Bernardo Coutinho
Bruno Mariz
Vinicius Ariel

Shaders de pós-processamento visual open-source para Unreal Engine 5

São Paulo, SP

2023

Bernardo Coutinho
Bruno Mariz
Vinicius Ariel

Shaders de pós-processamento visual open-source para Unreal Engine 5

Trabalho de conclusão de curso apresentado ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro.

Universidade de São Paulo – USP

Escola Politécnica

Departamento de Engenharia de Computação e Sistemas Digitais (PCS)

Orientador: Prof. Dr. Ricardo Nakamura

São Paulo, SP

2023

Catalogação-na-publicação

Coutinho, Bernardo
Shaders de pós-processamento visual open-source para Unreal Engine 5 / B. Coutinho, B. Teixeira, V. dos Santos -- São Paulo, 2023.
p.

Trabalho de Formatura - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Shaders 2.Game Engine 3.Computação Gráfica I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t. III.Teixeira, Bruno IV.dos Santos, Vinicius

Resumo

Neste trabalho apresentaremos a motivação, planejamento e desenvolvimento de um *Asset Pack* para a *game engine* Unreal Engine 5 que disponibiliza *Shaders* para a aplicação de efeitos visuais de pós-processamento em vídeo-games. Nosso produto final é gratuito e *open-source*, características que não estão presentes na maioria dos pacotes que disponibilizam esse tipo de conteúdo. Levando em consideração o contexto da indústria de jogos no Brasil, que é composta principalmente por pequenas empresas independentes e com poucos recursos, acreditamos que o pacote desenvolvido representa uma alternativa sólida para equipes que desejem experimentar efeitos visuais sem onerar o tempo de produção de seus artistas técnicos ou até mesmo para projetos que não têm um profissional especializado na produção de *Shaders*. Nesse sentido, preocupamo-nos em facilitar o acesso aos recursos desenvolvendo em uma *engine* de utilização gratuita e assegurando que a utilização dos efeitos visuais produzidos não afetaria negativamente o desempenho do jogo. Ao fim do projeto, os efeitos produzidos, juntamente com os requisitos de desempenho, foram validados utilizando "*Sample Games*", pequenos projetos disponibilizados gratuitamente pelos desenvolvedores da Unreal Engine.

Palavras-chave: Shaders. Game Engines. Computação Gráfica. Pós Processamento.

Abstract

In this work we present the motivation, planning and development of an Asset Pack for the Unreal Engine 5 that includes a set of Shaders that apply post-processing visual effects to video games. Our final product will be free and open-source, characteristics that are lacking in most packs that provide this kind of content. Considering the context of Brazilian game industry, which is made up mainly by small independent companies with little resources, we believe that the developed pack will become a solid alternative to teams that desire to experiment visual effects without overtaxing the production time of techincal artists or even to projects that do not have a specialized professional to develop Shaders. With that in mind, we facilitated the access to the resources by developing in an free-to-use engine and assuring that the visual effects created would not affect the game's performance negatively. At the end of the project, the Shaders developed and the performance requirements were validated utilizing "Sample Games", small and complete projects made freely available by the developers of Unreal Engine.

Keywords: Shaders. Game Engines. Computer graphics. Post Processing.

Sumário

1 Introdução

O motor de desenvolvimento Unreal Engine 5 (UE5) conta com várias ferramentas necessárias para a produção de jogos e experiências interativas em geral, fornecendo ao usuário um framework básico para a programação, animação, modelagem, design de som e renderização (EPIC GAMES, h). Nesse último tópico, a Unreal disponibiliza uma ferramenta chamada *Material Editor* (ver Figura 1) que permite, entre outras coisas, um sistema de programação visual através de grafos que possibilita a criação de *Shaders*, algoritmos que controlam como cada pixel será renderizado na tela e são preferencialmente executados na GPU (*Graphics Processing Unit*) (VRIES, 2014).

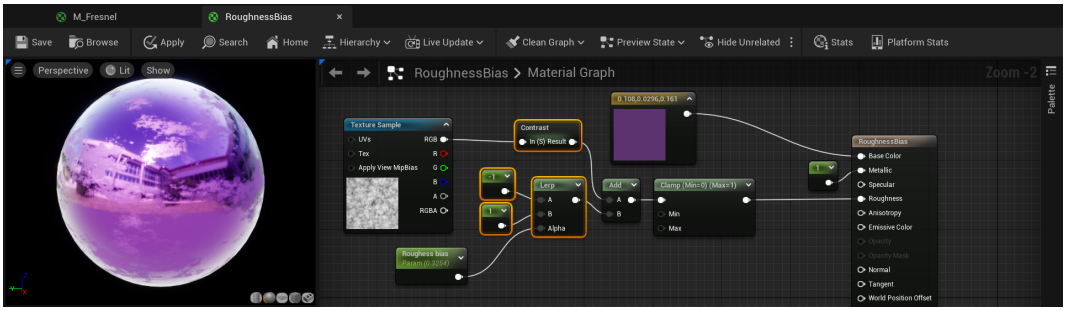


Figura 1 – Editor de *Materials* da UE 5.1 (EPIC GAMES, d)

A maioria dos jogos atualmente utiliza alguma forma de *Shader* de pós-processamento (LAUKKANEN, 2018), sendo importantes não só para melhorar a aparência de uma cena como também servir de elemento de narrativa visual, estabelecendo o estilo e o clima do jogo. Alguns exemplos são o *Shader* de Dithering em Return of the Obra Dinn (ver Figura 2) (ILETT, 2020), o efeito de televisão de tubo em Loop Hero e o uso de *Shaders* de simplificação em Dead Cells que permitem fazer objetos 3D pareçam artes feitas em 2D (VASSEUR, 2018). Na própria versão base da UE5, estão inclusos outros efeitos de pós-processamento de uso comum para obter algum efeito astético específico como *Bloom*, *Depth of Field*, *Chromatic Aberration* e *Motion Blur* (EPIC GAMES, f).

1.1 Motivação

Utilizando o Editor de *Materials* da Unreal Engine, é possível que um artista técnico crie *Shaders* de pós-processamento que aplicam efeitos ao pixels da cena (salvos no buffer de renderização) antes que sejam de fato exibidos na tela. Por conta disso, existem vários *Asset Packs* voltados à UE5 que implementam *Shaders* para serem utilizados sem a necessidade de programação, de maneira prática. *Asset Packs* desse gênero, apesar de não serem usualmente utilizados em uma implementação final de um projeto grande,

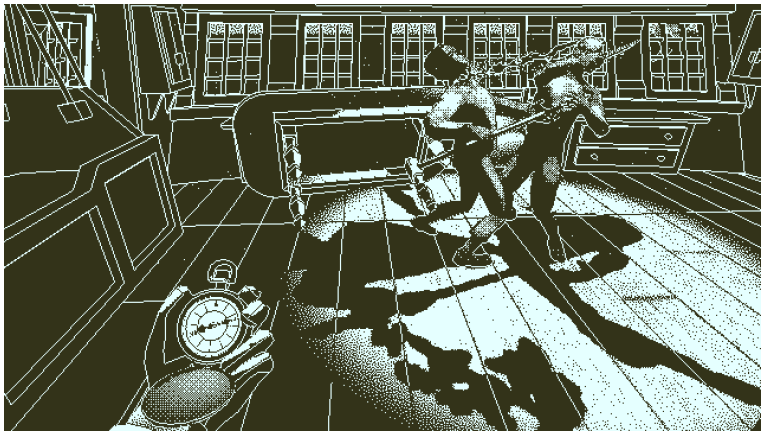


Figura 2 – Dithering *Shader* em Return of the Obra Dinn (POPE, 2019)

podem ajudar desenvolvedores no início de uma produção a testar vários tipos de efeitos de pós-processamento diferentes sem onerar os artistas técnicos da equipe em desenvolver vários *Shaders* que poderiam ser posteriormente descartados.

Além disso, também consideramos o contexto de projetos menores, em que as equipes são menores e a exigência de recursos computacionais para o desenvolvimento é relativamente pequena. Nesse caso, a produção de recursos mais complexos de arte técnica pode estar além do escopo dos desenvolvedores. Portanto, Pacotes que disponibilizem esses recursos podem ser mantidos na versão finalizada do projeto.

No entanto, a grande maioria dos pacotes de pós-processamento são pagos (ENGELMANN, 2018; ESENIN, 2022; EPIC GAMES, i), com preços não ajustados à moeda brasileira. Os poucos pacotes gratuitos que estão disponíveis no Marketplace oficial da Unreal limitam-se a um ou dois *Shaders* diferentes. Assim, percebe-se a falta de uma alternativa *open-source* e gratuita para dar acessibilidade às vantagens que um conjunto de efeitos de pós-processamento fornece a um projeto. Na Figura 3, temos um exemplo de um dos *Asset Packs* pagos disponíveis no Marketplace da Unreal Engine.

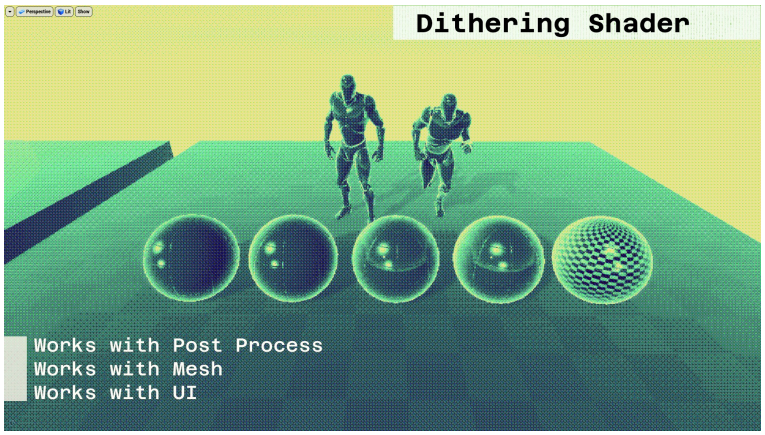


Figura 3 – *Asset Pack* que disponibiliza efeito de Dithering *Shader*, disponível no Marketplace da Unreal (ESENIN, 2022)

1.2 Objetivos

Este trabalho tem como objetivo a criação de um *Asset Pack* para a Unreal Engine 5 que permita a rápida iteração por diferentes efeitos visuais obtidos com *Shaders* de pós-processamento. Para tanto, foi implementado um conjunto de efeitos com parâmetros customizáveis e uma integração coerente com o restante da *engine*. O principal diferencial do projeto é ser uma alternativa *open-source* e grátis aos pacotes já existentes na loja da Unreal Engine, que estabelecem uma barreira monetária indesejada para pequenos desenvolvedores.

Ao final do projeto, pretendemos ter um pacote que permita aos desenvolvedores de jogos testar diferentes efeitos visuais em seus jogos de maneira prática, modular e rápida. Assim, nosso trabalho tem foco na usabilidade dentro de projetos pequenos ou em estados iniciais (desde a pré-produção até o Alpha), tendo como benefício primordial evitar que artistas técnicos (responsáveis pela criação de materiais, *Shaders* e efeitos visuais) gastem tempo de produção com a criação de *assets* que podem ser posteriormente descartados.

A utilização do produto desenvolvida será facilitada através da disponibilização do *Asset Pack* em um repositório no Github. Dessa forma, qualquer desenvolvedor poderá integrar os efeitos produzidos ao seu projeto apenas baixando a pasta de materiais disponibilizada, de maneira mais simples e intuitiva.

1.3 Justificativa

Com esse trabalho, buscamos criar uma solução acessível e simples para projetos de diversos portes que desejam experimentar de maneira simples diferentes efeitos visuais. Fornecer uma alternativa gratuita e com uma certa variedade de opções de efeitos visuais facilitará parte do processo de desenvolvimento de jogos com menos recursos, principalmente em países como o Brasil, onde o cenário de games ainda está em seus estágios iniciais, com um número pequeno de empresas atuando na área (ALVES, 2008). Por este motivo, o pacote foi desenvolvido na plataforma da Unreal Engine, uma das ferramentas gratuitas para desenvolvimento mais populares para criação de jogos, que já possui uma grande base de usuários, incluindo empresas de grande porte e pequenos desenvolvedores, com várias extensões da *engine* produzidas independentemente ao longo dos anos.

Como a indústria brasileira de jogos é majoritariamente composta empresas com pequenos times de desenvolvimento (FORTIM, 2022), a especialização de profissionais em uma área específica do desenvolvimento não uma situação comum. Ao contrário, os profissionais se generalizam ao terem que atender demandas diversas dentro de um projeto (ZOETHOUT; JAGER; MOLLEMAN, 2008). Portanto, o tempo gasto na produção de cada recurso é escasso e pode impactar significantemente o término de projetos nos prazos

estipulados. Dessa forma, propomos que, se uma quantidade maior de recursos gratuitos estiver disponível para agilizar o processo de iterar por diferentes possibilidades nos campos de desenvolvimento de jogos, o risco dos meses iniciais de produção diminuirá, ao passo que a acessibilidade aos conhecimentos e ferramentas necessárias para a produção de experiência interativas aumentará. Nosso projeto visa disponibilizar parte desses recursos em uma área específica: efeitos visuais de pós-processamento.

1.4 Organização do Trabalho

Neste documento, primeiramente são apresentados no capítulo 2 os aspectos conceituais que permeiam o projeto para a seguir descrever a metodologia utilizada para a pesquisa e trabalho de implementação no capítulo 3. No capítulo 4, são enumerados os requisitos elencados para o projeto no que diz respeito a quantidade de efeitos diferentes a serem disponibilizados, bem como o impacto de desempenho esperado ao utilizar os *Shaders* de pós-processamento. Também são descritos em mais detalhes os efeitos visuais esperados de cada um dos *Shaders* propostos.

Finalmente, no capítulo 5, são apresentadas em detalhe as soluções desenvolvidas, incluindo os algoritmos utilizados, e os resultados dos testes de desempenho feitos com os *Sample Games* disponibilizados pela Epic Games. As conclusões e considerações finais no capítulo 6 contam com uma análise a respeito do quanto o projeto alcançou os objetivos apresentados e sugestões de possíveis melhorias que podem ser feitas para melhorar a ferramenta desenvolvida.

2 Aspectos Conceituais

2.1 *Shaders* Programáveis

Em cenas renderizadas com técnicas de computação gráfica, é possível representar objetos em 3D através conjuntos de vértices que delimitam a malha da superfície do objeto num espaço tridimensional. Durante o processo de renderização, a iluminação das faces de cada objeto é determinada pelos *Shaders* (ANGEL; SHREINER, 2011). Os "*Vertex Shaders*" podem manipular informações referentes aos vértices, como sua cor e posição. Isso pode ser útil, por exemplo, para criação de terrenos, renderização de representações de água, efeitos de distorção de objetos ou para implementar *metaballs* (isosuperfícies que podem ser renderizadas em tela). Outro tipo comum de *Shader*, que também é o foco deste projeto, é o "*Pixel Shader*", que atua nos fragmentos gerados após a rasterização de uma geometria. Os "*Pixel Shaders*" podem ser programados para gerar diferentes efeitos visuais de pós-processamento, incluindo alguns dos efeitos propostos para este trabalho.

2.2 Pós-processamento

Pós-processamento é uma técnica de *Shader* para aplicação de efeitos visuais sobre os pixels já pré-renderizados de uma imagem ou cena 3D antes de serem de fato exibidas na tela. Tratando-se de arquiteturas mais modernas, as informações desses pixels estão geralmente armazenadas em *buffers* especiais de GPUs, chamados *frame buffers*. Na Figura 5, uma arquitetura simples de *frame buffers* é representada, composta por um *backbuffer* e um *front buffer*. Nesse caso, a representação no *display* alterna entre os dois para otimizar o tempo de troca de *frames*. É sobre esses *buffers* que os efeitos de pós-processamento são aplicados.

Efeitos de pós-processamento definem o visual de uma cena através de uma combinação de propriedades que afetam coloração, mapeamento de tons, luz entre outros (EPIC GAMES, f).

As descrições de cada um dos efeitos de pós-processamento propostos para este trabalho será dada no capítulo 4, na seção de requisitos funcionais.

2.3 Arte Técnica

O termo "Arte Técnica"refere-se, dentro do contexto da produção de jogos digitais, ao campo de desenvolvimento de recursos artísticos que utilizam, em algum grau, a

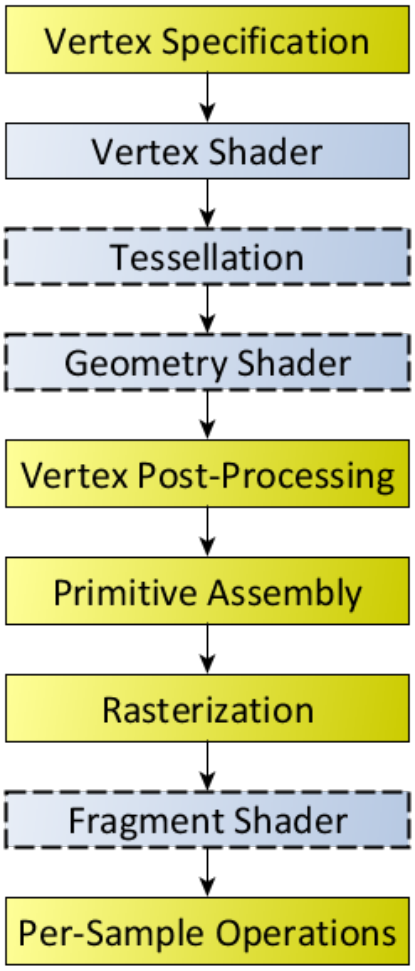


Figura 4 – Modelo de uma *Rendering Pipeline*. Os blocos em azul representam estágios onde *Shaders* programáveis são incluídos (KHRONOS GROUP, 2022). Os *Pixel Shaders* são também chamados de *Fragment Shaders*, e estão no final da *pipeline*.

