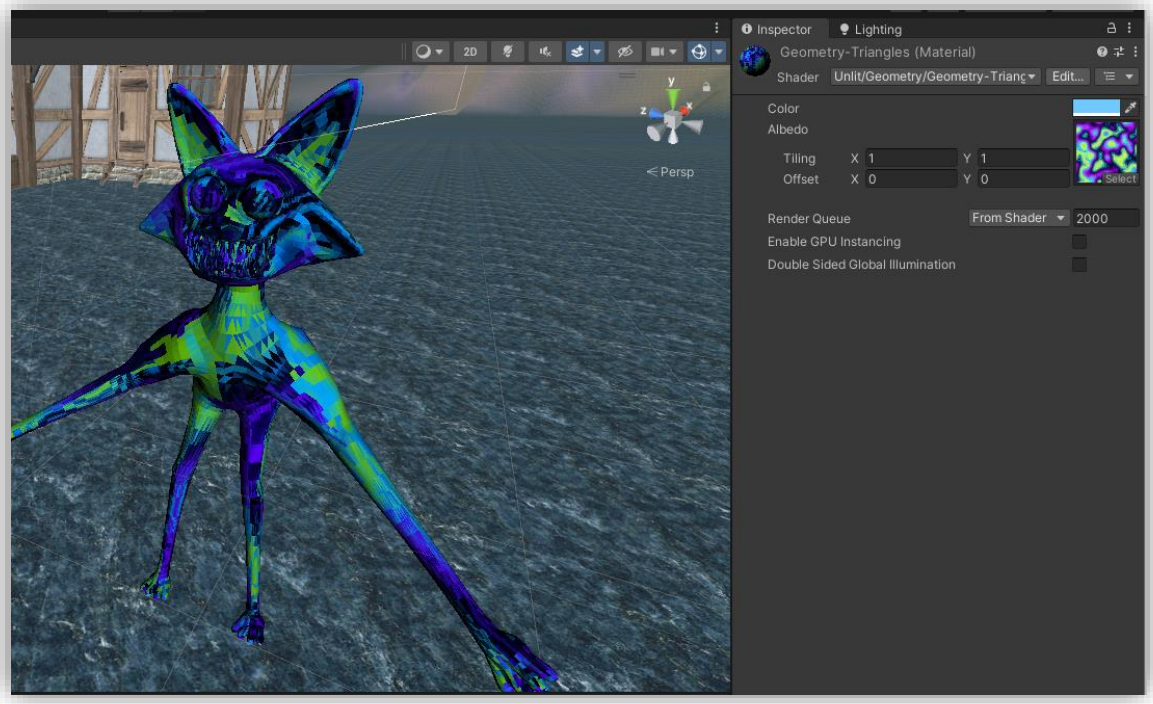


Figura 16 - Objecto com Material aplicado - Pormenor do Inspector.

2.5.3. IDEIAS DE APERFEIÇOAMENTO

Para aprimorar o shader “Unlit/Geometry/Geometry-Triangles”, poderíamos considerar algumas melhorias que enriqueceriam a sua funcionalidade. Primeiramente, poderíamos (e podemos sempre fazê-lo de forma relativamente simples) introduzir uma maior flexibilidade no controlo da luz e sombra, permitindo ajustes mais finos na intensidade e direção da iluminação. Isso poderia ser alcançado através da adição de propriedades adicionais no shader, que os usuários poderiam manipular dentro do editor da Unity.

Outra melhoria seria a implementação de um sistema de LOD (Level of Detail), que ajustaria a complexidade da geometria com base na distância da câmara. Isso não só otimizaria o desempenho em cenas mais complexas, mas também manteria a qualidade visual quando visto de perto. Este aspeto em concreto é algo que por falta de tempo, não conseguimos aprofundar com detalhe.

A manipulação do shader de geometria é, de fato, uma área complexa, mas com grande potencial para melhorias e inovações. A complexidade advém da capacidade do geometry shader de gerar e modificar geometria em tempo real, o que pode ser computacionalmente intensivo e desafiador para otimizar. No entanto, essa mesma complexidade oferece oportunidades para criar efeitos visuais impressionantes e dinâmicos que não seriam possíveis de outra forma.

Melhorar a manipulação do geometry shader pode envolver a implementação de técnicas mais eficientes de processamento de geometria, como o uso de tesselação adaptativa para detalhar a geometria onde é mais necessário, ou simplificá-la onde menos detalhes são visíveis.

De forma muito resumida, faltou dar ao usuário uma ferramenta que o deixasse controlar o número de triângulos em que queria subdividir a renderização da mesh e das suas texturas, algo que pela complexidade do processo, não nos desafiamos a atingir. Por agora.

2.6. POST PROCESSING SHADER - ENDING SPOTLIGHT

Figura 17 - End Spotlight Shader - Logo.

O "Hidden/Ending-Spotlight," cria um efeito visual de destaque onde uma área centrada na tela é iluminada e as áreas circundantes são gradualmente escurecida. A intensidade do efeito diminui à medida que a distância do ponto central aumenta, com a taxa de decaimento controlada pelos parâmetros _Radius e _Sharpness.

Os principais efeitos deste shader incluem a iluminação centralizada, onde uma região circular no centro da tela é destacada, tornando-a mais brilhante em comparação com as áreas ao redor. Além disso, há um decréscimo gradual da intensidade da iluminação a partir do centro até as bordas do círculo definido pelo raio. A transição entre áreas claras e escuras pode ser ajustada com a nitidez (_Sharpness), que controla quão abrupta ou suave é a mudança de intensidade. A flexibilidade de configuração é outra característica importante, permitindo que o utilizador ajuste a posição central do efeito na tela, bem como o tamanho e a nitidez do efeito.

Em termos de vantagens e funcionalidades em videojogos, este shader pode ser utilizado para direcionar a atenção do jogador para uma área específica da tela, importante para momentos narrativos, por exemplo. Além disso, o efeito de iluminação pode criar uma atmosfera dramática ou misteriosa, aumentando a imersão do jogador ao enfatizar ou ocultar certas partes da cena. Pode também servir como um indicador visual para destacar itens coletáveis, pontos de interesse ou áreas interativas no ambiente de jogo. Além disso, é leve em termos de processamento, pois a maioria dos cálculos são realizados no fragment shader.

2.6.1. CONTEXTUALIZAÇÃO TEÓRICA

A elaboração deste shader contribuiu positivamente para o estudo das luzes envolvida num processo de renderização. No nosso caso, o efeito de iluminação é aplicado gradualmente, com a intensidade diminuindo conforme a distância do centro aumenta. Isso é controlado por uma interpolação suave, permitindo uma transição contínua entre áreas iluminadas e escuras. Vários parâmetros ajustáveis, como posição central, raio e nitidez, permitem controlar o comportamento do efeito, oferecendo flexibilidade para ajustar a aparência conforme necessário. A fórmula usada para calcular a intensidade do efeito envolve operações aritméticas e a função de potência, controlando quão abrupta ou suave é a transição do efeito. A cor final do pixel é obtida multiplicando a cor original da textura pelo fator de efeito calculado, escurecendo os pixels conforme eles se afastam do centro e criando um efeito de destaque visual.

