PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
ESCOLA POLITÉCNICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

EMPREGO DE TÉCNICAS E ALGORITMOS PARA DESENVOLVIMENTO DE JOGOS REALISTAS

GABRIEL HÚGLIO PINELI SIMÕES

GOIÂNIA
2023

GABRIEL HÚGLIO PINELI SIMÕES

EMPREGO DE TÉCNICAS E ALGORITMOS PARA DESENVOLVIMENTO DE JOGOS REALISTAS

Trabalho de Conclusão de Curso apresentado à Escola Politécnica, da Pontifícia Universidade Católica de Goiás como parte dos requisitos para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador(a): Prof. Me. Max Gontijo de Oliveria

GOIÂNIA

2023

GABRIEL HÚGLIO PINELI SIMÕES

EMPREGO DE TÉCNICAS E ALGORITMOS PARA DESENVOLVIMENTO DE JOGOS REALISTAS

Este Trabalho de Conclusão de Curso julgado adequado para obtenção do título de Bacharel em Ciência da Computação, e aprovado em sua forma final pela Escola Politécnica da Pontifícia Universidade Católica de Goiás, em ____/____/_____.

_____

Profª. Me. Ludmilla Reis Pinheiro dos Santos

Coordenador(a) de Trabalho de Conclusão de Curso

Banca examinadora:

_____

Orientador: Prof. Me. Max Gontijo

_____

Prof. Me. Fernando Gonçalves Abadia

_____

Prof. Me. Rafael Leal Martins

GOIÂNIA

2021

Dedico este trabalho a mim mesmo, reconhecendo o esforço e a dedicação investidos. Que este seja um lembrete constante de que as ambições podem se tornar realidade, e os sonhos podem ser concretizados. Acredite na sua capacidade ilimitada e saiba que o caminho para o sucesso está pavimentado com estudo contínuo e esforço incansável. Que cada conquista, por menor que seja, seja celebrada como um passo em direção a grandes realizações. Desejo a mim mesmo toda a sorte e prosperidade, tanto na vida pessoal quanto na carreira que está por vir. Que este seja apenas o início de uma jornada repleta de crescimento e conquistas extraordinárias.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer à Inteligência Suprema, Causa Primária De Todas As Coisas.

Agradeço imensamente à minha família: ao meu irmão Lucca e especialmente à minha mãe Luciana e ao meu pai Fábio por sempre me apoiarem durante minha jornada acadêmica, sempre me aconselhando e desejando o meu bem. Sem a presença e o suporte de vocês minha formação, esse trabalho e perspectivas futuras não seriam uma realidade.

Agradeço a meus avôs e avós por todo o apoio e por sempre se orgulharem de minhas conquistas e vitórias ao longo de meu constante desenvolvimento. Assim como agradeço muito aos meus avós maternos pelo grande presente que facilitou muito a locomoção para a faculdade.

Agradeço imensamente à minha namorada Letícia pelo apoio, suporte e paciência que me proporcionou durante minha formação. Espero continuarmos crescendo juntos, que a nossa parceria continue a nos fortalecer enquanto buscamos realizar nossas ambições. Te amo.

Agradeço ao meu professor, Max Gontijo de Oliveira pelas orientações e apoio ao longo do trabalho desenvolvido, apesar de minha falta de cordialidade quanto à prazos.

Também agradeço aos bons professores da PUC por me ensinarem e proporcionarem conhecimento e aprendizado. Aos maus professores agradeço por me mostrarem o que não devo me tornar.

Gostaria de expressar meu sincero agradecimento à professora e coordenadora Carmen por sua inestimável ajuda e constante apoio ao longo da minha formação. Sua dedicação na formação de bons estudantes reflete a excelente profissional que você é.

Por fim, gostaria de expressar meu profundo agradecimento ao estimado professor e grande amigo, Alexandre Ribeiro. Ele desempenhou um papel fundamental ao longo de toda a minha trajetória como estudante. Sou eternamente grato por suas orientações incisivas e puxões de orelha constantes para que eu continuasse nos estudos. Sua paixão pelo ensino é evidente, e mesmo diante da minha teimosia, reconheço que minha jornada acadêmica, na maratona de programação e minha jornada pessoal seriam completamente diferentes sem suas broncas. Expresso o desejo sincero de que você continue influenciando positivamente a formação de bons profissionais e cidadãos, mesmo quando, ocasionalmente, falte o apoio e incentivo necessários por parte da própria universidade. Sua dedicação e impacto positivo na vida dos alunos transcende as barreiras institucionais, e isso é verdadeiramente inspirador. Dito isso, te considero com certeza um dos melhores, se não o melhor, professor dessa universidade.