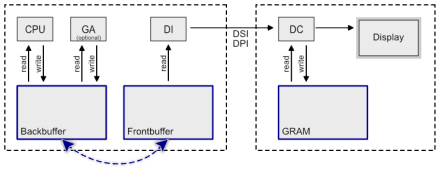


Figura 5 – Modelo que representa a parte final da *pipeline* de renderização. Os *frame buffers* são o *backbuffer* e o *frontbuffer*, que são alternados entre sí para otimizar a renderização no *display*.

programação direta desses recursos de maneira a controlar sua aparência e efeitos sobre outros elementos em cena.

Estão inclusas dentro do campo da Arte Técnica a produção de texturas, *Materials*, iluminações e *Shaders*.

2.4 *Materials* e Texturas

No contexto da computação gráfica, texturas são imagens bidimensionais que podem ser aplicadas às malhas poligonais que definem a superfície de um objeto 3D. Elas mapeiam uma cor a cada ponto da superfície do objeto (FOLEY, 1996). A imagem da Figura 6 representa como a imagem contendo cada uma das faces de um dado pode ser mapeada a um cubo, de maneira a representar o objeto desejado.

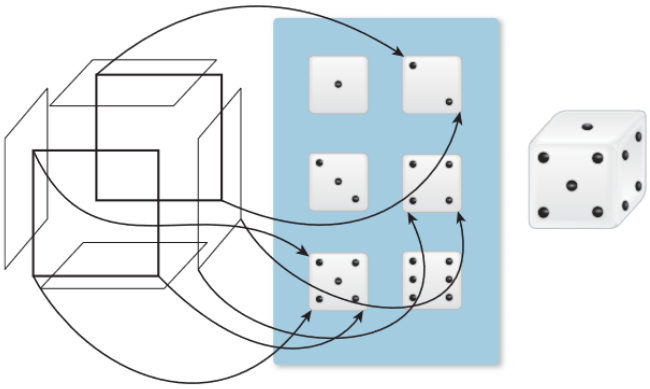


Figura 6 – A imagem bidimensional mapeia cores a cada parte da superfície do cubo (FOLEY, 1996).

Na Unreal Engine, os *Materials* são um tipo específico de *asset* que determina as propriedades do objeto ao qual é aplicado, como sua cor (geralmente mapeada através de texturas), refletividade, textura, opacidade, entre outros (EPIC GAMES, e). Dessa forma, os *Materials* são estruturas de dados que podem ser associados a objetos e englobal não apenas as texturas definidas (ou seja, as cores) como também como a luz interage com a superfície, influenciando na renderização. Os *Post Process Materials* da Unreal, no entanto, permitem usar boa parte da lógica de produção de *Materials* para a criação também de efeitos de pós-processamento na tela de renderização do projeto, controlando as propriedades da própria visualização.

Além disso, a Unreal também disponibiliza *assets* chamados de *Material Instances*, que são criados como objetos filhos dos *Materials*. Eles são versões em que as regiões do código de *Shader* que não são parametrizáveis são pré-computadas, de maneira a otimizar o tempo de renderização. A utilização de *Material Instances* é sempre recomendada, a

menos que se deseje sempre rodar o código do *Shader* por completo sempre que o mesmo for executado.

2.5 *Buffer* de Renderização

O *buffer* de renderização é uma região especial da memória da GPU que guarda um *array* de bytes que contém a informação do que deve ser renderizado na tela, baseado no primeiro processamento feito nos núcleos dessa GPU. Ou seja, o *buffer* de renderização contém informação guardada como uma única imagem, que poderá seguir diferentes formatos (KHRONOS GROUP, 2014) e que poderá ser renderizada posteriormente (PATHAK; CHOUDHARY, 2014).

Acesso a esse recurso é disponibilizado a programas através de APIs de computação gráfica como, por exemplo, o OpenGL. Efeitos de pós-processamento são aplicados sobre os dados da imagem armazenada no *buffer* de renderização.

2.6 Buffer de Distância

O *buffer* de distância é uma parte da memória na GPU que também contém um *array* de dados associados a cada um dos pixels da cena renderizada. No entando, ao invés de guardar informações a respeito das cores ,como é o caso do *buffer* de renderização, este armazena informações da distância de cada objeto em cena em relação ao ponto de observação.

Assim, a cada pixel é atribuído um valor de 0 a 1 (que, em representação de cores, vai de preto a branco) baseado na distância do ponto de vista e o objeto que esse pixel representa. Por exemplo, na Figura 7, podemos ver a representação dos valores desse *buffer* como uma imagem. Pixels de objetos mais próximos do ponto de vista têm "cores"mais próximas de zero (que são representados como cores mais escuras, próximas do preto) enquanto pixels pertencentes a objetos mais distantes têm "cores"mais próximas de 1 (ou seja, mais próximas do branco).

2.7 *Game Engine*

O termo *Game Engine* surgiu nos anos 90 e começou a se popularizar devido à disponibilização de *toolkits* gratuitos que permitiam que qualquer pessoa editasse jogos já existentes. Esses *toolkits*, publicados pelos próprios desenvolvedores, resultaram no desenvolvimento da comunidade de editores (referenciada de maneira genérica como "*mod community*") e serviu como marco inicial do processo de acessibilidade de tecnologias para criação de jogos de computador.

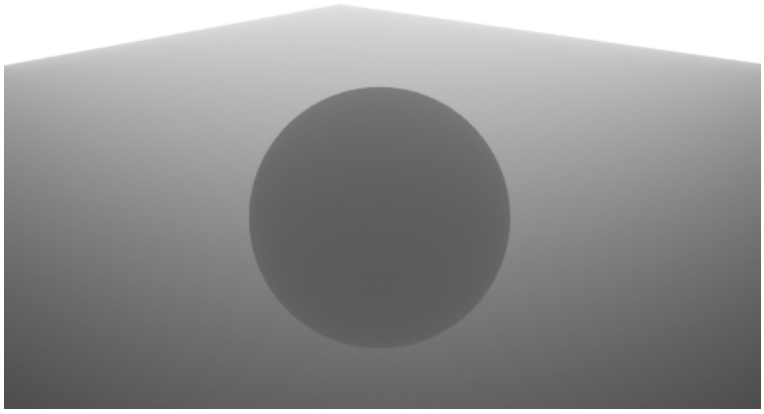


Figura 7 – Visualização dos valores atribuídos a cada pixel em cena pelo *buffer* de distância, na forma de uma imagem.

Uma *Game Engine* (Motor de jogo) é uma ferramenta de *software* que possui diferentes módulos que permitem a produção de um jogo. Esses módulos variam em cada motor, mas geralmente incluem: **implementação de lógica de jogo** (seja através de uma linguagem de *scripting* ou extensão de uma linguagem de programação já existente), **simulação física do mundo do jogo** (que pode ser editada), **recursos de manipulação de áudio**, **sistema de animação**, **gerenciamento de entradas e saídas**, **recursos de comunicação com uma rede**, **renderização da cena criada**, além de uma interface visual que permita interagir com todos esses subsistemas (GREGORY, 2018). Na imagem da Figura 8, Gregory (2018) apresenta um modelo de arquitetura comum de uma *Game Engine*. Nela, é possível perceber os módulos fundamentais, de mais baixo nível e mais próximos do sistema operacional e os módulos que dizem respeito a implementação de um jogo em específico, em níveis mais altos.

Portanto, também nota-se a necessidade da *Engine* comunicar-se com o sistema operacional na qual está sendo executada, para ter acesso aos recursos computacionais que serão utilizados pelos módulos. A renderização da cena de jogo é um exemplo claro que explica essa necessidade, já que usa diretamente recursos da GPU, acessados através do sistema operacional.

2.8 Unreal Engine e Plugins

A Unreal Engine é um dos vários motores de desenvolvimento de jogos disponibilizados gratuitamente e teve sua última grande atualização (versão 5.0) oficialmente liberada ao público no ano de 2022. Esse *software*, desenvolvido pela Epic Games, é uma interface que dá ao usuário acesso a várias ferramentas necessárias para o desenvolvimento de jogos incluindo programação, animação, modelagem e criação de *Shaders*. A Unreal é uma *engine* cujo código fonte está disponível gratuitamente e, portanto, possibilita que usuários possam extender suas funcionalidades básicas. Assim, desenvolvedores criam

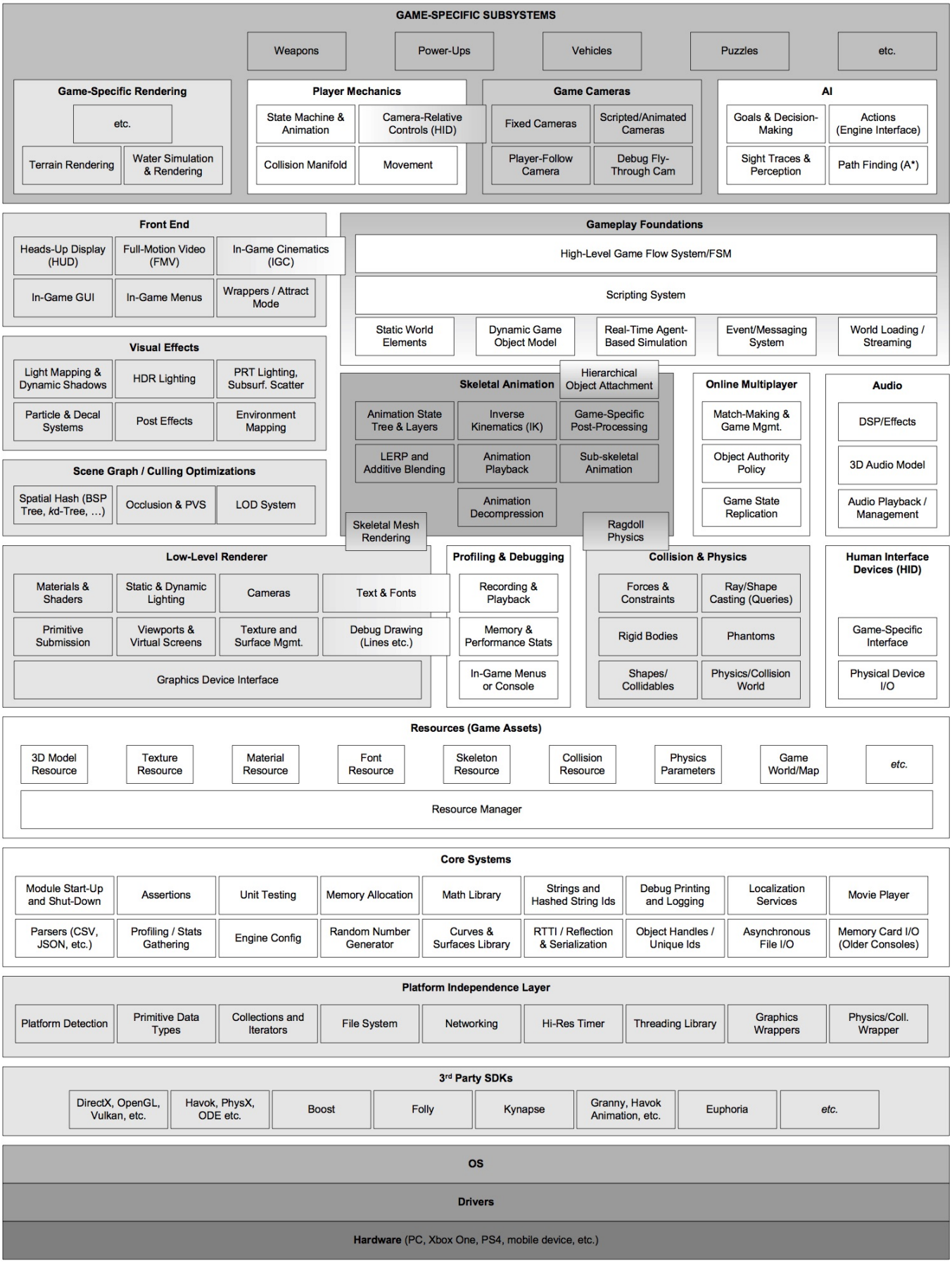


Figura 8 – Arquitetura comum de uma *Game Engine* (GREGORY, 2018)

extensões para o motor, chamados de Plugins, e as publicam online. A Epic Games possui um catálogo oficial de Plugins produzidos pela própria empresa ou por desenvolvedores independentes que podem ser adquiridos por um determinado preço.

2.9 *Assets* e *Asset Packs*

Asset é o nome que a Unreal Engine dá para qualquer tipo de conteúdo utilizável pelo desenvolvedor para construir sua aplicação. Esses *assets* podem ser de diversos tipos, porém o mais importante para o projeto em questão são os *assets* do tipo *Material* (EPIC GAMES, a). Um *Asset Pack* é um pacote compartilhável que contém dentro de si múltiplos *assets*, sendo uma forma muito comum de publicar conteúdo para ser reutilizado em outros programas. Na Figura 9, temos um exemplo de *Asset Pack* disponível no Marketplace da Unreal Engine.

Sendo uma forma específica de Plugin, os *Asset Packs* não se propõem a modificar a maneira como a UE funciona ou estender suas funcionalidades, mas sim disponibilizar conteúdo utilizável na forma de *assets*. O pacote da imagem 9, por exemplo, contém vários *assets* de *Materials*, que podem ser imediatamente utilizados num projeto.

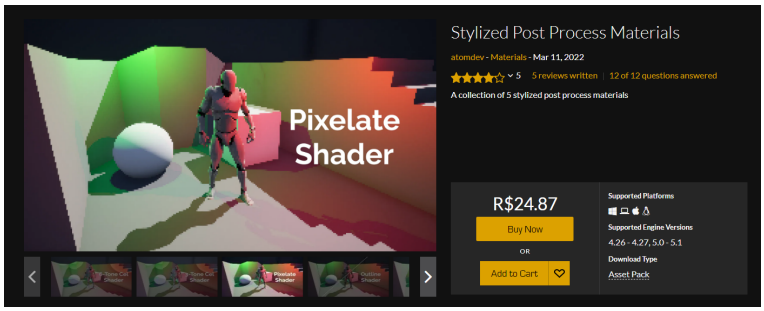


Figura 9 – Exemplo de um *Asset Pack* de *Shaders* disponível no Marketplace da Unreal (EPIC GAMES, i)

Um projeto pode fazer uso de qualquer tipo de Plugin ao incluí-lo em uma pasta específica do projeto e compilar o projeto novamente. Portanto, a adição de novos Plugins pode ser feita apenas extraindo os arquivos necessários no diretório correto. No caso de dos pacotes disponibilizados oficialmente no Marketplace da Epic, a integração com um projeto existente pode ser feita através da própria interface do *launcher*, como ilustrado na Figura 10.

2.10 Dithering

Em computação gráfica, Dithering é uma técnica de introdução de ruído ou padrões em imagens com uma profundidade de cores limitada com o objetivo de criar uma ilusão de uma profundidade de cores maior (STROTHOTTE; SCHLECHTWEG, 2002). Em

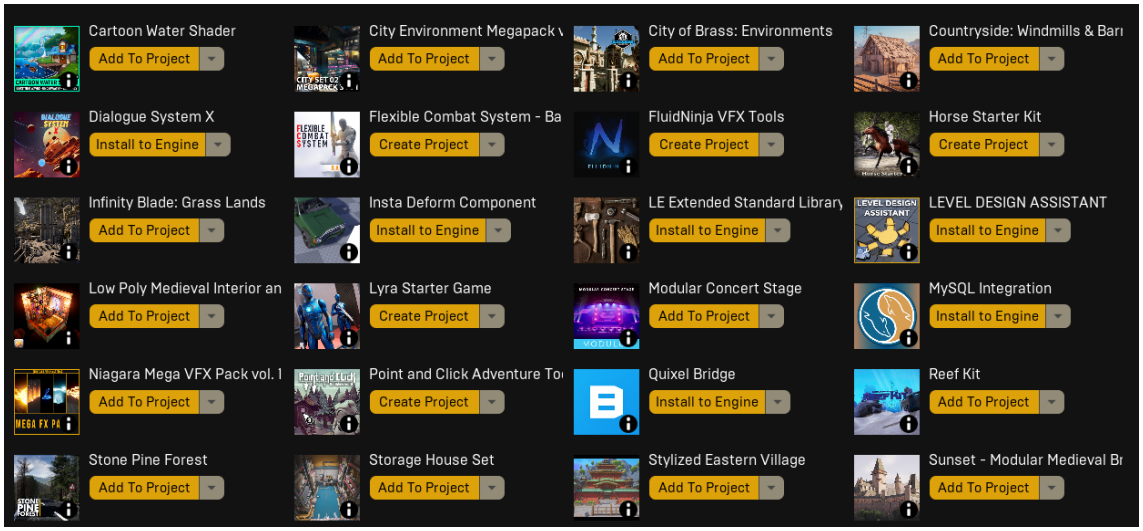


Figura 10 – É possível adicionar um Plugin ou *Asset Pack* pela interface da Epic

computadores antigos, eram algoritmos importantes para manter uma fidelidade visual mesmo com as limitações de cores do *hardware* de 8 bits, 4 bits, ou até mesmo de 1 bit. Hoje em dia, Dithering também é utilizado como um efeito visual para resgatar o imaginário desses computadores antigos. Na Figura 11, temos exemplos de três diferentes algoritmos de Dithering com uma profundidade de cores de 1 bit, somente preto e branco.

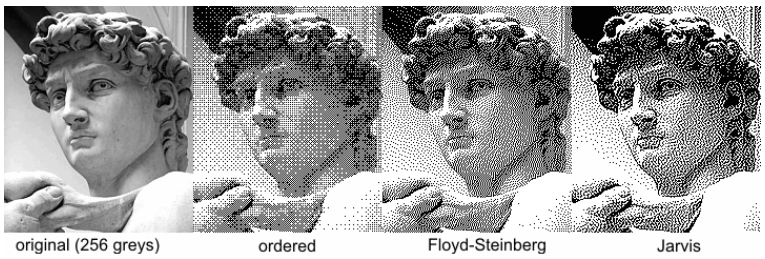


Figura 11 – Exemplo de diferentes algoritmos de Dithering (TOUSSAINT, 2006)

2.11 Cel Shading

Cel Shading é um método utilizado na computação gráfica para atingir efeitos visuais que reproduzam a estética de desenhos animados e quadrinhos tradicionais (MASUCH; RÖBER, 2004). Esse efeito pode ser alcançado de diversas maneiras, geralmente alterando a forma como a luz afeta os materiais, reduzindo gradientes para dar a impressão de que os objetos são 2D. Também é comum a utilização de diversas técnicas para transformar sombras em hachuras (ver Figura 12) ou puntilismo.