Em resumo, o shader "Hidden/Ending-Spotlight" utiliza conceitos teóricos de amostragem de textura, transformação de coordenadas, cálculo de distância (a distância entre dois pontos em um espaço bidimensional é calculada usando a fórmula da distância euclidiana. Neste shader, isso é utilizado para determinar a distância de cada pixel ao centro do efeito), interpolação de efeitos (que neste caso são aplicados gradualmente, com a intensidade do efeito a diminuir conforme a distância do centro aumenta), operações aritméticas (função que eleva a distância normalizada à potência de um parâmetro de nitidez, controlando quão abrupta ou suave é a transição do efeito) e multiplicação de cores. Esses conceitos permitem criar um efeito visual dinâmico e personalizável, que pode ser utilizado em vários contextos dentro de videojogos para aprimorar a experiência visual e narrativa.

Para complementar o estudo e elaboração do nosso shader seguimos vários elementos bibliográficos de autores descritos anteriormente.

2.6.2. DESCRIÇÃO TÉCNICA

A estrutura do nosso shader está, como foi dito no passo inicial de descrição; elaborada de forma muito simples. O shader "Hidden/Ending-Spotlight" apresenta os seguintes segmentos:

Properties: incluem a textura principal (_MainTex), as coordenadas do centro do efeito (_CenterX e _CenterY), o raio (_Radius) e a nitidez (_Sharpness) do efeito de iluminação.

SubShader: contém uma única **Pass**, que especifica as operações de renderização. Dentro da Pass, são definidos o vertex shader e o fragment shader, que são responsáveis por processar os vértices e os fragmentos (pixéis), respetivamente.

Vertex Shader: O vertex shader é especificado pela diretiva **#pragma vertex vert_img**, indicando que usa a função vert_img fornecida pela biblioteca **UnityCG.cginc**. Esta função é responsável por transformar os vértices dos objetos 3D para a posição de tela.

Fragment Shader: O fragment shader é onde a maior parte da lógica do efeito é implementada. Ele é especificado pela diretiva **#pragma fragment frag** e usa a função frag. A função frag recebe uma estrutura v2f_img, que contém a posição do fragmento na tela (i.pos) e as coordenadas de textura (i.uv).

Dentro do fragment shader, os seguintes conceitos teóricos são aplicados:

- Amostragem de Textura: O comando tex2D(_MainTex, i.uv) é utilizado para amostrar a textura principal (_MainTex) nas coordenadas de textura (i.uv). Isso retorna a cor do fragmento da textura, que será modificada pelo efeito de iluminação.
- Cálculo da Distância: A distância do fragmento ao centro do efeito é calculada usando a função distance. O centro é definido pelas coordenadas (_CenterX, _CenterY), e a posição do fragmento é obtida pela transformação da posição da tela (ComputeScreenPos(i.pos).xy / _ScreenParams.xy).
- Parâmetros de Controle: Vários parâmetros ajustáveis controlam o efeito. _CenterX e _CenterY definem a posição central do efeito na tela. _Radius define o tamanho do círculo onde o efeito é aplicado, e _Sharpness controla a nitidez da transição entre a área iluminada e a área escurecida.
- Aplicação do Efeito de Iluminação: O efeito de iluminação é aplicado com base na distância calculada. A intensidade do efeito é determinada pela expressão 1 - pow(dist / _Radius, _Sharpness). Aqui, a função pow eleva a razão da distância pelo raio à potência da nitidez, criando uma transição suave ou abrupta dependendo do valor de _Sharpness.
- Combinação de Cor e Efeito: A cor final do fragmento é obtida multiplicando a cor original da textura pelo fator de efeito calculado (col * effect). Isso escurece a cor conforme a distância do centro aumenta, criando o efeito de destaque centralizado.

Resta ainda salientar que recorremos ao uso de **UnityCG.cginc**. Como foi dito anteriormente, é uma prática comum em shaders Unity, pois inclui funções utilitárias e macros que facilitam a escrita de shaders. A função **ComputeScreenPos** e a variável **_ScreenParams** são exemplos de utilitários fornecidos por esta biblioteca.

2.6.3. IDEIAS DE APERFEIÇOAMENTO

Para melhorar este shader, surgiram algumas ideias após a finalização do projeto. Sentimos que o shader poderia encerrar ainda mais funcionalidades e acrescentar opções e propósitos ao seu uso, nomeadamente fazer uma animação do Spotlight, adicionando parâmetros de tempo para animar o movimento do seu centro, permitindo que ele se desloque pela tela ou siga objetos em movimento.

Outra adição prende-se com a variação de cores, incorporando como feature uma transição de cores nas extremidades do spotlight, criando efeitos visuais mais ricos e dinâmicos. Dentro do acréscimo de efeitos visuais outro acréscimo poderia passar por acrescentar efeitos de fading (fade in e fade out do spotlight) para suavizar a ativação e desativação do efeito, melhorando a experiência visual.

Estes aperfeiçoamentos podem enriquecer significativamente o impacto visual do shader em jogos, tornando-o ainda mais flexível e atraente para diversas situações de jogo; que era o principal objetivo neste trabalho.

2.7. POST PROCESSING SHADER - GLITCH - BLUR EFFECT

Figura 18 - Glitch-Blur effect - Logo.

O shader "Hidden/Glitch-Blur" proporciona dois efeitos principais: um efeito de glitch e um efeito de desfoque gaussiano. De facto, um dos nossos Post-Processing shader faz

uso destas funcionalidades para atingir uma sensação de tremor e distopia, que segue

em linha com o tema horror/bizarria que procura caracterizar o nosso conjunto de shaders.

O efeito de Glitch altera as posições dos canais de cores (vermelho, verde e azul) de forma independente, criando um efeito visual de distorção ou "glitch". Cada canal de cor pode ser deslocado por meio de propriedades ajustáveis (_RedX, _RedY, _GreenX, _GreenY, _BlueX, _BlueY), resultando em um efeito que lembra erros de transmissão de vídeo ou falhas digitais.

O efeito gaussiano é usado para suavizar a imagem. Isto é feito calculando uma média ponderada dos pixels vizinhos, usando um conjunto de deslocamentos e pesos predefinidos. O nível de desfoque é ajustável pela propriedade _BlurAmount. Este desfoque pode ser aplicado de forma uniforme ou combinado com o efeito de glitch para criar uma aparência visualmente interessante e única.

Em jogos este efeito possui uma estética distintiva podendo ser utilizado para criar uma estética visual única, especialmente em jogos que envolvem temas de cyberpunk, ficção científica ou distopia. Pode também ser usado para indicar um erro no sistema ou dano eletrônico de maneira estilizada.

Alem disso, o efeito de desfoque, combinado com o glitch, pode ser usado para direcionar a atenção do jogador para áreas específicas da tela ou para criar uma sensação de confusão ou desorientação temporária. Em jogos, o efeito de glitch pode ser utilizado como um feedback visual para indicar um evento específico, como a ativação de uma habilidade especial, uma interferência eletrônica ou uma transição de fase.

2.7.1. CONTEXTUALIZAÇÃO TEÓRICA

Uma das bases conceptuais de apoio para a construção deste shader foi (uma vez mais) um dos tutoriais de **Ronja**. A página "Blur Postprocessing Effect (Box and Gauss)" do site fornece um guia detalhado sobre como implementar um efeito de desfoque (blur) em shaders, com foco em dois tipos principais de desfoque: o desfoque de caixa (box blur) e o desfoque gaussiano (Gaussian blur).

Aqui explica-se a importância dos efeitos de desfoque em gráficos de computador e como eles podem ser utilizados para melhorar a qualidade visual em jogos e outras aplicações gráficas. O tutorial é voltado para quem deseja aprender a implementar esses efeitos diretamente nos shaders usando a linguagem de programação HLSL (High-Level

Shading Language).