RESUMO

Este trabalho busca destacar a aplicação de uma variedade de técnicas e algoritmos no desenvolvimento de jogos, com o objetivo central de criar experiências imersivas e realistas para os jogadores. Durante o estudo, serão abordados diversos aspectos fundamentais, desde a concepção e configuração de armas até a implementação de animações, programação envolvendo o comportamento do jogador e do inimigo, interações dinâmicas com o ambiente, além da criação de mapas orgânicos através de algoritmos como Dijkstra e Kruskal. Neste contexto, destaca-se a criação de um projeto realista e envolvente, servindo como um cenário prático para a aplicação das técnicas e algoritmos discutidos. O intuito é não apenas fornecer um conjunto abrangente de práticas para a construção de jogos, mas também oferecer insights valiosos sobre estratégias que aprimoram a atmosfera e a jogabilidade. Ao compartilhar conhecimentos detalhados, espera-se que este trabalho se torne uma referência para aqueles interessados no desenvolvimento de projetos semelhantes, proporcionando uma base sólida para a criação de experiências de jogos cativantes e autênticas.

Palavras-chave: Unreal Engine, Desenvolvimento de jogos, Realismo.

ABSTRACT

This work aims to highlight the application of a variety of techniques and algorithms in the development of games, with the central goal of creating immersive and realistic experiences for players. Throughout the study, various fundamental aspects will be addressed, ranging from the conception and configuration of weapons to the implementation of animations, programming involving player and enemy behavior, dynamic interactions with the environment, and the creation of organic maps through algorithms such as Dijkstra and Kruskal. In this context, the creation of a realistic and engaging project stands out, serving as a practical scenario for the application of the discussed techniques and algorithms. The intention is not only to provide a comprehensive set of practices for building games but also to offer valuable insights into strategies that enhance atmosphere and gameplay. By sharing detailed knowledge, it is hoped that this work becomes a reference for those interested in developing similar projects, providing a solid foundation for creating captivating and authentic gaming experiences.

Keywords: Unreal Engine, Game Development, Realism.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A Indústria brasileira de games tem apresentado nos últimos anos duas características vitais que mostram um desenvolvimento robusto. Por um lado, a demanda por jogos no Brasil manteve um crescimento de 3% em 2022, contrastando com a queda global de 4,3% constatada neste mesmo ano, segundo dados da *Newzoo*. Por outro lado, o número de estúdios e de lançamento de empresas brasileiras no exterior tem crescido, assim como o interesse de estúdios e *publishers* internacionais em investir no Brasil. Já são mais de 1000 estúdios mapeados no país, e mais de 2600 jogos próprios ou desenvolvidos para terceiros lançados entre 2020 e 2022, sendo 1008 apenas nesse último ano (ABRAGAMES, 2023).

Receitas globais de videogames aumentaram nos últimos anos, de acordo com dados da IDC, tornando a indústria de videogames mais lucrativa do que a indústria cinematográfica global e esportiva norte-americana combinadas (WITKOWSKI, W., 2021).

Em decorrência de tal sucesso, os últimos anos vêm se caracterizando pela "ascensão dos *indies*", graças ao crescente número de desenvolvedores independentes ingressando e estabelecendo-se no mercado (PRESSANTO, 2021).

Considerando essa perspectiva, esse trabalho foi realizado com a finalidade de explorar e servir de futura pesquisa e orientação para desenvolvedores independentes que buscam fazer jogos realistas, apresentando a utilização de técnicas e algoritmos do meio de programação computacional e mostrando como esse uso contribui para o desenvolvimento.

Desenvolver um jogo representa um processo intricado que amalgama criatividade, habilidades técnicas e uma compreensão profunda da interação entre design e tecnologia. Ao longo dessa jornada, que vai desde a concepção da ideia até a entrega do produto final, os desenvolvedores se deparam com uma experiência desafiadora e entusiasmante, onde cada etapa se configura como uma oportunidade única para inovação e aprendizado.

A criação de jogos não apenas se destaca como uma expressão artística e criativa, mas também emerge como um desafio enriquecedor que impulsiona o desenvolvimento de habilidades cognitivas, lógicas e colaborativas. Além de ser um campo estimulante, o processo de desenvolvimento de jogos oferece aos criadores uma oportunidade única para explorar e aplicar técnicas avançadas. Ao incentivar a

imersão dos jogadores, os jogos não só proporcionam entretenimento, mas também servem como uma plataforma inovadora para a experimentação, contribuindo assim para o aprimoramento contínuo do setor de criação de jogos e suas diversas aplicações.

Assim, este trabalho propõe uma abordagem centrada na análise de técnicas e algoritmos aplicados ao desenvolvimento de um jogo realista do gênero *First Person Shooter* (FPS), especificamente situado em complexos sistemas de cavernas. É imperativo ressaltar que o jogo apresentado não assume a forma de um produto tangível decorrente deste Trabalho de Conclusão de Curso (TCC), mas antes se configura como um exemplo genérico e abstrato, concebido exclusivamente para fornecer um contexto para a aplicação e exploração das mencionadas técnicas e algoritmos.

1.1 OBJETIVOS

Expor as técnicas e algoritmos que foram pessoalmente aplicados no desenvolvimento de jogos de tiro em primeira pessoa (FPS), com a finalidade de estabelecer uma experiência visualmente realista e imersiva para o jogador. Além disso, busca-se explorar e apresentar as dificuldades específicas encontradas durante a concepção desses jogos digitais, destacando tanto os obstáculos superados quanto os resultados obtidos no processo.