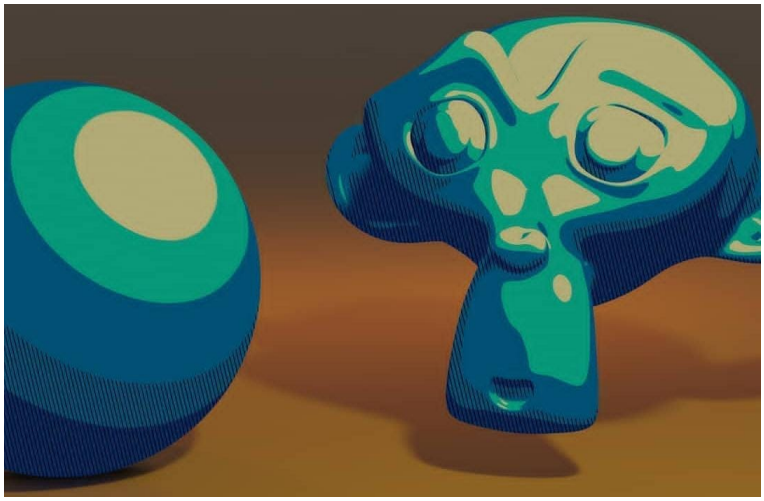


Figura 12 – Exemplo de Cel Shading em esfera e modelo simples de macaco

2.12 Kuwahara Filter

Essa técnica de pós-procesamento tem como objetivo a diminuição de ruído na imagem sendo processada sem afetar a nitidez do contorno dos objetos (BARTYZEL, 2016) (ou seja, sem causar *blur* nos contornos). A diminuição do ruído de cada pixel na imagem, com relação aos pixels em seu entorno, cria um efeito de simplificação nas cores da imagem, de maneira que ela pode parecer uma pintura feita à mão (ver Figura 13).

Figura 13 – Exemplo do Kuwahara Filter sendo aplicado sobre a imagem de um esquilo (PAPARI; PETKOV; CAMPISI, 2007)

2.13 Outline Effect

Outline é o nome dado à técnica utilizada para adicionar uma borda às fronteiras de objetos 3D em cena, dando destaque a essas fronteiras. Isso resulta num efeito estilizado de maneira cartunesca, semelhante a desenhos feitos à mão (WINKENBACH; SALESIN, 1994), como pode ser visto na Figura 14.

Esse tipo de efeito pode ser também aplicado a apenas um objeto, ao invés de toda a cena, destacando o objeto em questão em relação ao restante dos elementos.

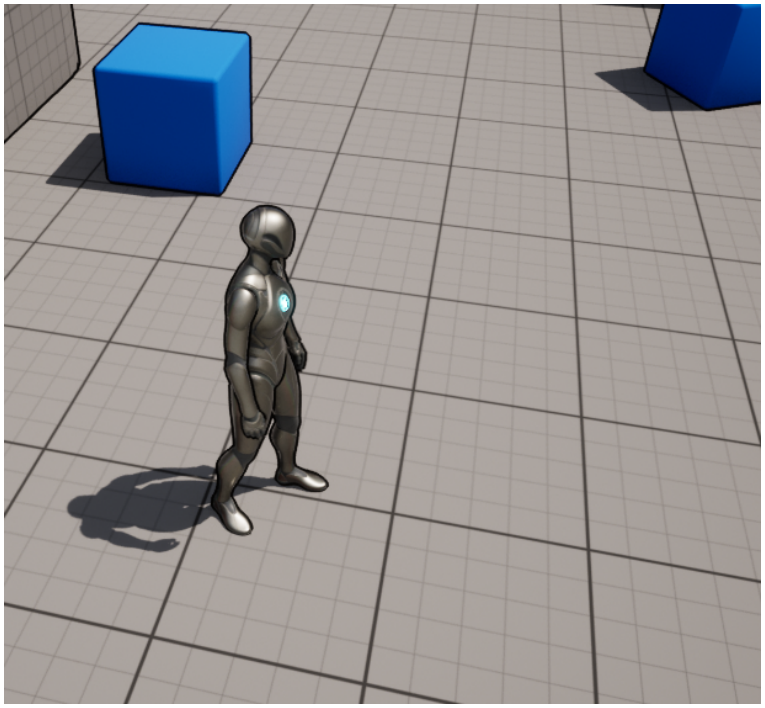


Figura 14 – Efeito Outline sendo utilizado dentro de uma cena de jogo. Os objetos 3D aparecem com um contorno preto.

2.14 Edge Detection

Esse efeito, que não foi listado na primeira enumeração dos *Shaders* propostos para o projeto, baseia-se em parte no algoritmo do Outline, apresentado acima. Esse *Shader* também adiciona borda às fronteiras de objetos 3D em cena, mas descolore todos outros elementos (Figura 38).

2.15 Pixelation

Dentro deste projeto, *Pixelation* se refere ao efeito de propositalmente renderizar um frame de modo que seus pixels fiquem grandes e visíveis, tal qual na Figura 16. Como comentado na introdução desta monografia, este tipo de *Shader* pode ser utilizado dar

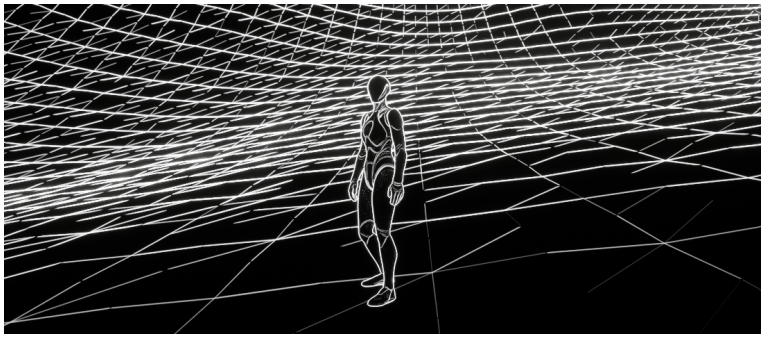


Figura 15 – Efeito de Edge Detection utilizado dentro de uma cena. O contorno dos objetos está em branco enquanto todo o restante está em preto.

um aspecto "retrô"e mais simples a modelos 3D complexos, tal qual no jogo Dead Cells (VASSEUR, 2018).

Figura 16 – Efeito de *Pixelation* utilizado para pixelizar uma imagem no Photoshop.

2.16 CRT Filter

Os monitores de tubos de raios catódicos (ou CRT na sigla em inglês, abreviando "*Cathode Ray Tube*") foram umas das principais formas de se apresentar imagens digitais a usuários antes da criação das tecnologias de LCD (*Liquid Crystal Display*), plasma e LED (*Light Emitting Diode*). Atualmente, muitos desenvolvedores de jogos digitais buscam reproduzir a estética criada pelos monitores CRT como escolha estilística que remete a jogos antigos.

Esse efeito é caracterizado visualmente pelo efeito de "arredondamento"da tela, de maneira a assemelhar-se à curvatura de um *display* CRT antigo. Além disso, há também a presença de *scanlines*, que representavam, na época, regiões em que os raios catódicos não atingiam, colorindo a tela de uma cor diferente, e ocorrendo sempre numa mesma frequência 17.

Figura 17 – Exemplo de uma tela reproduzindo os efeitos de CRT, com *scanlines* (SCANLINES, 2019).

3 Metodologia do Trabalho

O desenvolvimento da ferramenta proposta foi feito em etapas de pesquisa e implementação, como ilustrado no calendário de planejamento da Figura 18. Durante os dois primeiros meses, foram feitos estudos a respeito dos conceitos teóricos de pós-processamento e suas aplicações, exploração das ferramentas e *Asset Packs* de pós-processamento existentes (focando às naquelas presentes no catálogo oficial da Unreal, mas não limitando-se a elas) e código da *Engine*. Com isso, ao final de Abril, consolidamos os requisitos para o nosso pacote de *Shaders* de pós-processamento, com a lista de efeitos e os parâmetros de desempenho esperados, descritos com mais detalhes na próxima secção.

Assim, a primeira fase de implementação da ferramenta foi iniciada, utilizando a Unreal Engine 5.1, na qual foram produzidos o efeitos de "Kuwahara Filter", "Dithering"e "CRT Filter". Essa fase seria seguida por uma busca de *feedbacks*, principalmente de profissionais que atual no mercado de jogos, no entanto essa etapa foi cortada do processo final, para dar mais fôlego ao processo de codificação dos *Shaders*, que se mostrou mais complexo que o esperado. A seguir, foi feita mais uma etapa de estudos e leituras de referências, que focados em resolver as dificuldades que se apresentaram durante a primeira parte da implementação e preparar o desenvolvimento dos filtros restantes.

Na segunda parte da implementação foram produzidos mais três efeitos visuais: "Outline", "Edge Detection"e "Pixelation". Também foi implementado um mapa de testes dentro do projeto, para apresentar rapidamente os efeitos feitos.

Com isso, foi feita uma segunda implementação, seguida de um polimento final do pacote e uma etapa de validação utilizando o *Sample Game* Lyra. O Lyra é um jogo amostra presente na Unreal Engine que foi utilizado para aplicar os diversos efeitos da ferramenta e testar seu funcionamento tanto visual quanto relacionados a desempenho (EPIC GAMES, b). Por fim, planejava-se disponibilizar a ferramenta de forma gratuita no Marketplace oficial da Unreal ao término do projeto. No entanto, a disponibilização sem cobrança de Plugins que incluem *Asset Packs* não é suportada pelo catálogo da Epic. Assim, a disponibilização do trabalho se dará apenas através do GitHub.

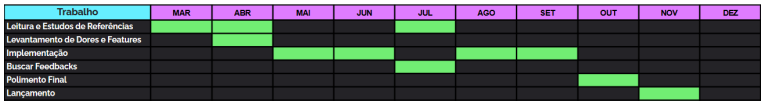


Figura 18 – Calendário de Planejamento do Trabalho de Conclusão de Curso

4 Especificações e Requisitos

Os requisitos funcionais e não funcionais do *Asset Pack* de pós-processamento proposto neste trabalho serão enumerados a seguir. Eles foram elencados de maneira que a conformidade do projeto com os requisitos pudesse assegurar uma ferramente capaz de atingir os objetivos apresentados no início deste documento.

4.1 Requisitos Funcionais

Neste trabalho, os requisitos funcionais serão a disponibilidades de efeitos para uso, bem como uma forma de utilização simples e adequada, que não cause atrito ao desenvolvimento tomando muito tempo.

4.1.1 Efeitos Visuais

Dentre poucos *Asset Packs* que disponibilizam gratuitamente efeitos de pós-processamento no Marketplace oficial da UE, o que maior número de efeitos diferentes encontrados num único pacote foi 5 (EPIC GAMES, g). E mesmo nesse caso, todos efeitos faziam parte de uma mesmo padrão de estilização.

Portanto, foi estipulado que o *Asset Pack* tendo, pelo menos, 5 efeitos de pós-processamento diferentes, estaremos disponibilizando uma quantidade relativamente grande de opções visuais para uma ferramenta gratuita. Desta forma, elencamos um total de 9 efeitos diferentes para serem produzidos, conforme a ordem de prioridade descrita na lista a seguir.

- Efeitos Visuais de Alta Prioridade
  - Dithering.
  - Cel Shading.
  - Painting Effect (Kuwahara Filter).
  - Outline.
  - Pixelation.
  - CRT Filter.
- Efeitos Visuais de Baixa Prioridade
  - Thermal Vision.

- Night Vision.
- Underwater Effect.

É válido ressaltar nesta seção que nem todos os 10 efeitos foram de fato implementados. Além disso, por conta do tempo de desenvolvimento, o efeito de Cel Shading inicialmente proposto foi substituído pelo Edge Detection.

O efeito visual que deseja-se obter com cada um desses *Shaders* será descrito a seguir. A descrição conceitual desses efeitos foi deixada para este capítulo por se tratar de um requisito que o trabalho possa ser considerado uma ferramenta funcional.

4.1.2 Utilização do *Asset Pack*

A utilização do *Asset Pack* desenvolvido deve ser feita mediante a adição do conteúdo da ferramenta dentro da pasta "Plugins"de um projeto da Unreal Engine 5. Após a adição da ferramenta, será necessário fazer uma nova *build* do projeto, para adicionar o caminho correto do código HLSL em cada *Shader* desenvolvido.

Uma vez que a *build* tenha sido concluída sem erros, é possível acessar os *Materials* de pós-processamento contendo os *Shaders*. A utilização deles em cena é realizada através de um objeto da classe *PostProcessVolume*. Esse objeto determina uma região 3D para aplicação de efeitos de pós processamento. O *Asset Pack* deve funcionar de modo que, para adicionar à cena o efeito visual desejado, basta adicionar ao *PostProcessVolume* uma instância de um dos *Materials* desenvolvidos. É relevante realçar que os *Materials* podem não estar disponíveis dentre as opções listadas na Unreal caso a *checkbox* de visibilidade para *assets* dentro de Plugins esteja desmarcada. Além disso, a utilização das *Material Instances* ao invés dos *Materials* originais é recomendada por questões de desempenho.

Com isso, o efeito de pós-processamento escolhido será aplicado na tela de um jogador que esteja dentro da região determinada pelo *PostProcessVolume*. Caso deseje-se que o efeito seja aplicado a qualquer lugar da cena, independente do tamanho e formato do volume, deve-se selecionar a opção *Infinite Extent* nos detalhes do objeto (ver Figura 19). Desta forma, espera-se que o uso do pacote seja facilmente implantável e configurável.

4.2 Requisitos Não-Funcionais

4.2.1 Estrutura

Assim como a maioria dos outros pacotes de pós-processamento para a Unreal Engine, nosso projeto é um *Asset Pack* contendo múltiplos *Materials* de pós-processamento, cada um referente a um efeito visual específico. Esses *Materials* serão desenvolvidos utilizando HLSL (High Level Shading Language) e o editor de materiais da Unreal Engine.

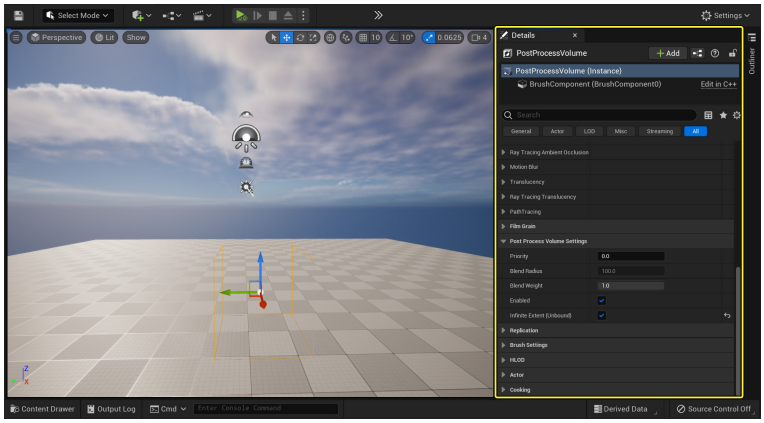


Figura 19 – Detalhes de um objeto da classe *PostProcessVolume* (EPIC GAMES, f)

O código em HLSL é salvo em arquivos especiais da Unreal do tipo ".usf"(*Unreal Shader File*) e acessados em nós customizados no *Material Editor* que descreve cada *Shader* de pós-processamento. Além disso, o projeto já disponibilizará *Material Instances* referentes a cada um dos efeitos desenvolvidos.

4.2.2 Desempenho

Apesar do foco deste trabalho não ser fazer uma ferramenta que deva ser utilizada em versões finalizadas de um grande ou outra experiência interativa, é importante que a adição do *Asset Pack* e a utilização de seus efeitos não causem aumentos significantes na carga de trabalho da CPU e da GPU. Dessa forma, é possível fazer estimativas e medições coerentes do desempenho de um projeto mesmo em estágios iniciais e utilizando o pacote desenvolvido Do contrário, desenvolvedores precisariam esperar estágios mais avançados de desenvolvimento para começar a identificar os gargalos do jogo.

Como é difícil estimar valores de desempenho para um recurso específico que será parte de um projeto maior e que ainda depende fundamentalmente de outros *Shaders* utilizados, quantidade de materiais em cena e iluminação, decidimos utilizar como alvo o valor padrão de "Frames por Segundo"(FPS) da indústria de jogos, que é de **60 FPS**. Além disso, a adição dos efeitos de pós-processamento **não deverá resultar num aumento maior que 1 ms do tempo necessário para a computação desse recurso**.

Para medir esses recursos, a Unreal disponibiliza comandos específicos que podem ser executados para acessar dados de desempenho do jogo. Com esses comandos, é possível detalhar quanto de um frame é gasto em cada tipo de computação (GPU e CPU), além de informações mais específicas como o FPS, tempo de GPU em um frame, número de comandos de renderização por frame, entre vários outros.

A medição do desempenho será feita utilizando projetos disponibilizados pela própria Epic Games. Esses projetos, chamados "*Sample Games*"são pequenos jogos completamente funcionais que contam com arte, UI, animação e jogabilidade desenvolvidos.

Portanto, testar utilizando esses projetos trará mais fidelidade às medições feitas do que medições feitas em cenas vazias ou com poucos elementos. Até o momento, existem quatro *Sample Games* disponíveis ao público.

4.2.3 Documentação

O projeto desse ser acompanhado de uma documentação clara, que explique detalhadamente utilizar o *Asset Pack* e os *Shaders* elaborados, bem como seu funcionamento. Desta forma, novos usuários terão um texto base ao qual voltar-se em casos de dúvidas a respeito da implementação dos efeitos, fazendo desse um material de estudo também. A documentação será armazenada e atualizada no arquivo "README.md"do projeto no GitHub bem como no site oficial do projeto.