Relativamente ao Gaussian Blur refere que é um tipo de desfoque é mais avançado e produz resultados visualmente mais suaves. A página explica o conceito matemático por trás do desfoque gaussiano, que envolve o uso de uma função gaussiana para calcular os pesos dos pixels vizinhos.

O efeito de glitch foi recolhido através de visualização de tutoriais de variados canais de youtube, e sendo construído a partir da interpretação de como poderia "partir" os canais de cor RGB e manipular de forma independente.

Figura 19 - Glitch-Blur na Demo Scene.

2.7.2. DESCRIÇÃO TÉCNICA

Quanto à divisão do código elaborado pelas suas principais partes, apresentamos o seguinte esquema explicativo:

Properties: As propriedades definem os parâmetros ajustáveis que podem ser configurados diretamente no Unity Inspector. Estão definidas a _MainTex (textura principal), _RedX, _RedY (Deslocamentos para o canal vermelho da textura), _GreenX,

_GreenY (Deslocamentos para o canal verde da textura), _BlueX, _BlueY (Deslocamentos para o canal azul da textura); e o _BlurAmount, que controla a intensidade do efeito do filtro gaussiano.

SubShader: Define como o objeto será renderizado. Neste caso, é definido como "Opaque", indicando que não é transparente.

Pass: Onde os shaders CG (C for Graphics) são definidos.

Pragmas: #pragma vertex vert e #pragma fragment frag: Especifica as funções que atuam como os shaders de vértice e fragmento, respetivamente.

#include "UnityCG.cginc": Inclui a biblioteca comum de shaders do Unity, que fornece funções e variáveis úteis para shaders.

Estruturas: a struct appdata_t, que define os dados de vértice, incluindo a posição e coordenada de textura; e a struct v2f, que define os dados interpolados para o fragment shader, incluindo a posição do vértice e coordenada de textura.

Vertex Shader (vert): Transforma a posição do vértice do espaço do objeto para o espaço de recorte. Passa ainda a coordenada de textura diretamente para o fragment shader.

Fragment Shader (frag): Calcula as coordenadas de textura ajustadas para os canais de vermelho, verde e azul com base nos deslocamentos especificados.

De seguida, realiza a leitura da textura para cada canal (R, G, B) e mantém o canal alpha da textura original.

Aplica um efeito de borrão gaussiano sobre a textura usando uma matriz de pesos e deslocamentos. Finalmente aplica um processo que mistura o resultado do efeito de glitch com o efeito de blur, controlado pela intensidade _BlurAmount.

2.7.3. IDEIAS DE APERFEIÇOAMENTO

Da construção deste shader ficou a ideia de que uma das variáveis poderia ter sido mais explorada para aumentar as possibilidades de aplicabilidade: o tempo.

Atualmente, o shader aplica deslocamentos estáticos nos canais de cor, o que cria um efeito de glitch uniforme e previsível.

Para tornar o efeito mais interessante e realista, poderia-se considerar a introdução de variações temporais nos deslocamentos. Isso poderia ser alcançado adicionando uma componente de ruído ou aleatoriedade aos deslocamentos _RedX, _RedY, _GreenX, _GreenY, _BlueX, _BlueY ao longo do tempo. Isso faria com que o glitch parecesse mais orgânico e imprevisível, sem perder o controle sobre a intensidade e direção do efeito.

Além disso, explorar diferentes técnicas de mistura entre o efeito de glitch e o efeito de blur poderia criar transições mais suaves e estilizadas, adaptando-se melhor ao conteúdo da textura. Isso poderia envolver o ajuste das curvas de interpolação (lerp) ou a aplicação de máscaras adicionais para controlar onde e como os efeitos são aplicados.

Em resumo, ao adicionar elementos de variação temporal aos deslocamentos de glitch e explorar técnicas avançadas de mistura e ajuste; possibilitaria que o nosso shader produzisse um efeito final ainda mais dinâmico, estilizado e atraente para uma variedade maior de aplicações visuais.

2.8. POST PROCESSING SHADER - SOBEL - GAUSSIAN EFFECT

Figura 20 - Sobel-Gaussian Effect - Logo.

O shader aqui apresentado chama-se “Hidden/Sobel-Gaussian”, e combina dois efeitos de processamento de imagem: o filtro **Sobel** e o filtro **Gaussiano**. O filtro Sobel é utilizado para detetar extremidades na imagem, realçando as transições de intensidade e criando um contorno nítido. Já o filtro Gaussiano é um filtro de suavização que reduz o ruído e os detalhes, resultando em uma imagem mais suave.

No contexto dos jogos, este shader pode ser extremamente útil para criar efeitos visuais distintos, como enfatizar elementos importantes em uma cena ou criar um estilo visual único. Por exemplo, o filtro Sobel pode ser usado para criar um efeito de esboço ou desenho animado, enquanto o filtro Gaussiano pode ser usado para simular efeitos de profundidade de campo ou visão desfocada. A combinação desses filtros pode ser ajustada para criar uma variedade de efeitos visuais que podem melhorar a experiência do jogador, adicionando atmosfera, foco e estilo artístico ao jogo.

2.8.1. CONTEXTUALIZAÇÃO TEÓRICA

Antes de iniciarmos a descrição das fontes que inspiraram a realização de shader do tipo **Post Processing** passemos a encontrar uma definição simples dos dois filtros utilizados no nosso código, e que no final dão um efeito visual bastante interessante e apelativo.

O filtro Sobel é um operador de processamento de imagem que destaca as extremidades e transições de intensidade em uma imagem. Calcula o gradiente da intensidade luminosa dos pixels para detetar regiões de alta variação de claro para escuro, o que é crucial na deteção de contornos.[9]

O filtro Gaussiano é um tipo de filtro de suavização que utiliza a distribuição normal ou gaussiana para atenuar o ruído e detalhes de uma imagem. Ele é amplamente utilizado para criar um efeito de desfoque, suavizando transições e removendo ruídos indesejados, mantendo a qualidade visual da imagem.[10]

Passando à construção do nosso shader, o ponto inicial foi ter encontrado um shader em linguagem GLSL. Assim, o shader adotado para a construção do nosso foi um shader presente no site **Shadertoy,** uma plataforma online onde artistas e programadores podem criar e compartilhar shaders WebGL. Neste caso específico, usamos um intitulado “Edgy Hallway”, criado pelo usuário **frettini** em 2023. [11]

Podemos, no entanto, afirmar que apenas se tratou de um ponto de partida, e que o shader criado não é uma transcrição pura do que foi analisado no trabalho de frettini. De facto, ambos os shaders usam filtros Sobel e Gaussiano para processar a imagem; os kernels estão definidos de forma similar; e as suas funções especificas estão definidas de igual forma no “Hidden/Sobel-Gaussian” que aqui apresentamos.

No entanto, as semelhanças terminam nesse ponto. Por forma a podermos obter uma aprendizagem positiva neste processo de “cópia” que achamos necessário para obter todas as experiências de criação que o mundo dos shaders nos apresenta, simplificamos o conceito do shader original; para podermos futuramente aprofundar o conceito com tempo para o fazer, e consolidar o nosso conceito.