Adicionalmente, existem objetivos mais delineados:

- Realizar testes, pesquisas e descobrir técnicas para a criação de animações que sejam realistas, orgânicas e responsivas.
- Estudar e desenvolver um método para a criação procedural de mapas.
- Documentar os desafios encontrados e as soluções desenvolvidas durante o processo de desenvolvimento.

1.2 ORGANIZAÇÃO TEXTUAL

A estrutura textual do presente trabalho segue uma organização cuidadosa. O Capítulo 2 oferece uma abrangente apresentação dos programas e conceitos empregados no desenvolvimento deste projeto. Além disso, no Subcapítulo 2.3 dedicado à *Unreal Engine*, são exploradas variadas funcionalidades e utilidades que

essa *engine* proporciona, todas destinadas a desempenhar um papel crucial nas fases subsequentes do trabalho.

Posteriormente à abordagem dos conceitos preliminares, o Capítulo 3, intitulado "O Jogo", delineia o desenvolvimento do projeto, estruturado em cinco etapas principais. A fase inicial consiste na concepção do *Game Design Document* (GDD) (Subcapítulo 3.1), onde são definidas os principais componentes que constituirão o jogo. As etapas subsequentes desdobram-se em três capítulos que documentam a criação das quatro partes essenciais do trabalho. O primeiro destes capítulos envolve a elaboração de modelos e programação de armas 3D (Capítulo 3.2), o segundo abarca a criação de animações e programação do personagem (Capítulo 3.3), enquanto o terceiro se dedica à concepção de um monstro (Capítulo 3.4), cuja origem e características foram previamente definidas no GDD. A última etapa compreende a criação de um mapa para o jogo, encerrando o ciclo de desenvolvimento de forma integral e coerente (Capítulo 3.5).

2 CONCEITOS PRELIMINARES

No presente capítulo, proporciona-se uma exposição abrangente de conceitos preliminares cruciais, acompanhada de explicações detalhadas acerca dos *softwares* utilizados neste estudo. O intuito é oferecer ao leitor uma compreensão substancial e aprofundada desses fundamentos essenciais, estabelecendo uma base sólida para a compreensão aprofundada do contexto teórico e prático que permeia a pesquisa em andamento.

2.1 BLENDER

O Blender é uma Suite de Criação 3D de Código Aberto gratuita que abrange todo o processo de modelagem e animação, incluindo modelagem, *rigging*, animação, simulação, renderização, composição e rastreamento de movimento. O programa também oferece recursos de Edição de Vídeo e Animação 2D com Grease Pencil (BLAIN, 2019).

Desenvolvido pela *Blender Foundation*, sua gratuidade para uso e distribuição confere-lhe uma acessibilidade notável, abrangendo uma vasta comunidade de artistas, animadores, designers e entusiastas 3D.

Entre suas múltiplas funcionalidades, aquelas mais frequentemente empregadas no desenvolvimento de jogos são a modelagem, o *rigging* e a animação. Estas ferramentas do Blender desempenham um papel crucial na materialização de elementos visuais interativos, proporcionando aos desenvolvedores recursos robustos para a criação e aprimoramento de ambientes tridimensionais dinâmicos e cativantes.