O *Asset Pack* também deve contar com uma *Demo*, um projeto de demonstração base para que novos usuários abram e consigam ver facilmente os efeitos em ação. Ela deve conter pelo menos um ator básico, como um personagem, além de alguns objetos ou formas geométricas para visualização, e deve vir já preparada com os *Materials* de efeitos visuais dentro da lista dos *assets* do projeto.

Na feira de apresentação do projeto, essa Demo será utilizada para ilustrar os efeitos produzidos ao longo do trabalho.

4.2.4 Preço

Um dos principais requisitos do projeto é ser completamente gratuito e *open-source*, aberto para que qualquer usuário da UE5 possa utilizar. Desta forma, o repositório online do projeto será público, permitindo que usuários possam fazer alterações e outras versões do projeto.

Portanto, licenciaremos o projeto sob a licensa *open-source* Apache License 2.0 (ASL 2.0). Essa licensa permissiva permite o uso do trabalho desenvolvido em projetos em outras licensa (ou seja, permite o uso em projetos comerciais), além de prever a possibilidade de modificação do código fonte original.

4.2.5 Distribuição

A distribuição do pacote será feita pelo GitHub, por meio do qual os usuários poderão baixar os *assets* desenvolvidos e consultar a documentação. A princípio, planejava-se distribuir o projeto também por meio do Marketplace oficial da UE, para que ele pudesse ser encontrado por desenvolvedores de maneira mais prática e automaticamente integrado a projetos. Entretanto, a Epic Games possibilita a disponibilização de pacotes gratuitos somente caso eles sejam Plugins de código (EPIC GAMES, c), o que não é o

caso deste *Asset Pack*. Deste modo, foi tomada a decisão de não aplicar para distribuição no Marketplace, já que um dos principais propósitos do projeto é ser gratuito.

5 Desenvolvimento do Trabalho

5.1 Tecnologias Utilizadas

5.1.1 *Material Editor* da Unreal Engine

O *Material Editor* é uma das ferramentas oferecidas pela Unreal e providencia um *work-flow* facilitado para a criação e edição de materiais utilizando um *script* visual. Esses materiais podem ser aplicados tanto em objetos 3D, definidos por uma malha de vértices (e, portanto, triângulos) no espaço tridimensional, quanto em efeitos de pós-processamento, como é o objetivo deste trabalho.

Dentre as funcionalidades do *Material Editor* utilizadas ao longo do projeto, destacam-se a interface de acesso a diferentes componentes da GPU, como o *buffer* de renderização e o *buffer* de distância (também chamado de *z-buffer*). O acesso a esses endereços de memória pode ser feito simplesmente adicionando nós ao *script* visual do Editor, como ilustrado na Figura 20.

Vale ressaltar também, outras pequenas funcionalidades úteis dessa ferramenta, como a possibilidade de criar parâmetros editáveis de diversos formatos de dados e a já citada capacidade de, a partir de um *Material*, criar um *asset* filho "*Material Instance*"(Figura 21), que permite pré-calcular parte do código que não esteja disponível a alterações, otimizando o tempo de renderização.

5.1.1.1 *High Level Shader Language*

Os *Shaders* de pós-processamento nesse trabalho foram produzidos utilizando *High Level Shader Language* (HLSL), uma linguagem baseada em C desenvolvida pela Microsoft para facilitar o desenvolvimento na área de computação gráfica (VARCHOLIK, 2014). Essa linguagem voltada para o desenvolvimento de *Shaders* foi disponibilizada pela primeira vez em 2002, com o lançamento da API gráfica DirectX 9.

A utilização do HLSL na Ureal é feita através de nós customizáveis do próprio Editor de *Materials* que permitem a inserção de código HLSL. Apesar de que seria possível produzir *Shaders* utilizando apenas os nós disponíveis no *script* visual, optamos pela utilização do HLSL para ter mais controle dos *Shaders* produzidos e seu funcionamento, além de ser mais simples de demonstrar o código produzido neste trabalho. Isso é feito através de nós customizáveis nos quais o *Material Editor* permite a inserção de código HLSL (Figura 22).

No entanto, escrever códigos mais longos e complexos se torna mais complicado

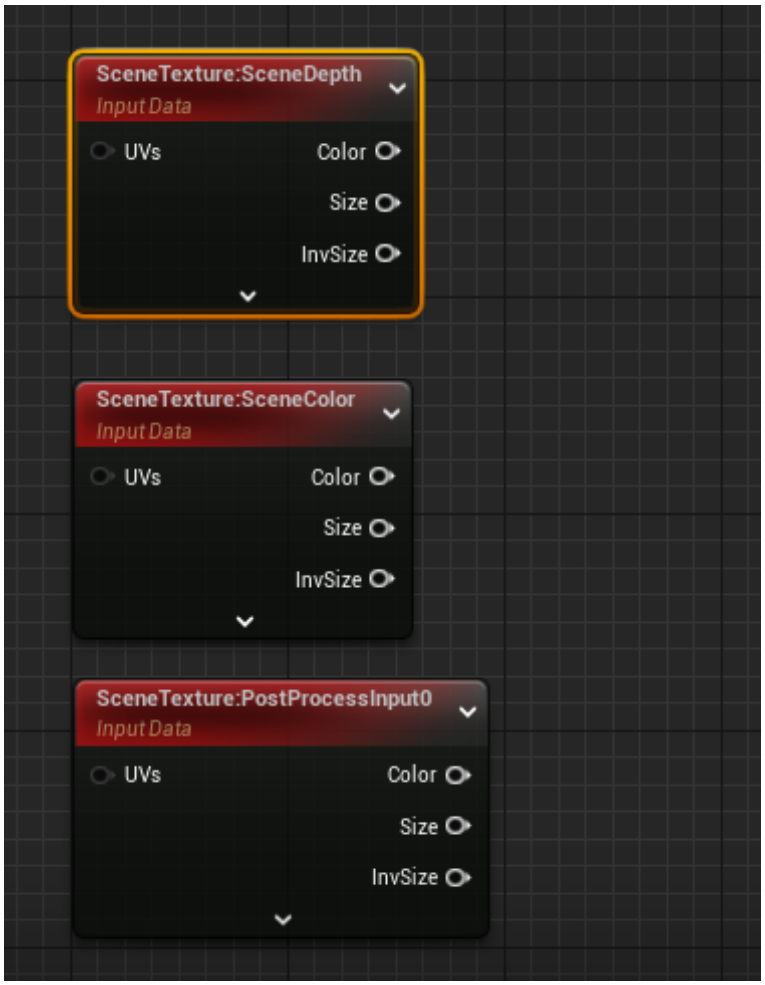


Figura 20 – Nós do *script* visual do Editor de *Materials* que permitem acessar, respectivamente, o *buffer* de distância, as cores da cena e as cores da cena já renderizada e pronta para o *post-processing*

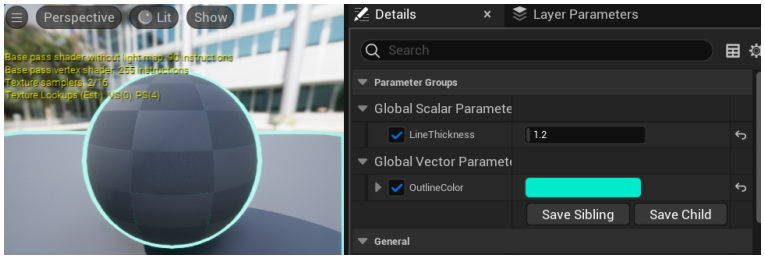


Figura 21 – *Material Instance* criada a partir do *Material* do efeito de Outline. Percebe-se os parâmetros "Line Thickness"e "Outline Color", que podem ser confiugrados pelo usuário.

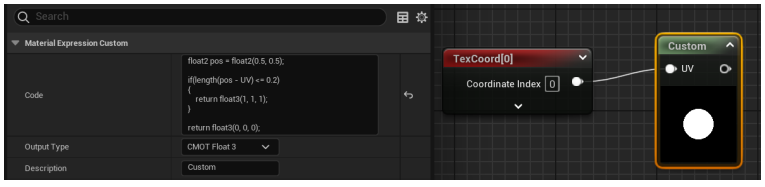


Figura 22 – Nó customizável simples, que desenha um círculo na tela. O código HLSL pode ser visto ao lado.

dentro de nós num editor de *script* visual. Além disso, com o código HLSL existindo como uma parte do *asset*, que é um arquivo binário, modificações ao código não podem ser rastreadas em detalhes pelo programa de versionamento utilizado, o git.

Para resolver esses problemas, o código HLSL foi incorporado a arquivos externos que são referenciados nos nós customizados (Figura 23), simplificando o conteúdo do nó e permitindo que alterações feitas sejam facilmente visíveis para todos os colaboradores do projeto.

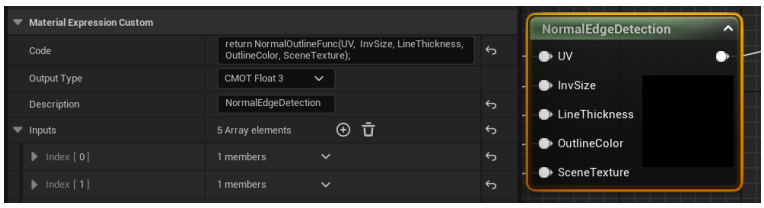


Figura 23 – Nó customizável que referencia um arquivo USF, contendo o código HLSL que define uma função para o *Shader* de Outline.

5.2 Projeto e Implementação

A seguir, descreveremos os algoritmos utilizados na implementação de cada um dos *Shaders* propostos. Os códigos HLSL para cada um dos efeitos pode ser visto no Apêndice A deste documento.

5.2.1 Dithering Ordenado

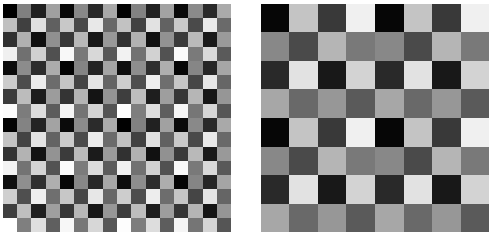
Existem múltiplas formas diferentes de implementar o algoritmo de Dithering Ordenado, como exemplifica bem o artigo "Joel Yliluoma's arbitrary-palette positional dithering algorithm"(YLILUOMA, 2011). Entretanto, muitos deles demandam tempo de processamento demais para serem efetivamente aplicados em tempo real. Desta forma, para a implementação dos algoritmos propostos neste projeto, optou-se por uma solução mais simples porém mais rápida.

A seguir, explicaremos o funcionamento do Dithering Ordenado programado para o nosso projeto utilizando a imagem 24 como exemplo:

Primeiramente, escolhe-se o padrão de Dithering a ser utilizado no algoritmo. Este é um parâmetro que o usuário do *Asset Pack* poderá definir, porém, como *default*, deixamos as opções de utilizar alguma matriz de Bayer (BAYER, 1973), como mostrado na Figura 25.

Além disso, também deve ser escolhida a paleta de cores utilizada para renderizar a imagem, que por padrão selecionamos como uma paleta com profundidade de cor de 12

Figura 24 – Imagem antes de passar pelo Dithering (OP1, 2023).



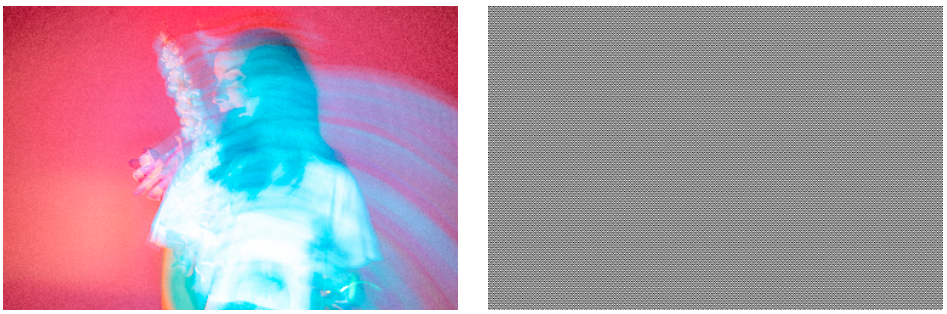
(a) Padrão 16x16 (b) Padrão 8x8

Figura 25 – Diferentes padrões de Bayer Dithering.

bits, sendo 4 bits para cada canal de cor. Entretanto, também é possível selecionar sua própria paleta de cor para utilizar o Dithering.

Em seguida, este padrão é repetido por todo o tamanho da imagem original e os valores de cor de cada pixel - de 0.0 a 1.0 - são somados, seguindo a equação 5.1. O fator r é ajustável e foi escolhido como sendo $\frac{1}{5}$.

$$dither\_color = find\_closest\_color(original\_color + r * (pattern\_color - 0.5)) \quad (5.1)$$



(a) Imagem Original (b) Padrão Repetido

Figura 26 – Comparação do padrão com a imagem original.

A função $find\_closest\_color$ recebe como *input* o resultado da soma e deve buscar na paleta de cores selecionada a cor mais próxima à calculada. Esta função foi implementada determinando individualmente a distância entre a cor calculada e cada cor da paleta de cores, utilizando algum cálculo de distância, como a distância euclideana, por exemplo.

No algoritmo proposto neste projeto, foi implementada uma soma ponderada entre a distância euclideana e a distância entre os valores de *luma* das cores, seguindo a equação 5.5. Esta escolha se deve ao fato do cérebro não perceber a diferença entre as cores de maneira euclideana, assim compensando calibrar uma função de distância diferente.

$$luma\_convert = (0.299, 0.587, 0.114) \tag{5.2}$$

$$euclidean\_distance = ((r_1, g_1, b_1) - (r_2, g_2, b_2))^2 \tag{5.3}$$

$$luma\_distance = (r_1, g_1, b_1) \cdot luma\_convert - (r_2, g_2, b_2) \cdot luma\_convert \tag{5.4}$$

$$distance = euclidean\_distance \cdot luma\_convert * 0.75 + luma\_distance \tag{5.5}$$

Após executar a função $find\_closest\_color$ para cada pixel da imagem, é possível renderizar a imagem final já com uma paleta de cores reduzida (Figura 27).

Figura 27 – Imagem após passar pelo Dithering.

A Figura 29 a seguir exemplifica os efeitos de Dithering Ordenado dentro da própria Unreal Engine, com duas paletas de cores diferentes.

5.2.2 Kuwahara Filter

A implementação do Kuwahara Filter (KUWAHARA et al., 1976) neste trabalho foi feita utilizando dois métodos diferentes. Primeiro, utilizou-se uma abordagem baseanda na definição de 4 quadrantes ao redor de um pixel de análise (Figura 30). Os quadrantes são quadrados com aresta de valor definível pelo usuário.

Em cada quadrante, somam-se a cores dos pixels e calcula-se a média e a variância das cores no quadrante. As equações 5.6 e 5.7 mostram o cálculo da cor média $\overline{c}$ e da variância σ^2 para um quadrante de n pixels.

$$\overline{c} = \frac{\sum_{i=0}^{n} c_i}{n} \tag{5.6}$$

$$\sigma^2 = \frac{\sum_{i=0}^{n} c_i^2}{n} - \overline{c}^2 \tag{5.7}$$

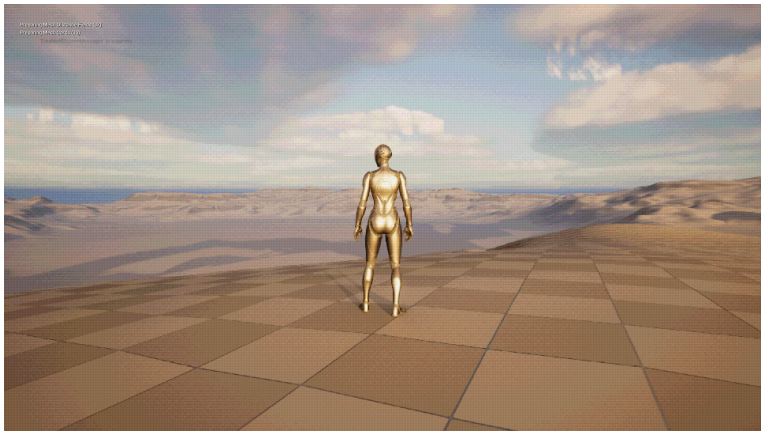


Figura 28 – Dithering Ordenado com paleta de cores de 12 bits

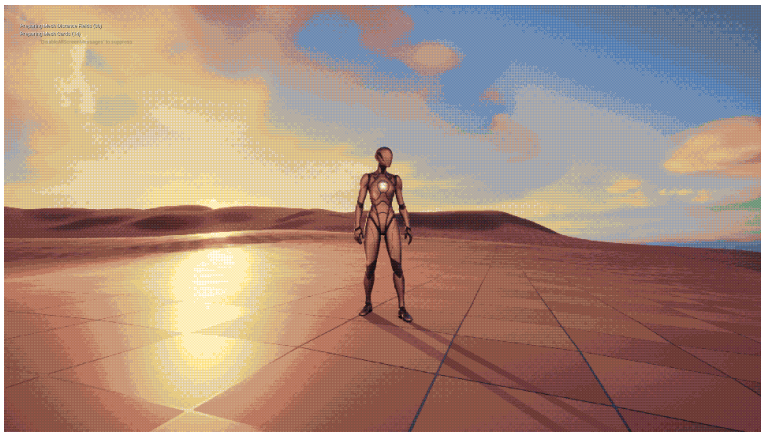


Figura 29 – Dithering Ordenado com paleta de cores Endesga-32 (ENDESGA, 2020)

Ao final do cálculo para todos os quadrantes, o pixel de análise recebe a cor média do quadrante com menor variância.