O shader do Shadertoy usa anti-aliasing (definido pela macro AA, uma técnica usada em imagens digitais para reduzir os defeitos visuais) para suavizar os edges; e para além disso, inclui uma combinação de cores mais pormenorizada baseada nos resultados dos filtros, criando um efeito visual específico.

Finalmente, e de forma óbvia, o shader de frettini é escrito para o ambiente do Shadertoy, que tem suas próprias variáveis uniformes e maneira de lidar com entradas e saídas, enquanto o shader "Hidden/Sobel-Gaussian" é escrito para ser usado dentro da Unity.

2.8.2. DESCRIÇÃO TÉCNICA

Tal como descrito anteriormente, o "Hidden/Sobel-Gaussian", é um shader de pós-processamento para Unity que utiliza dois filtros para criar efeitos visuais. Dissecando em cada segmento e nas suas funções principais temos:

Properties: Define as propriedades editáveis do shader, que são a textura principal (_MainTex), a resolução da tela (_Resolution), o fator de transição (_TransitionFactor) e duas cores para efeitos (_Color1 e _Color2).

SubShader: Contém as instruções de renderização, incluindo o tipo de renderização (opaco) e o nível de detalhe (LOD).

Pass: Define uma passagem de renderização que contém o código do programa de sombreamento (**CGPROGRAM**).

Pragmas: com **o** #pragma vertex vert que indica que a função vert é o shader de vértice; e o #pragma fragment frag que nos diz que a função frag é o shader de fragmento.

Vertex Shader (vert): Processa cada vértice da malha, passando a posição transformada e as coordenadas UV para o shader de fragmento.

SobelFilter: Uma função que aplica o filtro Sobel ao fragmento, realçando as extremidades da imagem.

GaussianFilter: Uma função que aplica o filtro Gaussiano ao fragmento, suavizando a imagem.

mainImage: Calcula a cor final de cada pixel combinando os resultados dos filtros Sobel e Gaussiano com as cores definidas nas propriedades.

Fragment Shader (frag): Usa a cor calculada pela função mainImage e aplica uma transição suave entre a cor original da textura e a cor filtrada, baseada no fator de transição.

FallBack "Diffuse": Especifica um shader de fallback caso este não seja suportado pelo hardware.

As funções principais deste shader são as funções SobelFilter e GaussianFilter, que aplicam os respetivos efeitos de pós-processamento à imagem renderizada. A função mainImage combina esses efeitos para criar a cor final que será exibida na tela. O shader é projetado para ser flexível, permitindo aos desenvolvedores ajustar facilmente os efeitos visuais através das propriedades expostas.

2.8.3. IDEIAS DE APERFEIÇOAMENTO

O shader apresentado assenta em conteúdos de estudo com alguma complexidade. De facto, foi isso mesmo que nos fez arriscar na elaboração deste shader. Partir de algo que já existe é um dos caminhos para a construção de shaders, e queríamos tirar partido dessa experiência para poder retirar as nossas próprias conclusões. Por isso mesmo, apesar de acharmos que o resultado final é chamativo para o utilizador; temos a plena consciência de que o trabalho poderia ser melhorado e aprofundado.

Para melhorar o shader "Hidden/Sobel-Gaussian" tanto em termos de efeito visual quanto na construção do código, deixamos aqui alguns apontamentos. Ao nível de melhorias no efeito visual, podíamos ter explorado mais os elementos que "abandonamos" do conceito inicial, que são a otimização do deslocamento cromático, tornando-o mais controlável e menos aleatório, introduzindo parâmetros adicionais que permitam aos desenvolvedores ajustar a amplitude e a frequência do efeito de acordo com as necessidades específicas da cena. Para além disso, fazer um refinamento do Anti-Aliasing, implementando técnicas mais avançadas como MSAA (Multisample Anti-Aliasing) ou FXAA (Fast Approximate Anti-Aliasing) para obter edges ainda mais suaves sem um impacto significativo no desempenho.

Quanto a melhorias na Construção do Código:

- Modularização: Atualmente, as funções de filtro estão todas contidas dentro do mesmo bloco de código. Podemos modularizar o código, separando cada filtro e função em suas próprias unidades lógicas. Isso não só torna o código mais legível, mas também facilita a manutenção e a reutilização em diferentes partes do projeto.

- Otimização de Desempenho: O shader pode ser otimizado para desempenho ao minimizar o número de acessos à textura e operações de loop. Por exemplo, poderíamos pré-calcular e armazenar os resultados de operações comuns em variáveis temporárias para reduzir a redundância.

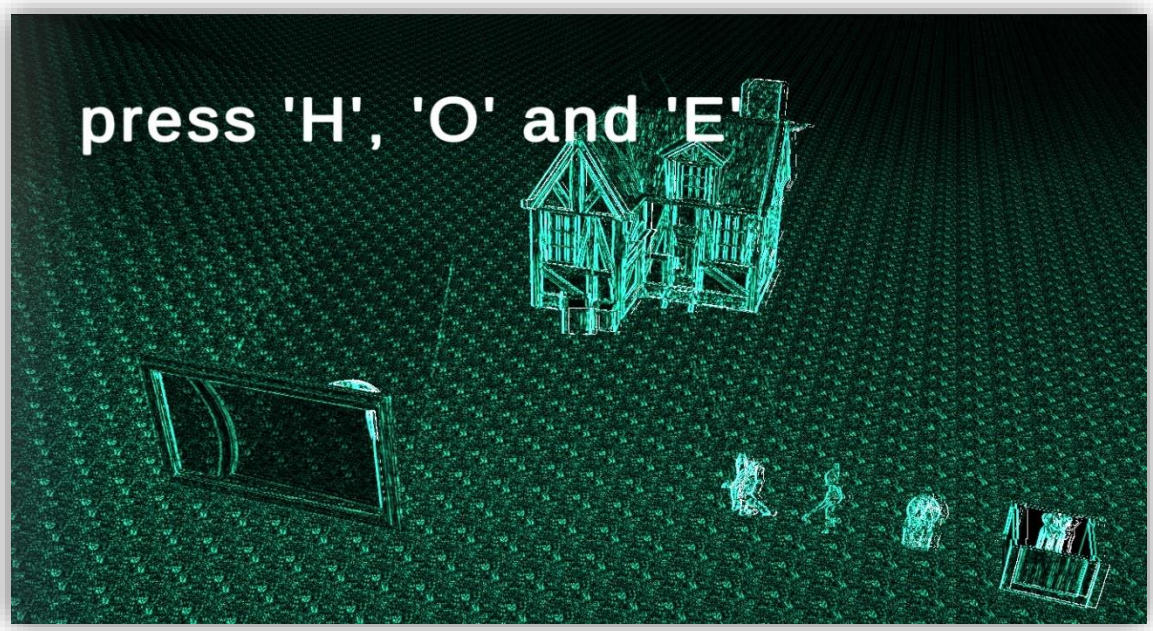


Figura 21 - Demo Scene com apresentação NightVision.

2.9. VERTEX/FRAGMENT SHADER - FRESNEL

Figura 22 - Fresnel - Logo.

O efeito Fresnel é um fenómeno visual que se descreve como a reflexão da luz que varia dependendo do ângulo de incidência entre a luz e a superfície. Em jogos de computador, o efeito Fresnel é frequentemente implementado através de shaders para simular características visuais mais realistas e imersivas.

Quando a luz atinge uma superfície, especialmente em materiais não metálicos como plástico, vidro ou água, parte da luz é refletida outra parte é refratada. O efeito Fresnel modula a intensidade da reflexão especular com base no ângulo de visão do observador em relação à superfície.