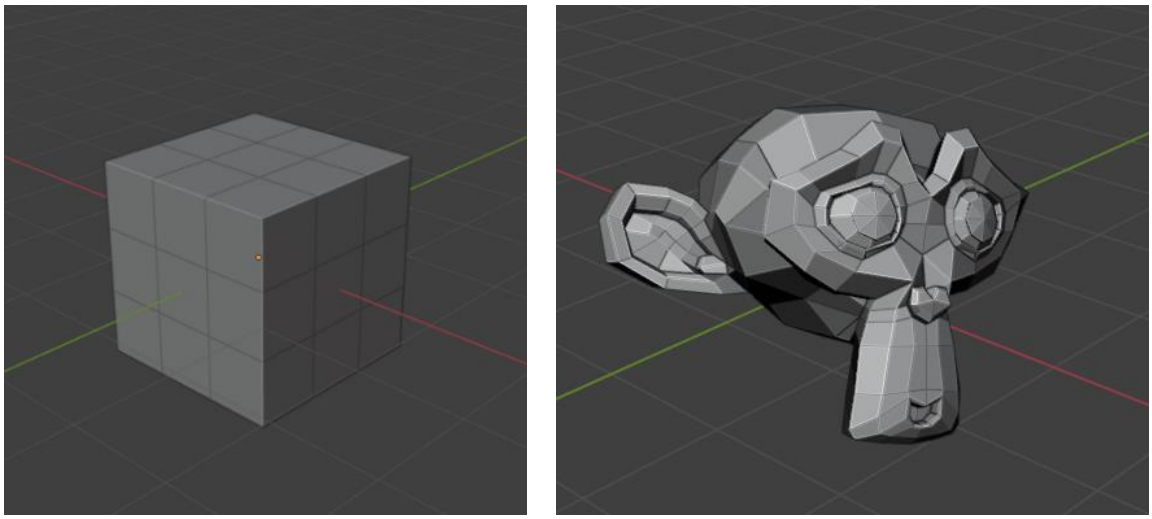
2.1.1 Modelagem

A modelagem, enquanto processo criativo, consiste na elaboração de representações tridimensionais de objetos ou cenários. Estas representações, uma vez concluídas, podem ser posteriormente importadas para uma *Engine* de Jogos, onde adquirem vida por meio de interações dinâmicas, animações envolventes e visualizações imersivas. Este estágio inicial e essencial do desenvolvimento visual contribui significativamente para a materialização de ambientes interativos, proporcionando a base estrutural para a experiência do usuário dentro do jogo.

2.1.1.1 Malha

No âmbito tridimensional, o termo "malha" refere-se a uma estrutura composta por vértices, arestas e faces, servindo como a base fundamental para a construção de modelos 3D, como pode ser visto na Figura 1. Essa malha desempenha um papel crucial ao definir a forma e a estrutura de um objeto, sendo a densidade de seus componentes, ou seja, os polígonos, determinante para a qualidade e o detalhamento do modelo resultante. Em suma, a malha constitui-se como um elemento central no processo de modelagem 3D, exercendo influência direta sobre a representação visual e a fidelidade do objeto dentro do espaço tridimensional.

Figura 1 - Modelos 3D no Blender



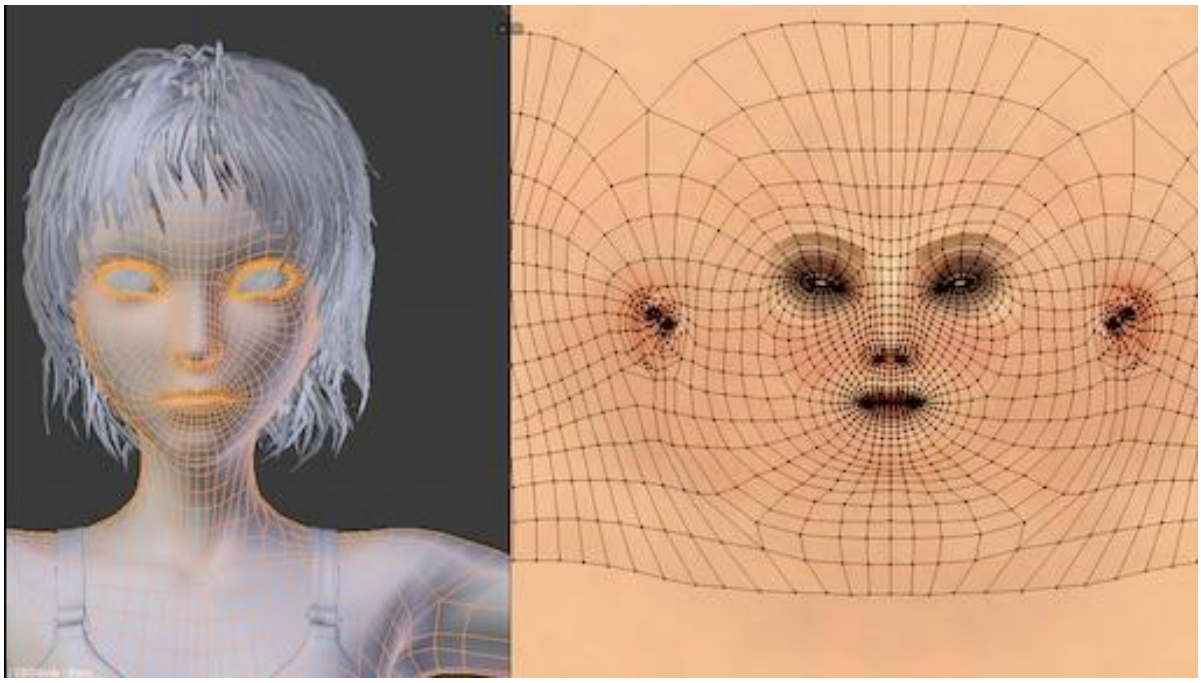
a) b)

Fonte: Autoria própria

2.1.1.2 UV Unwrap's

O UV *unwrapping* consiste no processo pelo qual a superfície tridimensional de um objeto é desdobrada e achatada em um plano bidimensional, resultando na criação de coordenadas de textura conhecidas como mapas UV, mostrado na Figura 2. Essas coordenadas desempenham um papel fundamental ao indicar como as texturas devem ser aplicadas às superfícies do objeto. Este procedimento é crucial na etapa de modelagem 3D, permitindo uma correspondência precisa entre a representação 2D das texturas e a geometria tridimensional do objeto.

Figura 2 - UV do rosto de uma personagem



Fonte: MEHRSHAD, 2022.