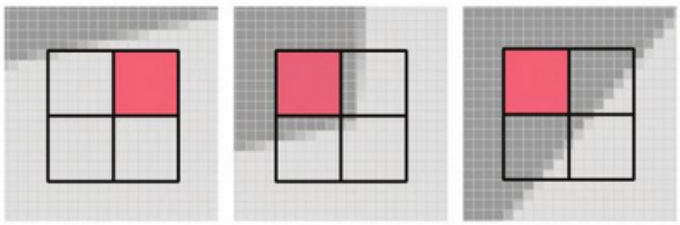


Figura 30 – Exemplos do algorítmo do Kuwahara Filter sendo aplicado em pixels em diferentes situações, com o quadrante com menor variância de cores destacado. A imagem provém de um artigo de 2009 (KYPRIANIDIS; KANG; DÖLLNER, 2009), que detalha o filtro não-linear original proposto em 1976 (KUWAHARA et al., 1976)

Isso permite a simplificação das cores dos objetos mas com a preservação das fronteiras bem definidas entre um objeto e outro, já que quadrantes com maior variação nas cores têm menos chances de serem escolhidos para pintar o pixel de análise. Assim, esse efeito de simplificação de cores confere à imagem um aspecto semelhante a uma pintura (Figura 31).

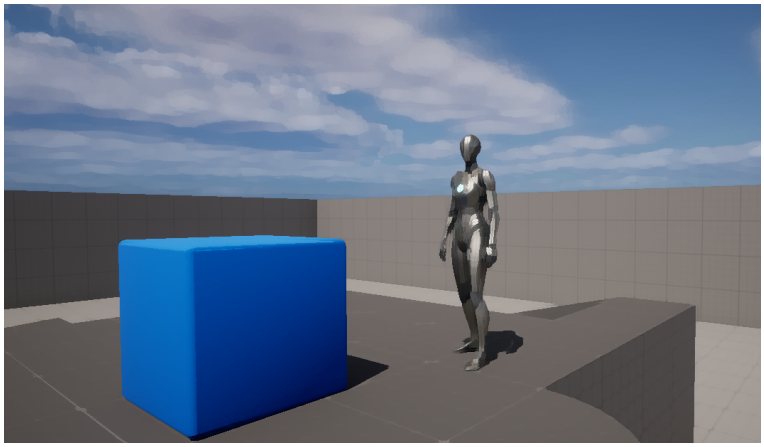


Figura 31 – Efeito do Kuwahara Filter com abordagem de quadrantes quadrados em jogo. Os objetos têm seus contornos preservados, mas suas cores internas são simplificadas.

A seguir, para tentar deixar a simplificação de cores menos brusca, foi utilizado uma região circular para definir os pixels ao redor do pixel de análise, ao invés de usar uma região quadrada. Essa abordagem foi baseanda nos trabalhos de Papari (PAPARI; PETKOV; CAMPISI, 2007), mas com a simplificação de ainda usar 4 quadrantes ao invés dos 8 quadrantes propostos nesse artigo. O usuário ainda pode definir o tamanho do raio do círculo.

Essa abordagem não resultou em diferenças notáveis nas cenas básicas testadas, nem causou alterações perceptíveis na taxa de *frames* por segundo do jogo em comparação com a primeira abordagem proposta.

Foi também feita uma tentativa de desenvolver um filtro que utiliza uma área de análise circular, mas permite que o usuário defina em quantos quadrantes o círculo será dividido para a análise, dentro um valor entre 1 e 16.

Esse método gerou um número muito maior de artefatos na imagem, além de impactos mais significativos no desempenho, mesmo ao definir um número de quadrantes igual a 4, semelhante à abordagem anterior (Figura 32). Por essa razão, esse algoritmo foi removido da versão final da ferramenta.

5.2.3 Outline Effect

Sendo o Outline um effeito aplicado sobre o contorno dos objetos, é necessário definir os limites de cada objeto na cena renderizada. Isso pode ser feito utilizando dois algoritmos que se complementam: um feito através do *buffer* de distância da GPU e outro feito através das normais de cada triângulo que compõem os objetos 3D, obtidas com o cálculo da geometria da cena (DECAUDIN, 1996).

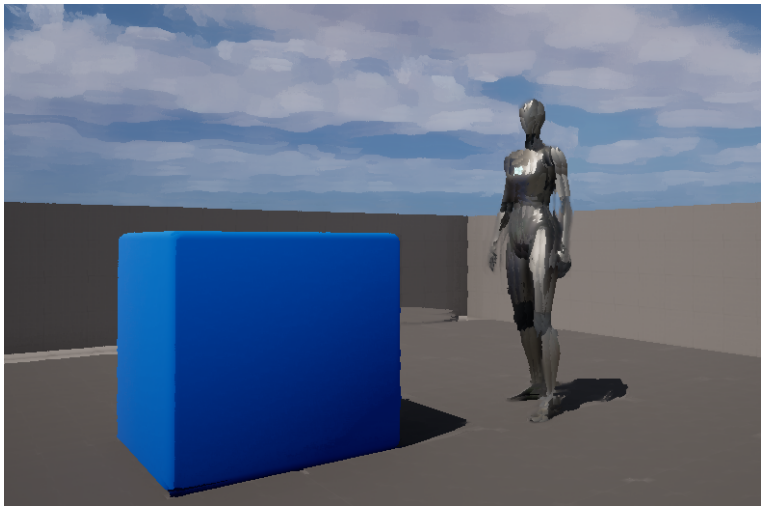


Figura 32 – Efeito do Kuwahara Filter com possibilidade do usuário definir a quantidade de quadrantes. O círculo foi configurado com raio 3.0 e com 4 quadrantes. É possível notas a quantidade maior de artefatos visuais, como serrilhado, nessa imagem.

5.2.3.1 Obtendo contornos de objetos a partir da distância

O *buffer* de distância, também conhecido com o*zbuffer* de uma GPU, trabalha com dados de apenas 1 ponto flutuante por pixel, em oposição à cena renderizada, que atribui vetores de 4 pontos flutuantes para cada pixel. Essa característica faz com que o uso das informações de distância seja mais otimizado do que trabalhar com a renderização da cena.

Fazendo um *shifting* da imagem em um pixel para a direita e subtraindo a imagem obtida da original, percebemos que os pixels de contorno e à esquerda do objeto são coloridos de branco. Isso ocorre pois pixels que tem a mesma distância do ponto de vista (e portanto possuem o mesmo valor no *buffer* de distância) resultam em zero (cor preta) quando subtraídos. No entanto, os pixels que representam a fronteira do objeto estarão adjacentes a pixels com distâncias diferentes, e a subtração com a imagem após o *shifting* resultará em valores diferentes de zero (Figura 33).

Fazendo isso para outras direções (cima, baixo e esquerda), podemos cobrir praticamente a totalidade das fronteiras dos objetos, como ilustrado na Figura 34.

Assim, com informações a respeito dos pixels que representam as fronteiras dos objetos em cena, é possível escolher pintá-los com outras cores. No exemplo da Figura 35, pintamos os pixels de contorno de azul.

Nota-se, no entanto, que regiões com distâncias muito próximas não recebem um contorno. Por exemplo, do terreno não está pintada de azul, pois a dos pontos na distância da face direcionada virada à esquerda é praticamente a mesma dos pontos pertencentes à face virada à direita. Esse problema pode ser resolvido utilizando o segundo algoritmo para detecção de bordas de objetos em cenas 3D.

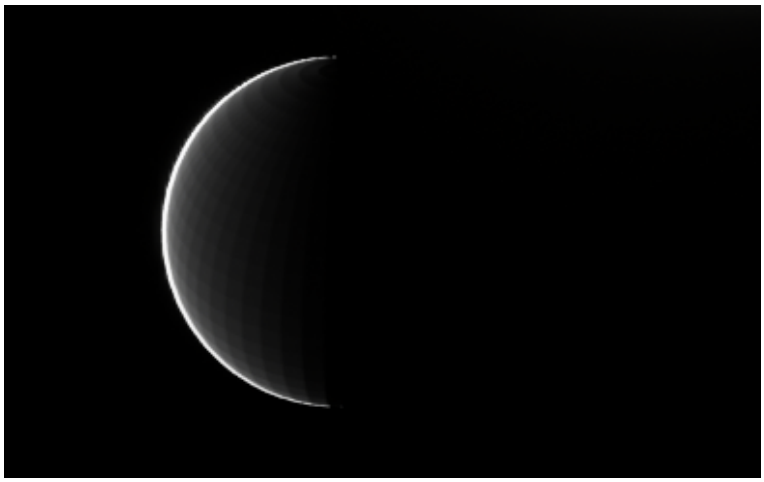


Figura 33 – Imagem de esfera após aplicarmos um *shifting* à direita e subtrair a imagem obtida da original.

Figura 34 – Imagem de esfera após aplicarmos um *shifting* à direita, esquerda, cima e baixo. e subtrair a imagem obtida da original.

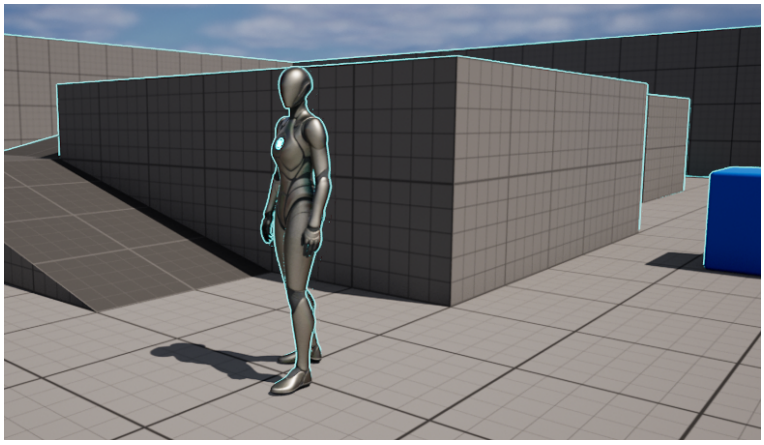


Figura 35 – Efeito de Outline com objetos 3D sendo contornados de de azul, para facilitar a visualização. Percebe-se que a aresta do terreno não foi colorida.

5.2.3.2 Obtendo contornos de objetos a partir das normais

Utilizando as informações dos triângulos representados em cada pixel da cena, é possível obter os vetores normais aos planos definidos por esses triângulos. Com esses dados, podemos identificar mudanças bruscas nas direções das normais de pixels adjacentes (utilizando um método semelhante ao *shifting*) para saber quando devemos pintar esses pixels.

Obtendo os valores absolutos da diferenças entre as normais dos pixels adjacentes na vertical (em cima e em baixo) e dos pixels adjacentes na horizontal (à direita e à esquerda), podemos definir o quanto a normal muda ao redor de um pixel somando esses valores. Esse processo está representado nas equações 5.8, 5.9 e 5.10.

$$\text{horizontal normal diff} = \text{abs}(\text{right pixel normal} - \text{left pixel normal}) \tag{5.8}$$

$$\text{vertical normal diff} = \text{abs}(\text{up pixel normal} - \text{down pixel normal}) \tag{5.9}$$

$$\text{normal diff} = \text{horizontal normal diff} + \text{vertical normal diff} \tag{5.10}$$

A partir dessas inforamções, obtemos o maior valor dentre os componentes do vetor "normal diff"e interpolaremos a partir dele um valor entre a cor original da cena e a cor do Outline. Os resultados podem ser vistos na Figura 36.

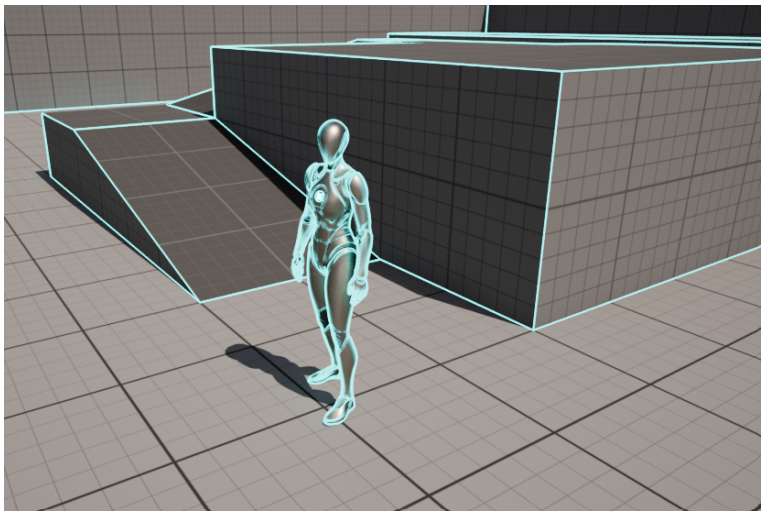


Figura 36 – Efeito Outline agora utilizando as normais dos objetos. Dessa vez, a aresta do terreno está colorida de azul, assim como várias partes do humanoide.

É válido ressaltar que ambos algoritmos também estavam gerando pequenos artefatos visuais quando objetos 3D se moviam rapidamente. Esse tipo de evento causa

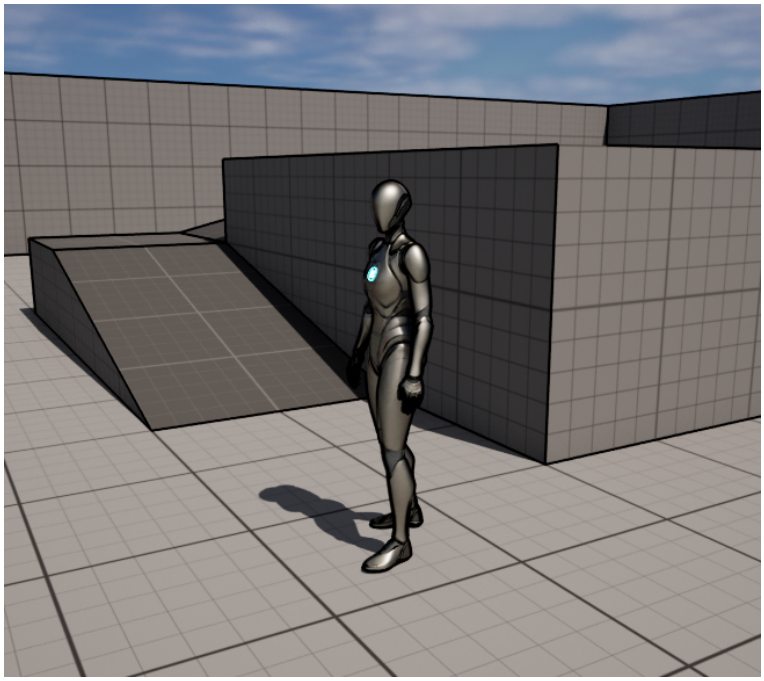


Figura 37 – Também é possível utilizar esse efeito para obter um aspecto visual mais parecido ao de desenhos feitos à mão quando definimos a cor de um Outline para preto.

"rastros"da cor do Outline deixados pelo movimento do objeto. Isso ocorre pois o movimento gera uma diferença momentânea tanto em relação às distâncias de objetos representados por pixels adjacentes ao objeto em movimento quanto pelas normais dos triângulos desses objetos em cena.

5.2.4 Edge Detection

Utilizando os mesmos algoritmos apresentados acima para o Outline, podemos criar o efeito de "Edge Detection". A diferença em relação ao Outline é que os pixels não pertencentes a bordas de objetos também são coloridos, mas com uma outra cor.

5.2.5 Pixelation

Para implementar o efeito "Pixelation", optou-se por um algoritmo simples com a seguinte lógica: um parâmetro *pixel_size* é passado e, em seguida, cada frame é dividido em M pixels de tamanho *pixel_size*. Em sequência, cada pixel amostrará uma única coordenada UV da imagem original, assim todos os pixels terão uma cor uniforme, dando um aspecto retrô ao resultado final. O funcionamento deste efeito fica mais claro ao observar a figura 39, com um antes e depois da aplicação do shader.

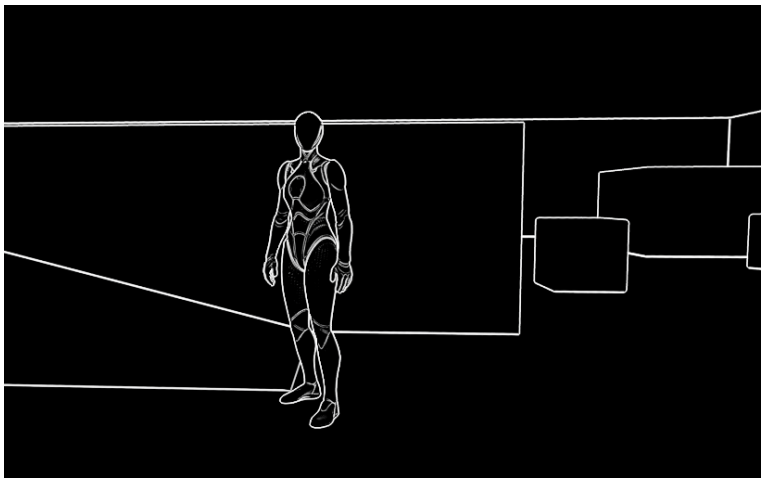


Figura 38 – Edge Detection com as bordas pintadas em branco e o restante dos pixels em preto. Apesar das semelhanças no funcionamento do algoritmo, os efeitos gerados são visualmente distintos e proporcionam um aspecto estético diferente.