A aplicação de shaders para implementar o efeito Fresnel em jogos permite melhorar a aparência dos materiais, adicionando um toque de realismo à interação da luz com as superfícies. Isso é particularmente notável em ambientes com elementos como água, vidro, plástico transparente ou materiais similares, onde o efeito Fresnel contribui significativamente para a sensação de profundidade, transparência e reflexão de luz.

Além disso, o efeito Fresnel pode ser ajustado para diferentes materiais e contextos dentro do jogo, garantindo que a aparência visual seja consistente com as expectativas estéticas e físicas dos jogadores. Essa técnica não apenas melhora a qualidade gráfica geral, mas também ajuda na imersão, criando uma experiência visual mais envolvente e detalhada nos jogos de computador modernos.

2.9.1. CONTEXTUALIZAÇÃO TEÓRICA

Para a realização deste shader seguimos um dos tutoriais recomendados pelo professor, presente no canal de Youtube: Benjamin Swee. Procuramos seguir o tutorial e não replicar copiosamente o seu código, que está aqui presente.[12]

Benjamin Swee é um Render Engineer especializado em Unit. Compartilha o seu conhecimento sobre shaders, renderização personalizada e desenvolvimento de efeitos visuais no Unity no Patreon (onde oferece acesso exclusivo a códigos de shader, explicações detalhadas e suporte à comunidade), no YouTube (onde compartilha tutoriais e insights sobre shaders e renderização no seu canal).

O shader de Benjamin Swee e o nosso "Unlit/Fresnel" são duas implementações distintas destinadas a adicionar efeitos visuais avançados em gráficos de jogos, cada um com suas próprias características e aplicabilidades.

Ambos os shaders compartilham uma estrutura básica comum, incluindo definição de propriedades como texturas principais e de distorção, controle de intensidade para efeitos como Fresnel e brilho externo/interno, além da possibilidade de utilizar mapas normais para detalhamento adicional das superfícies.

No entanto, existem diferenças entre eles. O shader do tutorial é projetado para renderização opaca, o que sugere que é mais adequado para superfícies sólidas e não transparentes. Além disso implementa dois efeitos Fresnel (normal e invertido), calculando a intensidade com base na direção de visualização e distorcendo essa intensidade com uma textura de distorção para criar um efeito de glow dinâmico

Por outro lado, o shader "Unlit/Fresnel" é destinado à renderização transparente, sendo ideal para elementos que necessitam de transparência, como vidro ou efeitos de partículas. Para a sua realização focamo-nos em calcular e misturar os brilhos externo e interno com base na normal da superfície e na textura de distorção. Este shader também oferece a opção de utilizar mapas normais para detalhar as superfícies de forma mais realista, adaptando-se melhor a necessidades específicas de detalhamento em superfícies mais complexas.

Além das diferenças estruturais e de renderização, as escolhas de blend mode (modo de mistura de cores) nos fragment shaders também são distintas, influenciando como as cores finais são combinadas e apresentadas na cena renderizada.

2.9.2. DESCRIÇÃO TÉCNICA

O shader "Unlit/Fresnel" é um shader destinado a criar efeitos visuais sofisticados em objetos renderizados onde a transparência é necessária. Vamos detalhar cada parte do shader:

Properties: As propriedades definem os parâmetros ajustáveis que podem ser configurados no editor do Unity para controlar a aparência visual do shader. São elas:

- _BaseTex: Textura principal que será aplicada à superfície do objeto.
- _DistortionTex: Textura de distorção que afeta a aparência da superfície.
- _DistortionStrength: Intensidade da distorção aplicada pela textura de distorção.
- _OuterGlowColor: Cor do brilho externo, que é aplicado na borda externa da superfície.
- _OuterGlowPower: Intensidade do brilho externo.
- _OuterGlowExponent: Exponente que controla a curva de intensidade do brilho externo.
- _InnerGlowColor: Cor do brilho interno, aplicado na borda interna da superfície.
- _InnerGlowPower: Intensidade do brilho interno.
- _InnerGlowExponent: Exponente que controla a curva de intensidade do brilho interno.
- _UseNormalMap: Toggle para ativar ou desativar o uso de mapa normal.
- _NormalMap: Textura que contém informações de mapa normal para detalhar a superfície.

SubShader: Define como o objeto será renderizado, incluindo tags específicas que influenciam o comportamento do renderizador ("Transparent", que define a fila de renderização como transparente, indicando que este shader é adequado para objetos com transparência, LOD (100) e o modo de mistura de cores (Blend SrcAlpha One).

Pass: etapa de renderização do shader, incluindo os shaders de vértice e fragmento que serão executados:

Vertex Shader (vert)

- appdata: Estrutura que define os dados de entrada para o shader de vértice, incluindo posição (vertex), coordenadas de textura (uv), normal (normal) e tangente (tangent).
- v2f: Estrutura que define os dados de saída do shader de vértice para o shader de fragmento, incluindo coordenadas de textura (uv), posição (vertex), normal (normal), direção da visão (viewDir), tangente (tangent) e bitangente (bitangent).

- Transformações: Converte a posição do vértice de espaço de objeto para espaço de recorte, normal de espaço de objeto para normal de espaço de mundo, e calcula a direção da visão no espaço de mundo.

Fragment Shader (frag)

- v2f: Recebe como entrada a estrutura definida pelo shader de vértice, contendo dados necessários para calcular a cor final.
- Cálculo de Distorção: Utiliza a textura de distorção para modificar a aparência da superfície com base nas coordenadas de textura.
- Cálculo da Normal Final: Se o mapa normal está ativado (_UseNormalMap), descompacta o mapa normal e calcula a normal final usando a tangente, bitangente e normal.
- Cálculo de Brilho Externo e Interno: Calcula a quantidade de brilho externo e interno com base na normal final e na direção da visão. A intensidade é influenciada pela distorção e controlada pelos parâmetros de força de distorção e exponenciais de brilho.
- Cor Final: Combina os brilhos externo e interno para formar a cor final que será renderizada.

```
// Calculate the outer glow amount
float outerGlowAmount = 1 - max(0, dot(finalNormal, i.viewDir));
outerGlowAmount *= distortionValue * _DistortionStrength;
outerGlowAmount = pow(outerGlowAmount, _OuterGlowExponent) * _OuterGlowPower;
float3 outerGlowColor = outerGlowAmount * _OuterGlowColor;

// Calculate the inner glow amount
float innerGlowAmount = max(0, dot(finalNormal, i.viewDir));
innerGlowAmount *= distortionValue * _DistortionStrength;
innerGlowAmount = pow(innerGlowAmount, _InnerGlowExponent) * _InnerGlowPower;
float3 innerGlowColor = innerGlowAmount * _InnerGlowColor;

// Calculate the final color
float3 finalColor = outerGlowColor + innerGlowColor;
return fixed4(finalColor, 1); // Return the final color
```

Figura 23 - outerGlowAmount/innerGlowAmount/finalColor - funções no Fresnel.shader.

2.9.3. IDEIAS DE APERFEIÇOAMENTO

Relativamente ao que poderíamos melhorar na implementação deste shader; podemos explorar uma determinada área: a do realismo. Assim, e sendo este um efeito bastante utópico, seria interessar configurar features ao mesmo que lhe configurassem mais realismo.