O UV *unwrapping* desempenha um papel crucial na minimização de distorções nas texturas quando aplicadas a um objeto tridimensional. A realização precisa do mapeamento UV é essencial, uma vez que distorções ou esticamentos nas texturas podem ocorrer nas superfícies do modelo se esse processo não for executado corretamente. Para objetos destinados à animação, o UV unwrapping é particularmente relevante, pois possibilita que as texturas acompanhem a animação de maneira suave, preservando a continuidade visual à medida que o objeto se move ou se deforma.

A execução bem-sucedida do UV *unwrapping* pode ser desafiadora, uma vez que demanda uma compreensão aprofundada da topologia da malha e das características específicas do objeto. Ferramentas e técnicas específicas, como a marcação de costuras ou a criação de cortes na malha, são empregadas para facilitar o desdobramento adequado das superfícies, contribuindo assim para o resultado visualmente preciso e coeso do modelo tridimensional.

2.1.2 Rigging

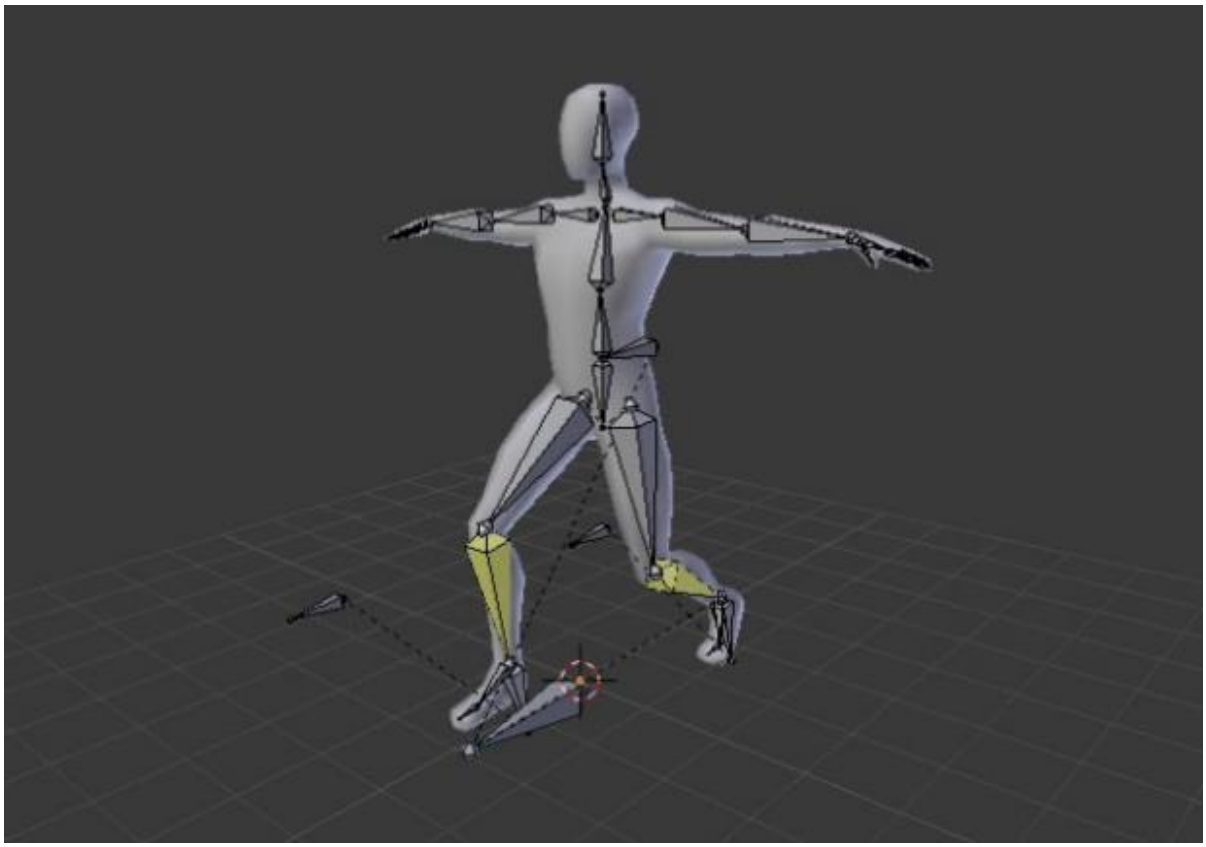
Depois de modelados os objetos que serão utilizados no jogo, deve-se fazer o *rigging*, um processo de adicionar um esqueleto virtual, conhecido como "*rig*" ou

armadura (“*Armature*” no blender) composta por 1 ou mais ossos, à um modelo tridimensional para permitir a animação eficiente e realista.

2.1.2.1 Armature

Esse esqueleto, composto por ossos virtuais interconectados, representa a estrutura anatômica do objeto 3D, evidenciado na Figura 3. A técnica de rigging desempenha um papel essencial ao possibilitar a articulação e movimentação controlada do modelo. Essa abordagem facilita a animação de personagens, objetos ou criaturas de maneira mais orgânica, permitindo a reprodução de movimentos naturais e até mesmo expressões faciais. Em suma, o rigging é um componente fundamental no processo de animação, conferindo uma dinâmica realista e fluidez aos elementos 3D, enriquecendo a experiência visual para o usuário.