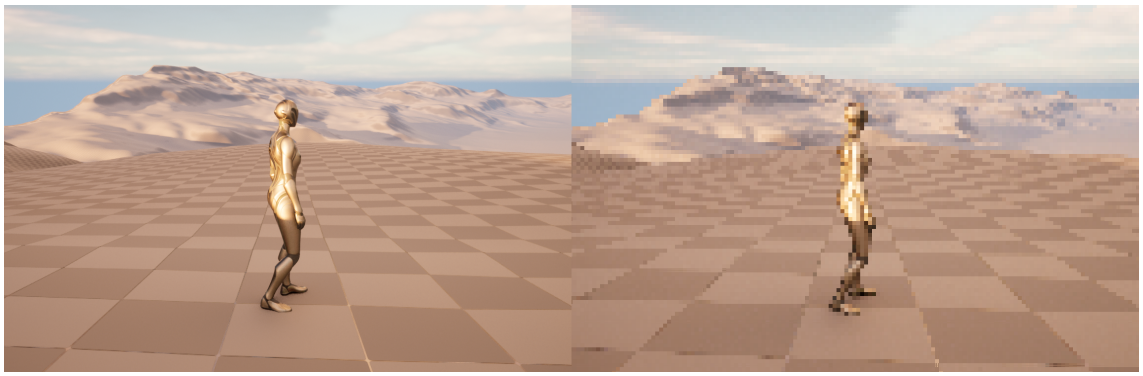


Figura 39 – Antes e depois do efeito "Pixelation".

Figura 40 – Efeito "Pixelation"sendo utilizado em uma cena de demonstração.

5.2.6 CRT Filter

Para se reproduzir o visual dos monitores de tubos de raios catódicos, alguns fatores principais foram considerados. O primeiro deles é o fato de que as imagens são geradas nesses monitores por meio da descarga de feixes de elétrons na tela, estimulando um material fosforescente. Devido a esses feixes de elétrons que varrerem a tela linha por linha, as telas ficam mais escuras no topo, onde a primeira linha é produzida, e claras na parte de baixo, onde as linhas haviam sido ativadas mais recentemente. Para resolver esse problema, os monitores CRT costumam ativar linhas alternadas, gerando assim uma imagem com brilho aparentemente mais homogêneo. O efeito colateral dessa solução é que ficam visíveis faixas de brilho mais forte e mais fraco, alternando entre uma linha e outra. A esse efeito é dado o nomes de *scanlines* (Figura 41). Para simular as *scanlines*, foi introduzido no código de pós-processamento uma seção que altera os componentes vermelho, verde e azul dos pixels da tela de acordo com ondas senoidais cujos parâmetros são definidos no *Shader*. Dessa forma, é possível gerar qualquer padrão de alternância entre pixels de uma linha e outra, inclusive sem a necessidade de alterar todos os canais de cor simultaneamente, o que permite uma maior flexibilidade nos efeitos produzidos.

Figura 41 – Exemplo de imagem com e sem *scanlines*

Outro fator que se sobressai ao observar imagens em monitores CTR é a distorção causada pela tela curva, que era feita neste formato para permitir que os feixes de elétrons atingissem a tela em um ângulo mais próximo a noventa graus. O formato curvo também visava proporcionar uma maior resistência mecânica à alta diferença de pressão dentro e fora do tubo de raios, no qual é mantido um espaço de vácuo para a transmissão dos elétrons. Para gerar esse efeito, foi introduzida uma distorção nas coordenadas UVs dos pixels, de forma que a imagem final aparente estar arredondada nas bordas. O conjunto de fórmulas utilizado para gerar a distorção na coordenada UV de cada pixel está descrito nas equações 5.11, 5.12 e 5.13.

$$UV_{d1} = \left(\frac{UV_o - 0,5}{2}\right) \tag{5.11}$$

$$UV_{d2} = \frac{c \cdot UV_{d1}}{\sqrt{c^2 - \text{dot}\,(UV_{d1}, UV_{d1})}} \tag{5.12}$$

$$UV_d = \frac{UV_{d2}}{2} + 0,5 \tag{5.13}$$

Nessas equações, UV_o é o valor original da coordenada UV do pixel, UV_{d1} é a coordenada do pixel mapeada para o intervalo de -1 a 1, UV_{d2} é a coordenada distorcida para esse novo intervalo, c é um parâmetro para controlar a intensidade da curvatura, e UV_d é a coordenada distorcida final utilizada para a busca da cor do pixel, após os valores serem mapeados de volta para o intervalo de 0 a 1, próprio das coordenadas UV.

Ao aplicar essa distorção, porém, os pixels curvados nas bordas da tela apresentam um claro serrilhamento, devido à sua natureza discreta. Para atenuar esse efeito, foi introduzida uma vinheta, que utiliza por sua vez a mesma fórmula da distorção da tela, aliado a uma função smoothstep para gerar um gradiente ao longo das bordas, escondendo o serrilhado (Figura 42).

Figura 42 – Efeito CRT sendo utilizado dentro de uma cena de jogo.

5.3 Testes e Avaliação

Para testar o funcionamento e principalmente o desempenho dos *Shaders* desenvolvidos em um ambiente similar a de um projeto de vídeo-game real, foram utilizados os *Samples Games* disponíveis no Marketplace da UE. Dentre os 4 *Samples Games* disponíveis no momento deste trabalho, selecionamos 3 para utilizarmos como ambiente de testes: *Lyra*, *StackOBot* e *Valley of the Ancient*. O último dessa lista, no entanto, não foi utilizado nos testes por ser um projeto muito grande e que necessitava de recursos computacionais mais robustos que os disponíveis na máquinas utilizadas para os testes.

5.3.1 Hardware Utilizado e Metodologia de Tests

Os testes foram performados utilizando dois computadores, com os componentes listados a seguir.

- Computador 1
  - CPU: Intel i5 2,5 GHz
  - Memória RAM: 8GB
  - GPU: NVIDIA GeForce RTX 3050
- Computador 2
  - CPU: Intel i7 4,2 GHz
  - Memória RAM: 16GB
  - GPU: NVIDIA GeForce GTX 1070

Além disso, todos testes foram feitos no modo *Standalone Play* da Unreal Engine, que permite que o jogo seja executado como uma aplicação a parte do motor de jogos. Isso significa que a execução de funcionalidades da *Engine* influenciarão menos nos resultados dos testes.

As aferição dos parâmetros foram feitas utilizando comandos de desenvolvimento que dão acesso às medições de FPS e tempo médio gasto no pós-processamento dentro da *thread* da GPU. Para tanto, os comandos utilizados foram, respectivamente, o **stat FPS** e o **profileGPU**, além da ferramenta de medições implementada como um programa a parte, a Unreal Insights (Figura 43). A medições foram feitas levando em consideração o período que o jogo leva para se estabilizar, ter instanciado e carregado na memória as classes centrais, de forma que não haja bruscos saltos nos valores medidos. Finalmente, é válido mencionar que, para melhor se adequar às limitações de hardware da máquina utilizada para testes, a API de renderização padrão dos projetos utilizados foi trocada da DirectX12 para DirectX11, que consome menos memória em sua execução.

5.3.1.1 Parâmetros nos *Materials*

Durante os testes, os seguintes parâmetros foram configurados em cada um dos *Material Instances* utilizados.

- Outline:
  - Line Thickness: 1.0
  - Color: RGB (0, 0, 0)

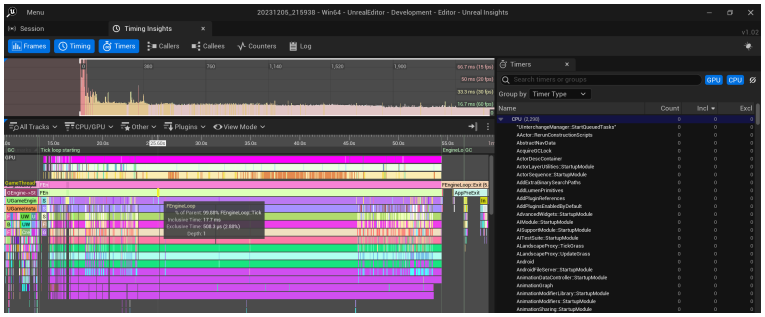


Figura 43 – Ferramenta para medição de Desempenho "Unreal Insights". Aqui é possível ver as medições de frames dos testes no jogo "StackABot".

- Kuwahara
  - Circle Radius: 5.0
- Edge Detection
  - Line Thickness: 1.0
  - Edge Color: RGB (1, 1, 1)
  - Fill Color: RGB (0, 0, 0)
- CRT
  - Curvature: 5.0
- Pixelation
  - Pixel Scale: 4.0
- Dither
  - Bit Depth: 16.0
  - Dithering Resolution: 2.0
  - Dither Pattern: bayer16.uasset
- Dither Palette
  - Dither Size: 1.0
  - Paletter Size: 32.0
  - Dither Palette: endesga-32-1x.uasset
  - Dither Pattern: bayer16.uasset

5.3.2 Testes no *Lyra*

Lyra é um *Sample Game* de tiro, feito para simular partidas online de jogadores num formato de times. Portanto, os testes de FPS foram realizados em três situações distintas que representam diferentes estados de jogo: **o jogador sozinho e parado**, **o jogador sozinho se movimentando e atirando**, e, por fim, **o jogador interagindo em cena com outros peões no jogo** (controlados por um algoritmo).

Todos os testes foram feitos no mapa "L_Expanse_Blockout.umap". Para acessar corretamente os conteúdos do Plugin criado, por conta das restrições impostas a esse projeto específico, os *Materials* de pós-processamento foram copiados para pastas internas do conteúdo do projeto.

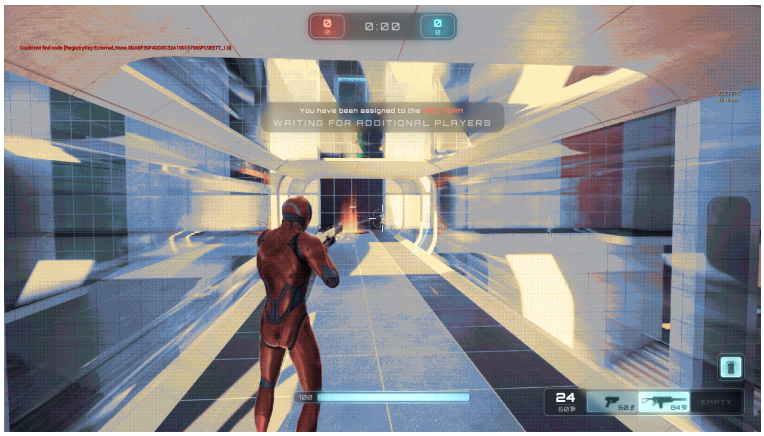


Figura 44 – Efeito de Dithering Palette aplicado no Lyra.

5.3.2.1 Resultados das medições de FPS

Os gráficos na Figura 45 ilustram as medições de Frames por Segundo em cada um dos cenários descritos anteriormente. Nesses gráficos, estão retratados os valores de pico e de vale das medições após a estabilização do jogo.

Percebe-se, primeiramente, que o jogo não chegava a performar a 60 FPS no Computador 1 mesmo na sua versão base, sem a aplicação dos efeitos de pós-processamento produzidos. Isso provavelmente se deu pelas limitações da máquina utilizada para os testes. Ao mesmo tempo, percebe-se que o impacto causado pelos *Shaders* causou, no máximo, uma queda de pouco mais de 13.4% no FPS, com o efeito do Kuwahara Filter. Esses valores, apesar de não alcançarem os 60 FPS planejados, podem ser considerados suficientemente satisfatórios. No Computador 2, por ter uma placa de vídeo inferior, as medições de FPS não chegaram a 30 FPS, porém não notou-se uma grande queda de desempenho em função dos efeitos de pós-processamento implementados.

Também é válido notar quer o maior impacto dos *Shaders* se mostrou quando o jogador estava parado. Ao se movimentar e interagir com outros peões dentro do jogo, a

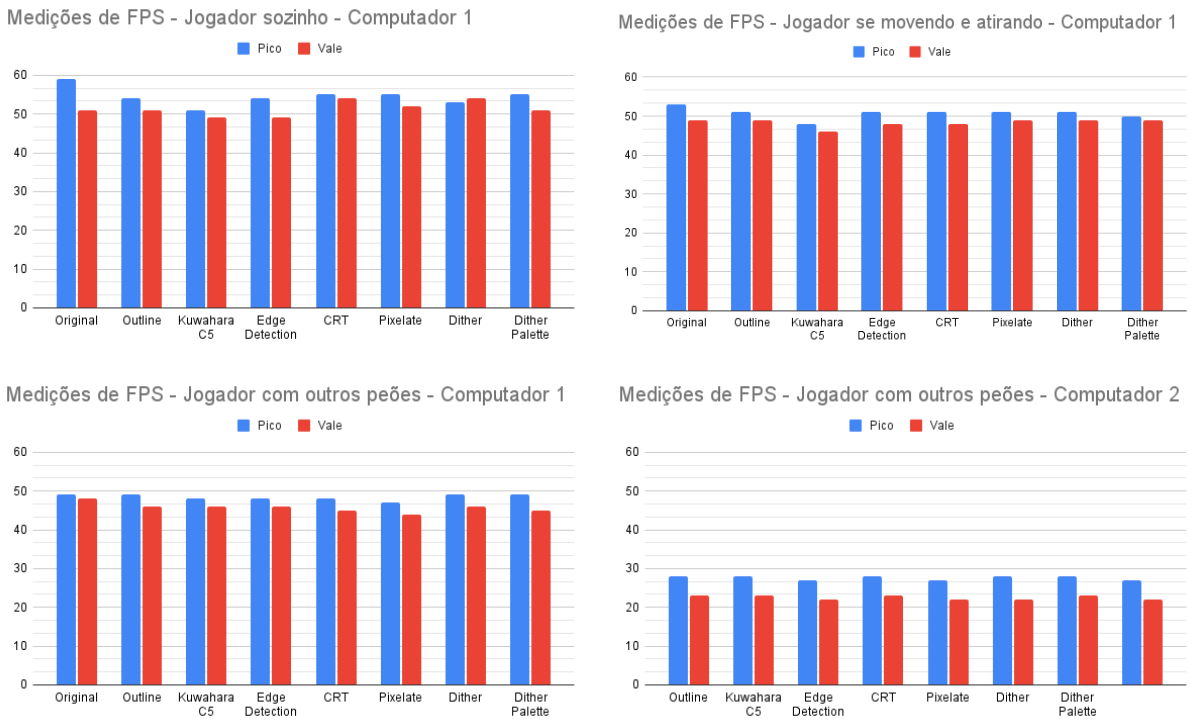


Figura 45 – Medições de FPS feitas no *Sample Game* "Lyra".

presença ou não dos efeitos de pós-processamento se tornou menos relevante para as taxas de Frames por Segundo medidas, que ficaram em torno de 46 FPS. Isso é um indicativo claro que **quando o jogador interage com outros peões nesse jogo, o gargalo de desempenho está na *thread* da CPU e não da GPU, muito provavelmente pela execução da lógica do comportamento dos outros peões**.

5.3.2.2 Resultados das medições de Pós-Processamento

Na Figura 46 estão indicados as medições do tempo gasto em execução de código de pós-processamento de cada shader na *thread* da GPU. Nesse caso, os únicos *Shaders* impactaram consideravelmente no tempo de pós-processamento foram o Kuwahara Filter e o Dither Palette, que utilizaram uma média de 2,01 ms e 0,56 ms do tempo na *thread*. Como o Kuwahara ultrapassou o tempo de computação de 1 ms, ele foi o único efeito que não atendeu ao requisito de desempenho descrito no capítulo 4.2.2

5.3.3 Tests no *StackOBot*

Para os testes com esse *Sample Game*, alguns ajustes foram necessários. Os primeiros testes apontavam um desempenho base muito pior que o visto no projeto anterior "Lyra", chegando a estabilizar em cerca de 22 FPS. Esse fato chamou atenção uma vez que o "StackOBot"é um jogo muito mais simples, focado em *puzzles* em um mapa pequeno e sem adição de *multiplayer* ou peões controlados pelo computador.

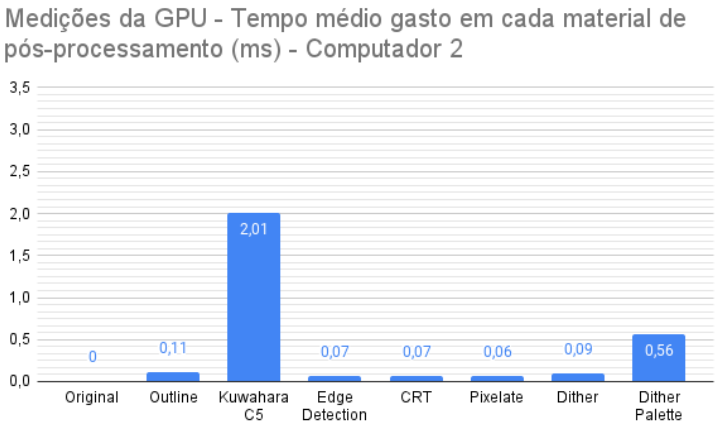


Figura 46 – Tempo médio gasto em pós-processamento na *thread* da GPU, medido em milisegundos.

Uma análise rápida mostrou que o gargalo estava na *thread* da GPU, especificamente nas seções de *pre pass* e *base pass* da renderização. Foram verificados quais eram os elementos mais custosos em cena utilizando a ferramenta de verificação de *Shader Complexity* na própria Unreal Engine. Assim, foram removidos esses elementos, que se tratavam de elementos visuais que demandavam muitas *draw calls* (rochas e vegetação) e efeitos de partículas de fumaça. Como esses elementos são puramente visuais, eles não afetariam os testes de jogabilidade. Com as mudanças feitas, o projeto pôde ser executado com ciclos atingido cerca de 70 FPS.

Para os resultados a seguir, considere que todos os testes foram feitos somente no Computador 1.

5.3.3.1 Resultados das medições de FPS

Na Figura 47 é possível ver os resultados das medições de FPS. O primeiro detalhe que é possível perceber é que esse projeto alcança valores de FPS mais altos que os alcançados nos testes no "Lyra". Além disso, há uma diferença maior entre os valores de pico e vale comparados ao projeto testado anteriormente. Foi averiguado que esses diferenças se davam por conta do ponto de vista do jogador. Ao renderizar cenas com terrenos mais complexos, os valores de FPS abaixavam. Isso demonstra que ainda haveria ainda mais espaço para otimização dos *assets* utilizados no projeto do *Sample Game*.

Do ponto de vista das taxas de Frame por Segundo, todos os efeitos se comportaram adequadamente e cumpriram com os requisitos alvo de desempenho de manter o jogo executando a 60 FPS. Novamente, o efeito do Kuwahara Filter teve o pior desempenho dentro os *Shaders* medidos.