Para otimizar o shader "Unlit/Fresnel" e elevar seus efeitos visuais, é crucial explorar duas áreas principais. Primeiramente, a adição de texturas processuais dinâmicas pode trazer uma camada extra de realismo e detalhamento às superfícies renderizadas. Essas texturas podem ser programadas para reagir ao tempo e às interações do jogador, criando padrões complexos e variáveis que são ideais para simular desde água em movimento até materiais metálicos reflexivos. Em segundo lugar, a implementação de reflexões e transparências dinâmicas é essencial para aumentar a imersão visual do ambiente do jogo. Esses efeitos, quando ajustados para responder de forma realista à iluminação e à posição da câmara, podem transformar materiais como vidro, plásticos transparentes e metais polidos, conferindo-lhes profundidade e autenticidade. Essas melhorias não apenas ampliam as possibilidades estéticas do shader, mas também enriquecem a experiência dos jogadores ao proporcionar ambientes virtuais mais ricos e envolventes.

2.10. VERTEX/FRAGMENT SHADER - HOLOGRAM

Figura 24 - Hologram - Logo.

Um shader de holograma, em termos de conceito visual, é uma ferramenta poderosa na criação de efeitos visuais que simulam a aparência de projeções tridimensionais luminosas e etéreas. Esse tipo de shader utiliza técnicas avançadas de renderização para capturar as características distintivas dos hologramas, que são amplamente reconhecidos por sua transparência, brilho, e capacidade de flutuar no ar sem um suporte físico visível.

A combinação harmoniosa dos seus componentes permite não apenas criar hologramas visualmente convincentes, mas também adicionar uma camada de imersão e realismo aos ambientes digitais em que são utilizados. Eles são fundamentais em aplicações de entretenimento, como jogos, filmes e realidade aumentada, onde a criação de efeitos visuais futuristas e impressionantes é essencial para engajar o público e proporcionar experiências visuais únicas e memoráveis.

Por todas estas razões incluímos este shader no trabalho, mesmo correndo o risco de repetirmos procedimentos de construção. Achamos que devíamos encarar o desafio como um todo, e o tema central do projeto merecia que incluíssemos nele uma ferramenta deste género.

2.10.1. CONTEXTUALIZAÇÃO TEÓRICA

Esta ferramenta foi, na verdade, abordada em contexto de aula, tendo sido inclusivamente realizado um surface shader denominado de rimLight, e aplicado a um carro. Foi por aí que começamos a realização do nosso shader. Podemos, no entanto, dizer que houve uma boa curva evolutiva neste shader em específico, fruto dos conhecimentos que temos agora e na altura não tínhamos.

Figura 25 - Hologram - variações implementadas em aula.

Neste capítulo vamos tentar fazer um paralelismo entre o shader que efetuamos e o shader apresentado de seguida (apenas os pontos fundamentais e realizado por uma aluna que frequenta a disciplina, que por razões confidencialidade não diremos o nome (mais especificamente no dia 15/03/2024). O objetivo passa por apresentar as semelhanças entre ambos, que cimentaram as nossas decisões iniciais no processo de execução do nosso Hologram; referindo em seguida o que os diferenciam, para poder também dar ênfase à nossa evolução na elaboração de conceitos desta disciplina.

```
Shader "Custom/Hologram"
  Properties
{
    _BumpMap ("Normal Map", 2D) = "bump" {}
    _RimColor ("Rim Color", Color) = (0, 1, 0, 1)
    _RimPower ("Rim Power", Range (0, 10)) = 3
    _LineThickness ("Line Thickness", Range (0, 60)) = 30
    _CountOfLines ("Line Counts", int) = 3
  }
  SubShader
  {Tags {“RenderType"="Transparent" "Queue" = "Transparent"}
    #pragma surface surf noLight noambient alpha:fade
    #pragma target 3.0

void surf (Input IN, inout SurfaceOutput o)
    {
      o.Normal = UnpackNormal(tex2D(_BumpMap, IN.uv_BumpMap));
      o.Emission = _RimColor;
      float rim = saturate(dot(o.Normal, IN.viewDir));
      rim = pow(1 - rim, _RimPower) + pow(frac(IN.worldPos.g * _CountOfLines -
_Time.y), _LineThickness);
      o.Alpha = rim;
    }
    float4 LightingnoLight(SurfaceOutput s, float3 lightDir, float atten)
    {
      return float4(0, 0, 0, s.Alpha);
    }
```

Os shaders são duas abordagens distintas para criar efeitos visuais de hologramas em ambientes gráficos. Ambos os shaders compartilham o objetivo de simular hologramas, mas diferem em como alcançam esse resultado visualmente.

O nosso shader é projetado como um shader não iluminado, focando-se em criar um efeito holográfico estilizado e uniforme. Utiliza propriedades como _TintColor e _RimColor para ajustar a cor e o brilho do holograma. Além disso, incorpora um efeito de "glitch" dinâmico com base no tempo, proporcionando uma distorção sutil que adiciona uma sensação de movimento e instabilidade ao holograma.

Por outro lado, o shader realizado na aula adota uma abordagem diferente, permitindo maior controle sobre a interação do holograma com a luz na cena. Ele utiliza uma função personalizada de iluminação (LightingnoLight) para determinar como a luz afeta o holograma, proporcionando uma aparência mais realista e detalhada. Este shader inclui propriedades como _RimColor, _RimPower, _LineThickness e _CountOfLines, que permitem ajustar a intensidade do efeito de borda, espessura das linhas e outros aspectos visuais do holograma. É mais adequado para situações onde se deseja uma representação mais precisa e controlada do holograma, com capacidade de personalização detalhada das suas características visuais.

2.10.2. DESCRIÇÃO TÉCNICA

Neste capítulo vamos explorar cada segmento deste shader detalhadamente:

Properties:

- _RimColor ("Highlight Color", Color): Define a cor do destaque que aparece ao redor do holograma.
- _TintColor("Base Color", Color): Define a cor base do holograma.
- _GlitchFrequency("Glitch Frequency", Range(0.01,3.0)): Controla a frequência do efeito de glitch no holograma.
- _WorldScale("Stripe count", Range(1,200)): Define o número de listras no holograma, afetando o padrão visual.
- _HologramSpeed("Hologram Speed", Range(0.1, 5.0)): Controla a velocidade de animação do holograma.

SubShader:

- **Tags:** Define as tags que especificam a fila de renderização como transparente, ignorando projetores e especificando o tipo de renderização como transparente. Também especifica o tipo de visualização para a esfera no editor Unity.

- **Blend:** Especifica o modo de mistura para transparência.
- **ColorMask**: Define quais componentes de cor serão afetados pela renderização.
- **Cull**: Define a face a ser removida durante o culling, neste caso, as faces de trás.

Pass (bloco que contém os shaders vertex e fragment)

Vertex Shader (vert): Recebe os vértices do objeto e transforma para coordenadas de recorte. Também calcula o efeito de glitch baseado no tempo, deslocando ligeiramente os vértices ao longo do eixo xz.

Fragment Shader (frag): Calcula a cor final do pixel que será renderizado, bem como o ângulo e o raio da posição do pixel em relação ao mundo, criando um efeito de espiral no holograma. Além disso, calcula a direção do olhar e a luz de borda (rim lighting) para simular o brilho ao redor do holograma. A cor final do holograma é combinada com os efeitos de espiral e luz de borda para criar o efeito holográfico desejado.