Figura 3 - Esqueleto de personagem humanóide



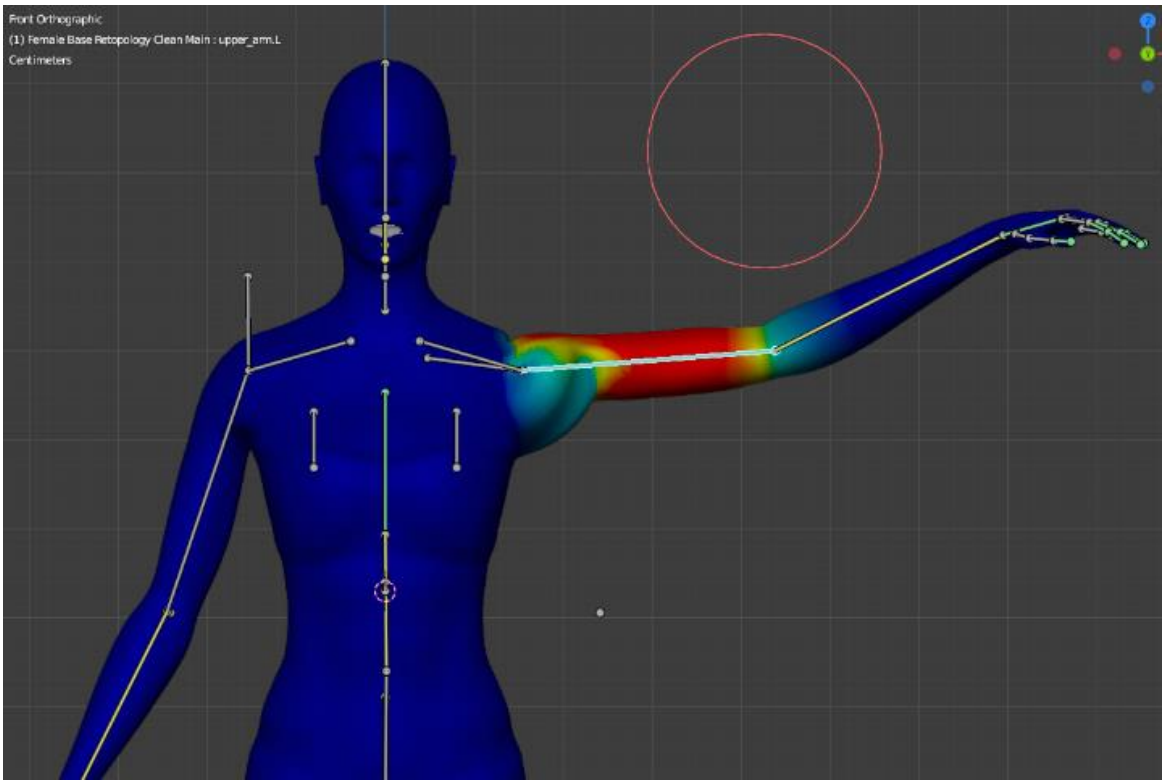
Fonte: TUTSPLUS, 2023.

2.1.2.2 Weight Paint

Após a configuração dos ossos e suas posições no personagem, inicia-se o processo de *Weight Paint*. O *Weight Paint* refere-se à atribuição de influência de cada osso aos polígonos da malha. Normalmente, a cor vermelha indica que o osso selecionado exerce influência máxima sobre esses polígonos. Este procedimento desempenha um importante papel na interação entre os ossos e a malha, permitindo um controle refinado sobre como a animação afeta diferentes partes do modelo.

Essa etapa ganha extrema importância, uma vez que um *Weight Paint* inadequado pode resultar em movimentos não naturais e deformidades prejudiciais nas malhas 3D. Um exemplo visual pode ser observado na Figura 4, onde uma má distribuição do *Weight Paint*, ao levantar o braço, ocasionando uma deformidade no ombro que destoa da naturalidade. A precisão e cuidado na aplicação do *Weight Paint* são essenciais para assegurar que a animação mantenha a fidelidade anatômica do modelo, evitando distorções indesejadas e contribuindo para a criação de movimentos fluidos e visualmente convincentes.