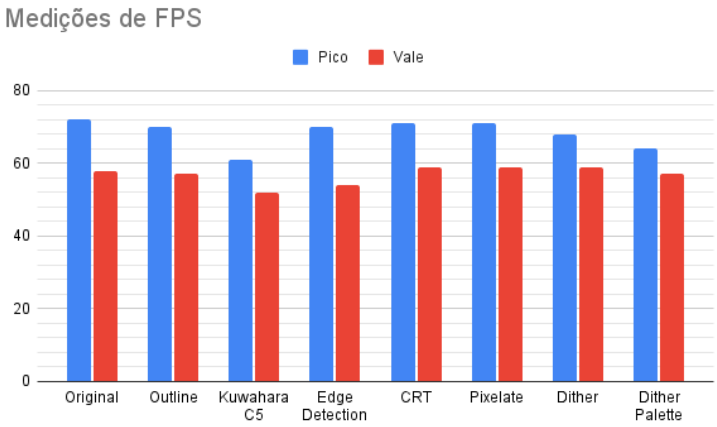


Figura 47 – Medições de FPS feitas no *Sample Game* "StackOBot".

5.3.3.2 Resultados das medições de Pós-Processamento

No "StackOBot", observou-se o tempo médio gasto de todo a pipeline de pós-processamento usando cada *Shader* desenvolvido na Figura 48, ao contrário das medidas de tempo gasto somente nos *Shaders* no "Lyra". Como o projeto "StackOBot"usa muito pouco pós-processamento base, ainda é possível ver o impacto dos diferentes efeitos na GPU.

Aqui, assim como no "Lyra", é verificável que o efeito "Kuwahara Filter"não cumpre o requisito de manter o jogo com um impacto abaixo de 1 ms na tarefa de pós-processamento. Todos outros efeitos, no entanto, se adéquam às medições estabelecidas e mantém um impacto baixo na *thread* da GPU.

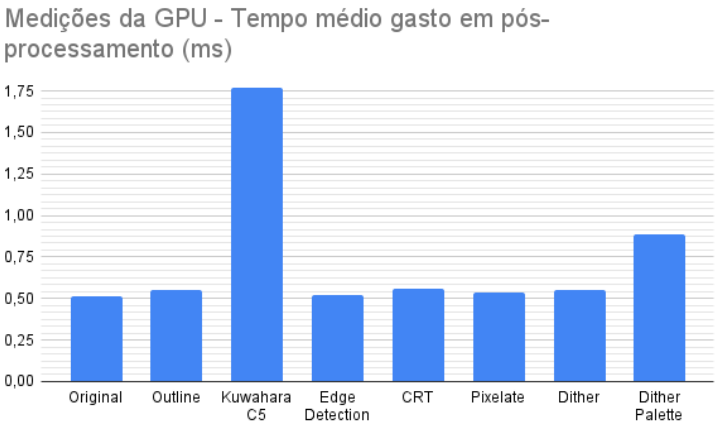


Figura 48 – Tempo médio gasto em pós-proecessamento na *thread* da GPU, medido em milisegundos.

6 Conclusão

6.0.1 Resultados do Trabalho

Segundos as métricas estabelecidas inicialmente no trabalho, 5 dos efeitos visuais produzidos apresentaram um comportamento adequado. O Kuwahara Filter foi o que teve o pior desempenho, mas ainda sim seu impacto na taxa de Frames não foi significativo em nenhum caso. Além disso, ele também não cumpriu totalmente o requisito de desempenho especificado no capítulo 4.2.2, levando um tempo base de pós-processamento maior que 1 ms em cada um dos *Sample Games*.

Com isso, é possível dizer os *Shaders* produzidos podem ser utilizados testes iniciais de efeitos visuais, de maneira que desenvolvedores podem aplicá-los sem a necessidade de gastar o tempo de um artista técnico. Além disso, o desempenho de vários *Shaders* se mostrou suficiente para servir como um conteúdo que pode ser aplicado numa implementação final de um projeto de menor porte, sem preocupações em relação aos impactos na jogabilidade. Por fim, a ferramenta criada é também um exemplo de uso de HLSL dentro da Unreal Engine, que é uma forma menos comum de criar *Materials* em comparação ao uso apenas do *script* visual. Desta forma, este trabalho pode servir também como fonte de estudo dessa forma de abordagem.

Como o *Asset Pack* ainda não foi divulgado ao público geral no tempo de escrita deste texto, ainda não foi possível medir que impacto o projeto teve ao tentar atender os desenvolvedores que necessitassem de efeitos de pós-processamento na Unreal Engine 5.

6.0.2 Possíveis Melhorias e Trabalhos Futuros

Esse trabalho deixa espaço para futuros aprimoramentos no projeto, como a melhoria de desempenho nos filtros relacionados ao Kuwahara Filter, bem como a produção de novos efeitos visuais. Além disso, uma pesquisa de campo voltada à utilização do *Asset Pack* em projetos reais também pode ser um caminho plausível para a continuidade do trabalho desenvolvido. Por fim, outras metodologias de testes, com diferentes parâmetros de desempenho poderiam ser realizados. Alguns exemplos seriam medir a utilização de dois efeitos de pós-processamento em conjunto ou a análise do mesmo *Shader* modificando seus parâmetros.

6.0.3 Considerações Finais

O desenvolvimento deste projeto desempenhou um papel fundamental no enriquecimento da formação dos alunos do curso de Engenharia de Computação. Ao engajarem-se

na criação de *Shaders* de pós-processamento utilizando a Unreal Engine 5 e a linguagem HLSL, os estudantes ganharam uma compreensão aprofundada da renderização em jogos digitais e da arquitetura de GPUs modernas. Este projeto proporcionou uma oportunidade prática para aplicar os conhecimentos teóricos adquiridos em sala de aula, incentivando a pesquisa e o estudo autodirigido.

Além disso, a avaliação do desempenho dos efeitos em projetos reais por meio dos *Sample Games* da Epic Games permitiu que os alunos vissem diretamente o impacto prático de seu trabalho. A medição da quantidade de frames renderizados por segundo e o monitoramento do impacto na *thread* da GPU forneceram uma perspectiva valiosa sobre as considerações práticas e otimizações necessárias no desenvolvimento de jogos. Em última análise, este projeto não apenas aprimorou as habilidades técnicas dos alunos, mas também os preparou para enfrentar desafios do mundo real na indústria de jogos e computação gráfica.

Referências

ALVES, L. R. G. Estado da arte dos games no brasil: trilhando caminhos. Universidae Católica Portuguesa, 2008. Citado na página 13.

ANGEL, E.; SHREINER, D. *Interactive Computer Graphics: A Top-Down Approach with Shader-Based OpenGL*. 6th. ed. USA: Addison-Wesley Publishing Company, 2011. ISBN 0132545233. Citado na página 15.

BARTYZEL, K. Adaptive kuwahara filter. *Signal, image and video processing*, Springer, v. 10, p. 663–670, 2016. Citado na página 23.

BAYER, B. *An optimum method for two-level rendition of continuous-tone pictures*. 1973. Disponível em: <https://web.archive.org/web/20130512190753/http://white.stanford.edu/~brian/psy221/reader/Bayer.1973.pdf>. Acesso em: 17 sep 2023. Citado na página 37.

DECAUDIN, P. Cartoon-looking rendering of 3d-scenes. *Syntim Project Inria*, Citeseer, v. 6, n. 4, 1996. Citado na página 41.

ENDESGA. *Endesga-32 Color Palette*. 2020. Disponível em: <https://lospec.com/palette-list/endesga-32>. Acesso em: 05 dec 2023. Citado na página 40.

ENGELMANN, A. *ASCII Post Process Material*. 2018. Disponível em: <https://www.unrealengine.com/marketplace/en-US/product/ascii-post-process-material>. Citado na página 12.

EPIC GAMES. *Assets and Content Packs in Unreal Engine*. [S.l.]. Disponível em: <https://docs.unrealengine.com/5.0/en-US/assets-and-content-packs-in-unreal-engine/>. Citado na página 21.

EPIC GAMES. *Lyra Sample Game*. [S.l.]. Disponível em: <https://docs.unrealengine.com/5.1/en-US/lyra-sample-game-in-unreal-engine/>. Citado na página 27.

EPIC GAMES. *Marketplace Guidelines - Unreal Engine 5*. [S.l.]. Disponível em: <https://www.unrealengine.com/pt-BR/marketplace-guidelines>. Citado na página 32.

EPIC GAMES. *Material Editor User Guide*. [S.l.]. Disponível em: <https://docs.unrealengine.com/5.0/en-US/unreal-engine-material-editor-user-guide/>. Citado na página 11.

EPIC GAMES. *Materials in Unreal Engine*. [S.l.]. Disponível em: <https://docs.unrealengine.com/5.1/en-US/unreal-engine-materials/>. Citado na página 17.

EPIC GAMES. *Post Process Effects in Unreal Engine*. [S.l.]. Disponível em: <https://docs.unrealengine.com/5.0/en-US/post-process-effects-in-unreal-engine/>. Citado 3 vezes nas páginas 11, 15 e 31.

EPIC GAMES. *Stylized Materials Pack*. [S.l.]. Disponível em: <https://www.unrealengine.com/marketplace/en-US/product/stylized-materials-pack>. Citado na página 29.

EPIC GAMES. *Unreal Engine 5*. [S.l.]. Disponível em: <https://www.unrealengine.com/en-US/unreal-engine-5>. Citado na página 11.

EPIC GAMES. *Unreal Engine Marketplace*. [S.l.]. Disponível em: <https://www.unrealengine.com/marketplace/en-US/store>. Citado 2 vezes nas páginas 12 e 21.

ESENIN, A. *Dithering Shader*. 2022. Disponível em: <https://www.unrealengine.com/marketplace/en-US/product/dithering-shader>. Acesso em: 19 mar 2023. Citado na página 12.

FOLEY, J. D. *Computer graphics: principles and practice*. [S.l.]: Addison-Wesley Professional, 1996. v. 12110. Citado na página 17.

FORTIM, I. *Pesquisa da indústria brasileira de games 2022*. 2022. Disponível em: <https://censojogosdigitais.com.br/pesquisa-da-industria-brasileira-de-games-2022/>. Citado na página 13.

GREGORY, J. *Game engine architecture*. [S.l.]: crc Press, 2018. Citado 2 vezes nas páginas 19 e 20.

ILETT, D. *Ultra Effects | Part 9 - Obra Dithering*. 2020. Disponível em: <https://danielilett.com/2020-02-26-tut3-9-obra-dithering/>. Citado na página 11.

KHRONOS GROUP. *Portal:OpenGL Objects/Renderbuffer Objects*. [S.l.], 2014. Disponível em: <https://www.khronos.org/opengl/wiki/Portal:OpenGL_Objects/Renderbuffer_Objects>. Citado na página 18.

KHRONOS GROUP. *Rendering Pipeline Overview*. [S.l.], 2022. Disponível em: <https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview>. Citado na página 16.

KUWAHARA, M. et al. Processing of ri-angiocardiographic images. *Digital processing of biomedical images*, Springer, p. 187–202, 1976. Citado 2 vezes nas páginas 39 e 40.

KYPRIANIDIS, J. E.; KANG, H.; DÖLLNER, J. Image and video abstraction by anisotropic kuwahara filtering. In: WILEY ONLINE LIBRARY. *Computer Graphics Forum*. [S.l.], 2009. v. 28, n. 7, p. 1955–1963. Citado na página 40.

LAUKKANEN, S. Post-processing in video games. Turun ammattikorkeakoulu, 2018. Citado na página 11.

MASUCH, M.; RÖBER, N. Game graphics beyond realism: Then, now and tomorrow. In: *Level UP: digital games research conference. DIGRA, Faculty of Arts, University of Utrecht*. [S.l.: s.n.], 2004. Citado na página 22.

OP1. *uma foto desfocada de uma pessoa segurando um skate*. 2023. Disponível em: <https://unsplash.com/pt-br/fotografias/uma-foto-desfocada-de-uma-pessoa-segurando-um-skate-28JTzr8cok8>. Acesso em: 21 sep 2023. Citado na página 38.

PAPARI, G.; PETKOV, N.; CAMPISI, P. Artistic edge and corner enhancing smoothing. *IEEE transactions on image processing*, IEEE, v. 16, n. 10, p. 2449–2462, 2007. Citado 2 vezes nas páginas 23 e 41.

PATHAK, S. K.; CHOUDHARY, A. An implementation of post processing technique in 3d graphics. In: IEEE. *2014 International Conference on Computing for Sustainable Global Development (INDIACom)*. [S.l.], 2014. p. 85–88. Citado na página 18.

POPE, L. *Return of the Obra Dinn*. [S.l.]: 3909 LLC, 2019. Citado na página 12.

SCANLINES. *scanlines, using CRT switchres with RetroArch on a VGA CRT monitor*. 2019. Disponível em: <https://www.reddit.com/r/crtgaming/comments/egef84/some_dummy_thicc_scanlines_using_crt_switchres/>. Acesso em: 20 sep 2023. Citado na página 26.

STROTHOTTE, T.; SCHLECHTWEG, S. *Non-photorealistic computer graphics: modeling, rendering, and animation*. [S.l.]: Morgan Kaufmann, 2002. Citado na página 21.

TOUSSAINT, D. *Comparison of different dithering algorithms (example: The statue of David by Michelangelo)*. 2006. Disponível em: <https://commons.wikimedia.org/wiki/File:Dithering_algorithms.png>. Acesso em: 20 sep 2023. Citado na página 22.

VARCHOLIK, P. *Real-time 3D rendering with DirectX and HLSL: A practical guide to graphics programming*. [S.l.]: Addison-Wesley Professional, 2014. Citado na página 35.

VASSEUR, T. *Art Design Deep Dive: Using a 3D pipeline for 2D animation in Dead Cells*. 2018. Disponível em: <https://www.gamedeveloper.com/production/art-design-deep-dive-using-a-3d-pipeline-for-2d-animation-in-i-dead-cells-i->. Citado 2 vezes nas páginas 11 e 25.

VRIES, J. de. *Learn OpenGL*. 2014. Disponível em: <https://learnopengl.com/Getting-started/Shaders>. Citado na página 11.

WINKENBACH, G.; SALESIN, D. H. Computer-generated pen-and-ink illustration. In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. [S.l.: s.n.], 1994. p. 91–100. Citado na página 24.

YLILUOMA, J. *Joel Yliluoma's arbitrary-palette positional dithering algorithm*. 2011. Disponível em: <https://bisqwit.iki.fi/story/howto/dither/jy/>. Acesso em: 19 mar 2023. Citado na página 37.

ZOETHOUT, K.; JAGER, W.; MOLLEMAN, E. Task dynamics in self-organising task groups: expertise, motivational, and performance differences of specialists and generalists. *Autonomous Agents and Multi-Agent Systems*, Springer, v. 16, p. 75–94, 2008. Citado na página 13.

Apêndices

APÊNDICE A – Código dos *Shaders* produzidos

A.1 Normal Outline

```
// INPUTS:
// UV: Texture coordinate of the rendered scene image
// InvSize: a float2 that represents 1/Size of the scene (this value will change
    depending on the window size)
// LineThickness: Thickness of the outline
// OutlineColor: Color of the outline
// SceneTexture: color of each pixel of the Scene

// OBS: InvSize is 1/ImageSize

float3 NormalOutlineFunc(float2 UV, float InvSize, float LineThickness, float3
    OutlineColor, float3 SceneTexture)
{
    float2 right_pixel_position = UV + (float2(1.0, 0.0) * InvSize * LineThickness);
    float2 left_pixel_position = UV + (float2(-1.0, 0.0) * InvSize * LineThickness);
    float2 up_pixel_position = UV + (float2(0.0, -1.0) * InvSize * LineThickness);
    float2 down_pixel_position = UV + (float2(0.0, 1.0) * InvSize * LineThickness);

    float3 right_pixel_normal = SceneTextureLookup(right_pixel_position, 8, false);
    float3 left_pixel_normal = SceneTextureLookup(left_pixel_position, 8, false);
    float3 up_pixel_normal = SceneTextureLookup(up_pixel_position, 8, false);
    float3 down_pixel_normal = SceneTextureLookup(down_pixel_position, 8, false);

    float3 horizontal_subtraction = abs(right_pixel_normal - left_pixel_normal);
    float3 vertical_subtraction = abs(up_pixel_normal - down_pixel_normal);

    float3 normal_sum = horizontal_subtraction + vertical_subtraction;

    float max_normal_sum = max(normal_sum[0], max(normal_sum[1],
        normal_sum[2]));
```

```
    float3 alpha = smoothstep(0.2, 0.7, max_normal_sum);

    return lerp(SceneTexture, OutlineColor, alpha);
}
```

A.2 Distance Outline

```
// INPUTS:
// SceneTexture: color of each pixel of the Scene
// UV: Texture coordinate of the rendered scene image
// InvSize: a float2 that represents 1/Size of the scene (this value will change
    depending on the window size)
// LineThickness: Thickness of the outline

// OBS: InvSize is 1/ImageSize

float3 DistanceOutlineFunc(float4 SceneTexture, float2 UV, float InvSize, float
    LineThickness)
{
    float3 original_color = SceneTextureLookup(UV, 1, false);

    float clamped_depth = clamp(original_color[0], 100, 600);

    float perc = (clamped_depth - 100) / (1000 - 100);
    float ranged_depth = perc * (2 - 8) + 8;

    float2 right_pixel_position = UV + (float2(1.0, 0.0) * InvSize * LineThickness);
    float2 left_pixel_position = UV + (float2(-1.0, 0.0) * InvSize * LineThickness);
    float2 up_pixel_position = UV + (float2(0.0, -1.0) * InvSize * LineThickness);
    float2 down_pixel_position = UV + (float2(0.0, 1.0) * InvSize * LineThickness);

    float3 right_pixel_color = SceneTextureLookup(right_pixel_position, 1, false);
    float3 left_pixel_color = SceneTextureLookup(left_pixel_position, 1, false);
    float3 up_pixel_color = SceneTextureLookup(up_pixel_position, 1, false);
    float3 down_pixel_color = SceneTextureLookup(down_pixel_position, 1, false);

    float3 subtract_color_right = original_color - right_pixel_color;
    float3 subtract_color_left = original_color - left_pixel_color;
```

```
    float3 subtract_color_up = original_color - up_pixel_color;
    float3 subtract_color_down = original_color - down_pixel_color;

    if(subtract_color_right[0] + subtract_color_left[0] + subtract_color_up[0] +
        subtract_color_down[0] > 25.0)
        return float(1.0);

    return float(0.0);
}
```