2.10.3. IDEIAS DE APERFEIÇOAMENTO

Para aprimorar o shader "Unlit/Hologram", existem algumas melhorias que podem ser implementadas para enriquecer o efeito holográfico e a experiência visual geral. Primeiramente, ajustes nos parâmetros existentes podem proporcionar maior controle sobre o holograma: refinando a frequência e a intensidade do efeito de glitch para criar variações mais sutis ou dramáticas conforme necessário.

Além disso, explorar opções de animação mais complexas para o movimento das listras holográficas pode adicionar dinamismo ao efeito, como variações na velocidade de animação ao longo do tempo para simular interações mais realistas com o ambiente virtual. (tal como vimos fazer em aula, que decidimos não acrescentar por forma a apresentar uma solução mais direta).

Outro aspeto seria expandir as opções de personalização visual, como permitir ajustes na forma e no padrão das listras holográficas, oferecendo aos utilizadores mais flexibilidade na criação de hologramas distintos e personalizados para diferentes contextos e estilos visuais. Essas melhorias não apenas refinariam a estética do holograma, mas também aprimorariam sua integração em ambientes interativos, proporcionando uma experiência mais imersiva e esteticamente rica para os usuários.

2.11. VERTEX/FRAGMENT SHADER - SECTION - REVEAL

Figura 26 - Section-Reveal - Logo.

O shader "Unlit/Section-Reveal" é um shader sem iluminação projetado para criar um efeito visual específico em gráficos 3D, que se enquadra na categoria de shaders de "revelação de seção", que são bastante utilizados para revelar ou ocultar partes específicas de uma cena 3D com base em determinadas condições.

O principal efeito produzido por este shader é a capacidade de revelar ou ocultar seções específicas de um objeto ou cena 3D. Ele faz isso descartando pixéis tendo por base a posição no mundo (mais especificamente o valor de altura) em relação a um valor de limite. Desta forma, os pixels acima do limite são descartados, criando efetivamente um efeito de "corte" onde apenas as partes do objeto acima de uma determinada altura são visíveis. Esse limite pode ser ajustado dinamicamente para revelar diferentes seções do objeto. Para além disso adicionamos como feature um controlador da largura da extremidade, o que acrescenta suavidade à transição.

A aplicação deste shader em videojogos é imensa, podendo ser usado para revelar o interior de um prédio enquanto mantém as paredes externas intactas; ou em jogos nos quais a exploração e a descoberta são essenciais (por exemplo, jogos de exploração de masmorras, salas de fuga), podendo ser usado para ocultar passagens secretas, alçapões ou áreas ocultas.

2.11.1. CONTEXTUALIZAÇÃO TEÓRICA

A inspiração para a realização deste shader foi o shader de **corte da melancia** apresentado e realizado em aula pelo docente e pelos alunos. Assumimos o risco/audácia de apresentar este shader, mesmo sabendo que relativamente ao que foi efetuado nas aulas não há muita diferença, porque queríamos tentar aperfeiçoar o que foi aprendido em aula. Entendemos que essa decisão pode e deve fazer parte do processo evolutivo do aluno, quer como consolidação da matéria dada; quer como uma etapa de desafio pessoal que é adequada ao tema.

Posto isto, podemos dizer que o objetivo foi conseguido pelas seguintes razões:

- **Lógica de Revelação:** O "Unlit/Section-Reveal" utiliza uma lógica mais robusta para revelar seções. Ele descarta explicitamente pixels acima do limite de altura (_Val), criando um efeito de corte preciso. O shader elaborado em aula modifica a posição do vértice para limitar a altura, mas não descarta pixels de maneira tão direta.
- **Parâmetros Personalizáveis:** O nosso shader oferece parâmetros como _Color1, _Color2 e _EdgeWidth, permitindo maior controle sobre a aparência e o comportamento do efeito. O shader fornecido tem apenas um parâmetro (_Slider), que não é tão versátil.
- **Clareza e Intenção:** O "Unlit/Section-Reveal" é mais claro em sua intenção e funcionalidade, tornando-o mais fácil de usar e entender. O shader fornecido pode exigir mais contexto ou ajustes para ser totalmente compreendido.

Em resumo, achamos que conseguimos uma evolução porque oferecemos com o nosso shader uma solução mais especializada e completa para o efeito de revelação de seções em comparação com o shader fornecido.

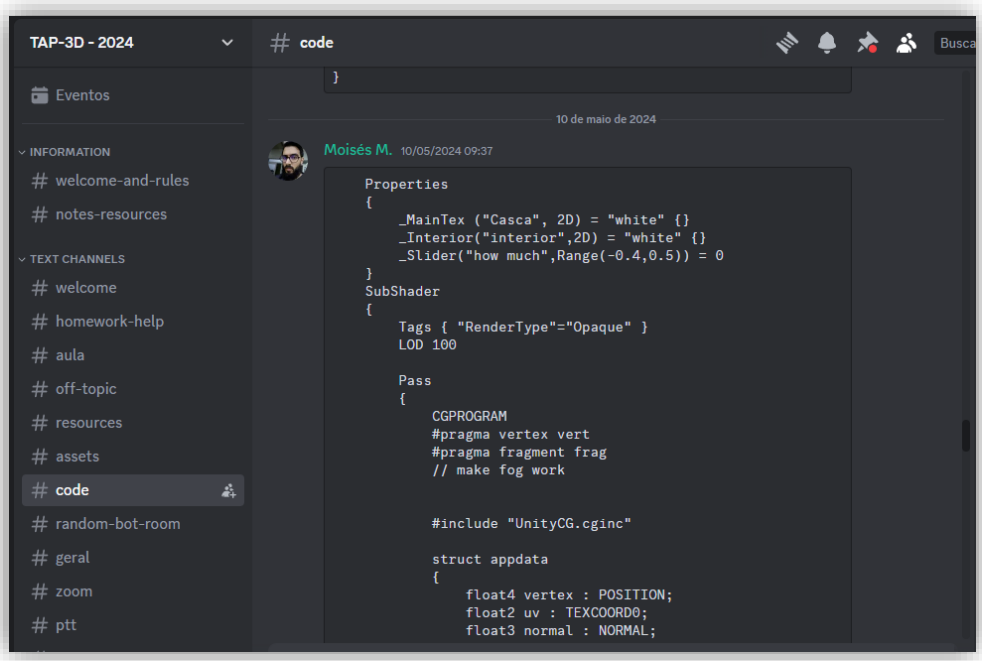


Figura 27 - Shader "melancia" feito em sala de aula.

2.11.2. DESCRIÇÃO TÉCNICA

O shader “Unlit/Section-Reveal” pode ser classificado como um shader personalizado ou shader de superfície personalizado. Passemos a analisá-lo para depois justificar esta nomenclatura.

Properties: com a _MainTex, a textura externa (textura principal) usada para a superfície do objeto; _Color2, que define a cor da seção (cor da parte revelada); _EdgeWidth onde podemos controlar a largura da extremodade para suavizar a transição entre as seções; e _Val, o valor de altura usado como limite para a revelação.

SubShader: Define o início do subshader. Utiliza a tag "Queue"="Geometry" para especificar a ordem de renderização.

//Pass 1 (Surface): Define a estrutura de entrada Input com campos worldPos e uv_MainTex; descartando pixels cuja posição no mundo (worldPos.y) seja maior que o valor de _Val; mostra a textura _MainTex e define a cor do albedo.