Figura 4 - Weight Paint do osso de um personagem



Fonte: WYMA, 2020

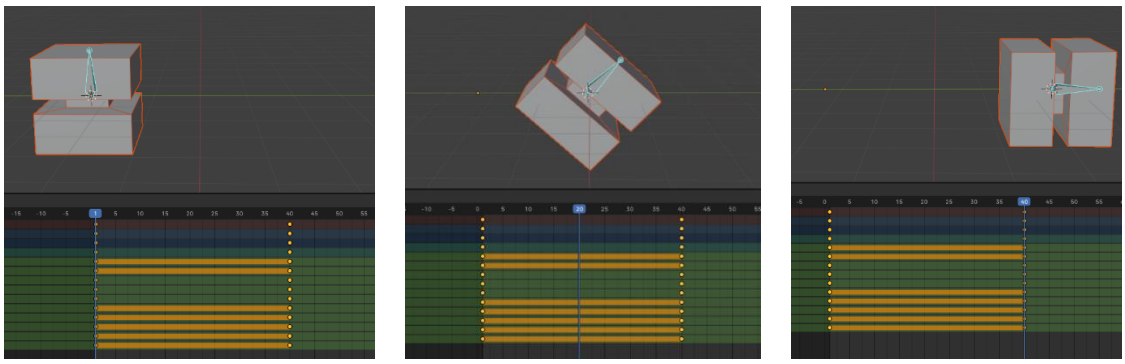
2.1.3 Animações

Após a conclusão de todos os passos, inicia-se a etapa de animação do personagem ou objeto em 3D. Nesse contexto, a animação refere-se ao processo de conferir movimento a objetos tridimensionais por meio de uma sequência de imagens ou quadros, criando a ilusão de movimento contínuo. Essa ilusão é alcançada ao modificar as propriedades e posições dos modelos 3D ao longo do tempo.

Dentro desse processo, uma "POSE" representa a configuração específica de posições, rotações e escalonamentos dos ossos de um objeto em determinado momento. Cada modificação realizada no esqueleto ao longo do tempo gera um *KeyFrame*. Durante a reprodução da animação, o *software* realiza, quando necessário, a interpolação entre o *keyframe* anterior e o próximo, criando uma pose em um *frame* intermediário. Esse processo garante a fluidez e continuidade dos movimentos, contribuindo para a criação de animações visualmente atraentes e realistas.

Ilustrativamente, na Figura 5 observa-se esse conceito em prática ordenados cronologicamente no ponto de vista da animação. Em (a) configura-se uma pose com a criação de um *keyframe* na posição 1 do tempo. Em (c) outra pose é configurada no tempo 40, também com a criação de um *keyframe*. Já em (b) evidencia a interpolação de frames intermediários dos *keyframes*, gerando poses automaticamente a partir da interpolação dos *keyframes* anteriores e posteriores. Esse processo proporciona uma transição suave e natural entre as diferentes poses ao longo da animação.

Figura 5 - Keyframes em animação



a) b) c)

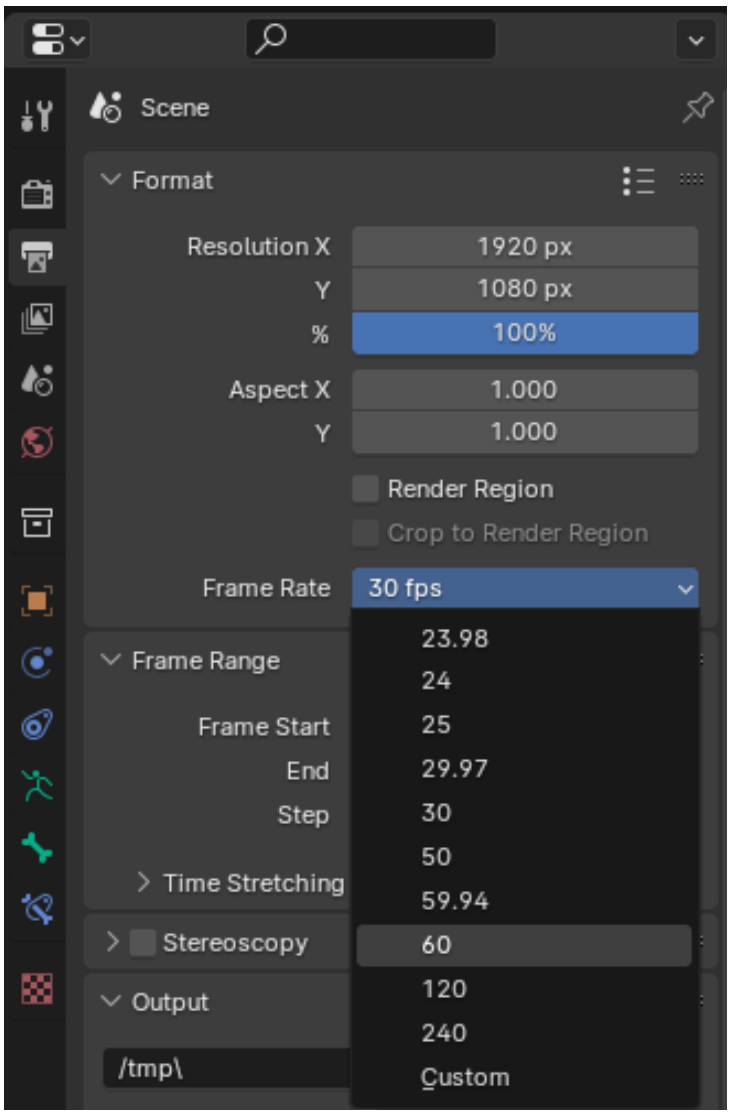
Fonte: Autoria própria

Antes de iniciar uma animação, é comum especificar a quantidade de *frames* por segundo (*Frame Rate*) a animação terá, conforme ilustrado na Figura 6. Embora uma alta taxa de *frames* por segundo proporcione uma animação com maior qualidade

visual, isso também resulta em arquivos substancialmente maiores e representa um desafio adicional para os artistas animadores, pois requer a configuração de um número significativo de *keyframes*.