A.3 Kuwahara Filter

```
// INPUTS:
// SceneTexture: Not explicity used in code, but necessary for the
    SceneTextureLookup function from Unreal's Material Compiler (check
    MaterialCompiler.h in Runtime Module)
// UV: Texture coordinate of the rendered scene image
// Radius: the radius of the circle
// ViewSize: the size of the viewport

float3 KuwaharaSimpleCircleFunc(float4 SceneTexture, float2 UV, float Radius, float2
    ViewSize)
{
    // These values will store the sum of the color (R, G, B) of the pixels in the
        Top-Left, Bottom-Left, Top-Right and Bottom-Right quadrants,
        respectively
    float3 sum_color_tl = {0, 0, 0};
    float3 sum_color_bl = {0, 0, 0};
    float3 sum_color_tr = {0, 0, 0};
    float3 sum_color_br = {0, 0, 0};

    // These values will store the sum of the color squared of the pixels in each
        quadrant
    // We'll use this later to calculate the variance and the standard deviation
    float3 sum_color_sqrd_tl = {0, 0, 0};
    float3 sum_color_sqrd_bl = {0, 0, 0};
    float3 sum_color_sqrd_tr = {0, 0, 0};
    float3 sum_color_sqrd_br = {0, 0, 0};
```

```
// These values will store the mean color of each quadrant
float3 mean_color_tl;
float3 mean_color_bl;
float3 mean_color_tr;
float3 mean_color_br;

// These values will store the variance of each quadrant
float3 variance_tl;
float3 variance_bl;
float3 variance_tr;
float3 variance_br;

// Number of pixels in a quadrant
// Will calculated dynamically, as this number may change dependin on the
    chosen radius
float n_pixels = 0;

// The final value calculated: the color of this pixel!
float3 return_color;

// Calculate Top-Left mean color and standard deviation

// iterates from {0,0} to {Radius, Radius} in a quadrant
for(int i = 0; i <= Radius; i++)
{
    for(int j = 0; j <= Radius; j++)
    {
        float2 pixel_position = float2(i,j) + float2(-Radius, -Radius); //
            offsets to Top-Left
        float2 position_uv = UV + pixel_position/ViewSize;    // divides by
            ViewSize so we can adjust to the user's window/screen size
        if(length(UV - position_uv) <= Radius)
        {
            float3 color = SceneTextureLookup(position_uv, 14, false); //
                samples color from this UV coordinate in Post-Process scene
                texture (which has id = 14)
            sum_color_tl += color;
            sum_color_sqrd_tl += color * color;
            n_pixels++;                    // we'll only calculate this once!
```

```
            }
        }
    }

    mean_color_tl = sum_color_tl / n_pixels;
    variance_tl = (sum_color_sqrd_tl / n_pixels) - (mean_color_tl *
        mean_color_tl);

    // Calculate Bottom-Left mean color and standard deviation
    for(int i = 0; i <= Radius; i++)
    {
        for(int j = 0; j <= Radius; j++)
        {
            float2 pixel_position = float2(i,j) + float2(-Radius, 0);   // offsets to
                Bottom_Left
            float2 position_uv = UV + pixel_position/ViewSize;
            if(length(UV - position_uv) <= Radius)
            {
                float3 color = SceneTextureLookup(position_uv, 14, false);
                sum_color_bl += color;
                sum_color_sqrd_bl += color * color;
            }
        }
    }

    mean_color_bl = sum_color_bl / n_pixels;
    variance_bl = (sum_color_sqrd_bl / n_pixels) - (mean_color_bl *
        mean_color_bl);

    // Calculate Top-Right mean color and standard deviation
    for(int i = 0; i <= Radius; i++)
    {
        for(int j = 0; j <= Radius; j++)
        {
            float2 pixel_position = float2(i,j) + float2(0, -Radius);   // offsets to
                Top-Right
            float2 position_uv = UV + pixel_position/ViewSize;
            if(length(UV - position_uv) <= Radius)
            {
```

```
                float3 color = SceneTextureLookup(position_uv, 14, false);
                sum_color_tr += color;
                sum_color_sqrd_tr += color * color;
            }
        }
    }

    mean_color_tr = sum_color_tr / n_pixels;
    variance_tr = (sum_color_sqrd_tr / n_pixels) - (mean_color_tr *
        mean_color_tr);

    // Calculate Bottom-Right mean color and standard deviation
    for(int i = 0; i <= Radius; i++)
    {
        for(int j = 0; j <= Radius; j++)
        {
            float2 pixel_position = float2(i,j) + float2(0, 0);    // 'offsets' to
                Bottom-Right
            float2 position_uv = UV + pixel_position/ViewSize;
            if(length(UV - position_uv) <= Radius)
            {
                float3 color = SceneTextureLookup(position_uv, 14, false);
                sum_color_br += color;
                sum_color_sqrd_br += color * color;
            }
        }
    }

    mean_color_br = sum_color_br / n_pixels;
    variance_br = (sum_color_sqrd_br / n_pixels) - (mean_color_br *
        mean_color_br);

    // Now that we have all standard deviations and mean colors, we choose the mean
        color of the quadrant which the lowest variance
    float3 means[4] = {mean_color_tl, mean_color_bl, mean_color_tr,
        mean_color_br};
    float3 variances[4] = {variance_tl, variance_bl, variance_tr, variance_br};
```

```
    float min = 3; //will store the min value the sum of all RGB channels of
        variance. Starts off with the max value possible (i.e white = {1, 1, 1})

    for(int i = 0; i < 4; i++)
    {
        float variance_channels_sum = variances[i].r + variances[i].g + variances[i].
            b;
        if(variance_channels_sum < min)
        {
            min = variance_channels_sum;
            return_color = means[i];
        }
    }

    return return_color;
}
```

A.4 Edge Detection

```
// INPUTS:
// SceneTexture: Not explicity used in code, but necessary for the
    SceneTextureLookup function from Unreal's Material Compiler (check
    MaterialCompiler.h in Runtime Module)
// UV: Texture coordinate of the rendered scene image
// InvSize: a float2 that represents 1/Size of the scene (this value will change
    depending on the window size)
// LineThickness: how thick the edge line should be
// EdgeColor: Color to paint edges
// FillColor: Color to paint everything besides the edges

// OBS: InvSize is 1/ImageSize

float3 EdgeDetectionFunc(float3 SceneTexture, float2 UV, float InvSize, float
    LineThickness, float3 EdgeColor, float3 FillColor)
{
    float2 right_pixel_position = UV + (float2(1.0, 0.0) * InvSize * LineThickness);
    float2 left_pixel_position = UV + (float2(-1.0, 0.0) * InvSize * LineThickness);
    float2 up_pixel_position = UV + (float2(0.0, -1.0) * InvSize * LineThickness);
    float2 down_pixel_position = UV + (float2(0.0, 1.0) * InvSize * LineThickness);
```

```
    float3 right_pixel_normal = SceneTextureLookup(right_pixel_position, 8, false);
    float3 left_pixel_normal = SceneTextureLookup(left_pixel_position, 8, false);
    float3 up_pixel_normal = SceneTextureLookup(up_pixel_position, 8, false);
    float3 down_pixel_normal = SceneTextureLookup(down_pixel_position, 8, false);

    float3 horizontal_subtraction = abs(right_pixel_normal - left_pixel_normal);
    float3 vertical_subtraction = abs(up_pixel_normal - down_pixel_normal);

    float3 normal_sum = horizontal_subtraction + vertical_subtraction;

    float max_normal_sum = max(normal_sum[0], max(normal_sum[1],
        normal_sum[2]));

    float3 alpha = smoothstep(0.2, 0.7, max_normal_sum);

    return lerp(FillColor, EdgeColor, alpha);
}
```

A.5 CRT Effect

```
// INPUTS:
// UV: Texture coordinate of the rendered scene image
// SceneTexture: Not explicity used in code, but necessary for the
    SceneTextureLookup function from Unreal's Material Compiler (check
    MaterialCompiler.h in Runtime Module)

float3 CRTFunc(float2 UV, float2 ViewSize, float2 PixelPosition, float curvature,
                    float rFrequency, float rPhase, float rAmplitude, float
                        rDisplacement,
                    float gFrequency, float gPhase, float gAmplitude, float
                        gDisplacement,
                    float bFrequency, float bPhase, float bAmplitude, float
                        bDisplacement
                    )
{
    #define PI 3.1415926
    #define SCREEN_WIDTH ViewSize.x
    #define SCREEN_HEIGHT ViewSize.y
```

```
static const int SceneTextureId = 14;

// Distortion
// float curvature = 5;
// float2 UV_scaled = UV * 2 - 1;
// float2 Offset = UV_scaled / curvature;
// float2 Offset2 = Offset * Offset;
// float2 UV_distorted_scaled = UV_scaled + UV_scaled * Offset2;
// float2 UV_distorted = UV_distorted_scaled * 0.5 + 0.5;
float2 UV_distorted = (UV - 0.5) * 2.0; // UV_distorted is now -1 to 1
UV_distorted = curvature*UV_distorted/sqrt(curvature*curvature -dot(
    UV_distorted, UV_distorted));
UV_distorted = (UV_distorted / 2.0) + 0.5; // back to 0-1 coords


// Vignette
// float2 UV_vignette = UV_distorted;
// UV_vignette *= 1- UV_vignette.yx;
// float vignette = UV_vignette.x * UV_vignette.y * 15;
// vignette = pow(vignette, 0.25);
float2 UV_vignette = (UV - 0.5) * 2.0; // UV_vignette is now -1 to 1
UV_vignette = curvature*UV_vignette/sqrt(curvature*curvature -dot(
    UV_vignette, UV_vignette));
UV_vignette = 1 - abs(UV_vignette);
float2 vignette = smoothstep(0, float2(0.1*SCREEN_WIDTH/SCREEN_WIDTH
    , 0.1*SCREEN_HEIGHT/SCREEN_HEIGHT), UV_vignette);
vignette = vignette.x * vignette.y;
vignette = pow(vignette, 5);


// Sample pixel color
// Side by side comparison
if (PixelPosition.x > SCREEN_WIDTH/2) {
float3 PixelColor = SceneTextureLookup(UV, SceneTextureId, 0).rgb;
// return PixelColor;
}
float3 PixelColor = SceneTextureLookup(UV_distorted, SceneTextureId, 0).rgb;
```

```
// float3 PixelColor = float3(length(vignette), length(vignette), length(vignette)
    );
// float3 PixelColor = float3(vignette.x, vignette.x, vignette.x);

// Black out areas where UV < 0
if (UV_distorted.x > 1 || UV_distorted.y > 1 || UV_distorted.x < 0 ||
    UV_distorted.y < 0) {
PixelColor = float3(0,0,0);
}


// Scan Lines
// float rFrequency = 2;
// float rPhase = PI/2;
// float rAmplitude = 0.15;
// float rDisplacement = 1.15;

// float gFrequency = 2;
// float gPhase = 0;
// float gAmplitude = 0.1;
// float gDisplacement = 1.1;

// float bFrequency = 2;
// float bPhase = PI/2;
// float bAmplitude = 0.15;
// float bDisplacement = 1.15;

float ScreenHeight = ViewSize.y;

PixelColor = float3((rDisplacement + rAmplitude*sin(UV.y*ScreenHeight*
    rFrequency + rPhase))*PixelColor.r,
            (gDisplacement + gAmplitude*sin(UV.y*ScreenHeight*gFrequency +
                gPhase))*PixelColor.g,
            (bDisplacement + bAmplitude*sin(UV.y*ScreenHeight*bFrequency +
                bPhase))*PixelColor.b);

// Apply vignette
PixelColor = PixelColor * float3(vignette.x,vignette.x,vignette.x);
```

```
    return PixelColor;
}
```

A.6 Pixelate

```
// INPUTS:
// UV: Texture coordinate of the rendered scene image
// ViewSize: Float2 resolution of game in pixels
// PixelPosition: Current Pixel Position
// PixelSize: Desired final pixelate size in pixels
// SceneTexture: Not explicity used in code, but necessary for the
    SceneTextureLookup function from Unreal's Material Compiler (check
    MaterialCompiler.h in Runtime Module)

float3 PixelateFunc(float2 UV, float2 ViewSize, float2 PixelPosition, float PixelSize)
{
    static const int SceneTextureId = 14;

    float2 Resolution = ViewSize / PixelSize;
    float2 PixelatePosition = PixelPosition / PixelSize;
    float2 PixelatePositionFloor = floor(PixelatePosition);
    float2 SceneUVFloor = UV * PixelatePositionFloor / PixelatePosition;

    // Luminosity of Scene Pixel
    float3 PixelColor = SceneTextureLookup(SceneUVFloor, SceneTextureId, 0).rgb;

    return PixelColor;
}
```

A.7 Dither

```
// INPUTS:
// UV: Texture coordinate of the rendered scene image
// ViewSize: Float2 resolution of game in pixels
// PixelPosition: Current Pixel Position
// SceneTexture: Not explicity used in code, but necessary for the
    SceneTextureLookup function from Unreal's Material Compiler (check
    MaterialCompiler.h in Runtime Module)
```

```
float3 DitherFunc(float2 UV, float2 ViewSize, float2 PixelPosition, Texture2D Tex,
    SamplerState TexSampler, float DitherSize, int bit_depth)
{
    static const int SceneTextureId = 14;

    float2 Resolution = ViewSize / DitherSize;
    float2 PixelPositionDither = PixelPosition / DitherSize;
    float2 PixelPositionDitherFloor = floor(PixelPositionDither);
    float2 SceneUVFloor = UV * PixelPositionDitherFloor / PixelPositionDither;
    UV = floor(UV * Resolution) / Resolution;

    // Luminosity of Scene Pixel
    float3 PixelColor = SceneTextureLookup(SceneUVFloor, SceneTextureId, 0).rgb;

    // Bayer pattern
    float2 TextureDim;
    Tex.GetDimensions(TextureDim.x, TextureDim.y);
    float2 PatternUV = int2(UV * Resolution) / TextureDim;
    float2 PixelUV = PixelPositionDitherFloor / TextureDim;
    float threshold = Texture2DSample(Tex, TexSampler, PixelUV).r;

    float threshold_ratioed = (threshold- 0.5) / 5.0;

    float3 PixelColorSum = PixelColor + threshold_ratioed;
    float3 PixelColorDepth = floor(PixelColorSum * float(bit_depth) + 0.5) / float(
        bit_depth);

    float3 result = PixelColorDepth;

    return result;
}
```

A.8 Dither Palette

```
// INPUTS:
// UV: Texture coordinate of the rendered scene image
// ViewSize: Float2 resolution of game in pixels
// PixelPosition: Current Pixel Position
```

```
// SceneTexture: Not explicity used in code, but necessary for the
    SceneTextureLookup function from Unreal's Material Compiler (check
    MaterialCompiler.h in Runtime Module)

float colorDistance(float3 color1, float3 color2) {
    float3 lumaMap = float3(0.299f, 0.587f, 0.114f);
    float luma1 = dot(color1, lumaMap);
    float luma2= dot(color2, lumaMap);
    float lumadiff = luma1-luma2;
    float3 diff = color1 - color2;
    float diff2 = dot(diff*diff, lumaMap);
    return diff2*0.75 + lumadiff*lumadiff;
}

float3 get_closest_color(float3 color, Texture2D DitherPalette, sampler
    DitherPaletteSampler, float ditherValue, int paletteSize) {
    float3 col;
    float x = 4;
    float3 threshold = float3(1/x, 1/x, 1/x);
    float3 colorThreshold = color + (ditherValue -0.5) * threshold;

    float least_penalty = 1e99;
    for(int index1 = 0; index1 < paletteSize; index1++) {
      float2 colorUV = float2(float(index1) /paletteSize, 0.5);
      float3 c = Texture2DSample(DitherPalette, DitherPaletteSampler, colorUV).rgb;

    float d = colorDistance(colorThreshold, c);

      if(d < least_penalty)
        {
            least_penalty = d;
            col = c;
        }
  }

    return col;
}
```

```
float3 DitherPaletteFunc(float2 UV, float2 ViewSize, float2 PixelPosition, Texture2D
    Tex, SamplerState TexSampler,
                        float DitherSize, Texture2D DitherPalette, SamplerState
                            DitherPaletteSampler, int PaletteSize)
{
    static const int SceneTextureId = 14;

    float2 Resolution = ViewSize / DitherSize;
    float2 PixelPositionDither = PixelPosition / DitherSize;
    float2 PixelPositionDitherFloor = floor(PixelPositionDither);
    float2 SceneUVFloor = UV * PixelPositionDitherFloor / PixelPositionDither;
    UV = floor(UV * Resolution) / Resolution;

    // Luminosity of Scene Pixel
    float3 PixelColor = SceneTextureLookup(SceneUVFloor, SceneTextureId, 0).rgb;

    // Bayer pattern
    float2 TextureDim;
    Tex.GetDimensions(TextureDim.x, TextureDim.y);
    float2 PatternUV = int2(UV * Resolution) / TextureDim;
    float2 PixelUV = PixelPositionDitherFloor / TextureDim;
    float ditherValue = Texture2DSample(Tex, TexSampler, PixelUV).r;

    float3 rgb = get_closest_color(PixelColor, DitherPalette, DitherPaletteSampler,
        ditherValue, PaletteSize);

    return rgb;
}
```