//Pass 2 (Section Reveal): Aqui definimos a estrutura v2f com campos pos e worldPos. Em seguida (e se necessário) ajusta-se a posição do vértice para limitar a altura, e descartam-se os pixéis acima do limite de altura (_Val). No final, retorna a cor _Color2 para a seção revelada.

//Pass 3 (Edge Width): Define-se outra estrutura v2f semelhante à Pass 2, onde se multiplica a posição do vértice pela _EdgeWidth para controlar a largura da edge. Por fim, uma vez mais descartam-se os pixeéis acima do limite de altura (_Val), e retorna a cor _Color2 para a seção revelada.

//Pass 4 (Surface with Edge Width): Define a estrutura de entrada Input com campo worldPos. Em seguida ajusta a posição do vértice multiplicando pela _EdgeWidth. Descarta pixéis acima do limite de altura (_Val), e define a cor do albedo como _Color1 para a seção não revelada.

Em resumo, o shader "Unlit/Section-Reveal" combina várias técnicas para criar um efeito de revelação de seções com base na altura. Ele controla a largura da borda, descarta pixels acima do limite e permite personalização de cores para as seções reveladas e não reveladas. Por esse motivo pode categorizar-se como um shader personalizado que combina as funções **surf, vert e frag** para criar um efeito específico de revelação de seções com base na altura. Não se enquadra nas categorias padrão de shaders (como "unlit", "standard" ou "transparent"), mas serve o seu propósito final de forma eficaz.

2.11.3. IDEIAS DE APERFEIÇOAMENTO

Apesar de considerarmos ter conseguido fazer um aperfeiçoamento do shader desenvolvido em sala de aula, concluímos (tal como em todos os shaders) que existe sempre uma possibilidade de melhorar e de acrescentar funcionalidades ao trabalho aqui desenvolvido.

Em termos de otimização de Desempenho, o nosso shader possui várias passagens e achamos que numa futura abordagem pode ser possível otimizar o desempenho reduzindo o número de passagens. Consideraríamos consolidar algumas das passagens, se possível, para simplificar o código e melhorar a eficiência.

O shader descarta pixels abruptamente acima do limite de altura. Para um efeito mais suave, é possível tentar usar uma função de interpolação para suavizar a transição entre as seções reveladas e não reveladas. Outra melhoria poderia ser em vez de usar apenas uma textura externa (_MainTex), considerar adicionar suporte para múltiplas texturas,

o que permitiria que diferentes partes do objeto tivessem texturas distintas, criando variação visual, dando um efeito de corte animado por uma sprite.

Outra feature que poderia acrescentar funcionalidade seria a de usar um gradiente ao invés de usar cores sólidas (_Color1 ou _Color2 para a seção revelada ou não revelada.

3. CONCLUSÕES

Figura 28 – Monstros a pensarem.

Concluir o relatório "House of Horrors", que abrange a criação e implementação de vários shaders, é essencial para refletir sobre o processo, os resultados alcançados e as lições aprendidas ao longo dessa jornada. Durante este relatório, exploramos e analisamos profundamente diferentes técnicas de shader, cada uma contribuindo de maneira única para o produto final. Este capítulo procura sintetizar as experiências, destacar os pontos principais e oferecer insights sobre o que foi alcançado e o que poderia ter sido melhorado.

Ao longo deste trabalho, dedicamo-nos à criação de uma variedade de shaders, desde surface shaders até técnicas mais avançadas, com o objetivo de desenvolver um ambiente imersivo e visualmente impactante para o projeto "House of Horrors". Cada tipo de shader apresentou desafios únicos, exigindo um entendimento profundo das suas funcionalidades e aplicações específicas. A metodologia adotada envolveu uma combinação de pesquisa bibliográfica, tutoriais práticos, adaptação de código GLSL para HLSL e consultas a materiais de aula, além da utilização de técnicas modernas como inteligência artificial para otimização e criação.

Inicialmente, começamos com surface shaders, utilizando as facilidades proporcionadas pelo Unity para criar e personalizar texturas e efeitos visuais. Aprendemos a importância de ajustar os parâmetros corretamente para alcançar o resultado desejado, equilibrando entre desempenho e qualidade visual. A implementação de grab pass permitiu acesso aos dados de renderização de um plano, expandindo para as possibilidades de efeitos pós-processamento.

Em seguida, exploramos shaders triplanar e geometria, ampliando nossa capacidade de mapeamento de texturas e manipulação de geometria em diferentes contextos. Estas técnicas foram cruciais para a criação do ambiente que desempenha um papel fundamental na estética do "House of Horrors".

A abordagem em shaders de pós-processamento permitiu a aplicação de filtros e efeitos visuais, refinando a aparência final da cena e proporcionando uma atmosfera coesa e envolvente. Estes shaders não apenas adicionaram camadas de profundidade e realismo à experiência, mas também demonstraram a versatilidade do Unity na criação de ambientes interativos e visualmente ricos.

Os shaders de vertex/fragment foram um marco significativo, envolvendo a manipulação direta de vértices e fragmentos para alcançar efeitos específicos, como iluminação avançada e sombreamento personalizado. Aprendemos a importância da matemática e da lógica por trás desses shaders, garantindo que cada vértice e fragmento contribuísse de forma eficaz para a representação visual desejada.

Tivemos ainda para apresentar o desenvolvimento de um shader personalizado, integrando funções surf, vert e frag, foi um dos pontos mais estranhos do projeto. Esta abordagem permitiu uma personalização completa do comportamento do shader. Exploramos a fundo o pipeline de renderização, desde a entrada dos vértices até a saída final do fragmento, garantindo controle total sobre cada aspeto visual do ambiente virtual.

No entanto, reconhecemos que há aspetos que poderiam ser melhorados ou explorados mais profundamente. A ausência da implementação de um shader de raymarching, solicitada pelo docente, destaca a complexidade contínua e o potencial de aprendizado nesse campo. Várias razões, como limitações de tempo e desafios técnicos, impediram a inclusão deste shader específico, destacando a natureza iterativa e adaptativa do processo criativo envolvido no desenvolvimento de shaders.

Concluindo, estamos extremamente satisfeitos com o produto final obtido. "House of Horrors" não apenas demonstra nossa habilidade em utilizar shaders avançados para criar ambientes imersivos, mas também reflete nosso compromisso em aprender e aplicar novas técnicas no campo de desenvolvimento visual. Este relatório não é apenas um registro de nossas conquistas, mas também um testemunho do potencial ilimitado de inovação e criatividade que os shaders oferecem no desenvolvimento de experiências interativas e envolventes.

4. BIBLIOGRAFIA

Figura 29 - Monstros na biblioteca.

1 https://www.ronja-tutorials.com/

2 https://kylehalladay.com/blog/tutorial/2015/11/10/Dissolve-Shader-Redux.html

3 https://catlikecoding.com/

4 https://www.khanacademy.org/computing/computer-programming/programming-natural-simulations/programming-noise/a/perlin-noise

5 https://copilot.microsoft.com/

6 https://openai.com/index/chatgpt/

7 https://www.youtube.com/@NedMakesGames

8 https://docs.unity3d.com/Manual/SL-BuiltinIncludes.html

9 https://web.fe.up.pt/~mandrade/tvd/2006/trabalhos1-2006/TD-trab1-grupo02.pdf

[10] https://www.ime.unicamp.br/~jbflorindo/Teaching/2019/MT530/T4.pdf

[11] https://www.shadertoy.com/view/DtBXRz

[12] https://docs.unity3d.com/Manual/StandardShaderFresnel.html