Em muitos casos, animações destinadas a jogos seguem padrões específicos, sendo comum adotar taxas de 30 ou 60 *frames* por segundo. Essa padronização visa equilibrar a qualidade da animação com a eficiência em termos de armazenamento de dados e carga de trabalho para os animadores. A escolha entre essas taxas de *frames* por segundo leva em consideração a capacidade de processamento dos dispositivos, a experiência do usuário e as demandas de desempenho do jogo.

Figura 6 - Configuração de framerate no blender



Fonte: Autoria própria

A animação pode ser realizada de duas formas principais: manualmente, por meio da criação artística de *keyframes* por artistas, ou através da captura de movimento, registrando os movimentos reais de um objeto ou pessoa. A animação manual oferece controle criativo preciso, possibilita estilos únicos e detalhes personalizados, mas é demorada, cara e desafiadora em termos de consistência. Já a captura de movimento economiza tempo, proporciona realismo e é eficiente para prazos apertados, embora possa limitar a liberdade criativa, requerer infraestrutura cara e frequentemente exigir correções pós-produção significativas. A escolha entre esses métodos depende dos objetivos do projeto, limitações de tempo, orçamento e preferências estilísticas.

Para este projeto, a opção pela animação manual faz sentido, oferecendo controle criativo preciso e a capacidade de criar um estilo único. Embora possa demandar mais tempo e recursos, os benefícios estilísticos superam as desvantagens, atendendo às necessidades específicas deste projeto.

2.2 SUBSTANCE PAINTER

O Substance Painter da Allegorithmic (adquirida pela Adobe) é uma ferramenta de texturização extremamente poderosa utilizada por estúdios em todo o mundo para texturizar ativos de jogos, ambientes e personagens (KUMAR, 2020).

Uma textura, também conhecida como *bitmap* ou mapa de textura, é um arquivo de imagem bidimensional que pode ser aplicado à superfície de qualquer modelo 3D para adicionar cor, textura ou outros detalhes de superfície (JANTUNEN, 2017).

Seria possível modelar todas as formas e detalhes de modelos 3D manualmente em um programa de modelagem 3D, mas geralmente essa não é a maneira mais eficaz para o uso pretendido e demandaria um tempo considerável para criá-los (JANTUNEN, 2017).

O Substance Painter tem sua abordagem baseada em camadas e isso permite uma flexibilidade notável, possibilitando a aplicação e ajuste de diversas texturas de forma não destrutiva. Essa característica é particularmente valiosa para artistas, pois podem experimentar e modificar as texturas com facilidade, mantendo um fluxo de trabalho intuitivo.

Ao implementar texturas, ocorre uma significativa economia de recursos. Em vez de armazenar informações detalhadas diretamente nos modelos 3D, as texturas possibilitam representar visualmente superfícies complexas sem a necessidade de aumentar drasticamente o consumo de recursos computacionais. Isso é especialmente crucial em ambientes de jogos, onde a eficiência do processamento é vital para garantir uma experiência fluida e responsiva.

Portanto, a utilização estratégica de texturas não apenas permite alcançar resultados visuais impressionantes, como mostrado na Figura 7, mas também é essencial para garantir que os jogos sejam visualmente envolventes e operem de maneira eficiente em uma variedade de dispositivos, atendendo às demandas de desempenho dos jogadores.

Figura 7 - Antes e depois de texturização

Fonte: FOWLER, 2020.

2.2.1 Physically based rendering

Renderização é o processo de produzir uma imagem a partir da descrição de uma cena 3D. Obviamente, esta é uma tarefa ampla, e há muitas maneiras de abordá-la. Técnicas baseadas em física tentam simular a realidade; ou seja, elas utilizam princípios da física para modelar a interação da luz com a matéria. Embora uma abordagem baseada em física possa parecer a maneira mais óbvia de abordar a

renderização, ela só foi amplamente adotada na prática nos últimos 15 anos aproximadamente (PHARR, JAKOB, HUMPHREYS, 2023).

No contexto de texturas, o Physically based rendering (PBR) implica a criação de mapas que representam informações físicas específicas, como mapas de albedo, rugosidade, metálico, normais e outros. Esses mapas são cruciais para alcançar resultados visuais de alta qualidade, garantindo que os materiais se comportem de maneira convincente sob diferentes fontes de luz.

A *Unreal Engine* é conhecida por adotar princípios PBR em seu sistema de renderização. O Substance Painter, portanto, torna-se uma escolha lógica para a criação de texturas, pois é projetado para suportar e otimizar o fluxo de trabalho PBR. Ao utilizar o Substance Painter, o projeto estará alinhado com as práticas modernas de desenvolvimento de jogos, facilitando a criação de texturas.

2.2.2 Textures

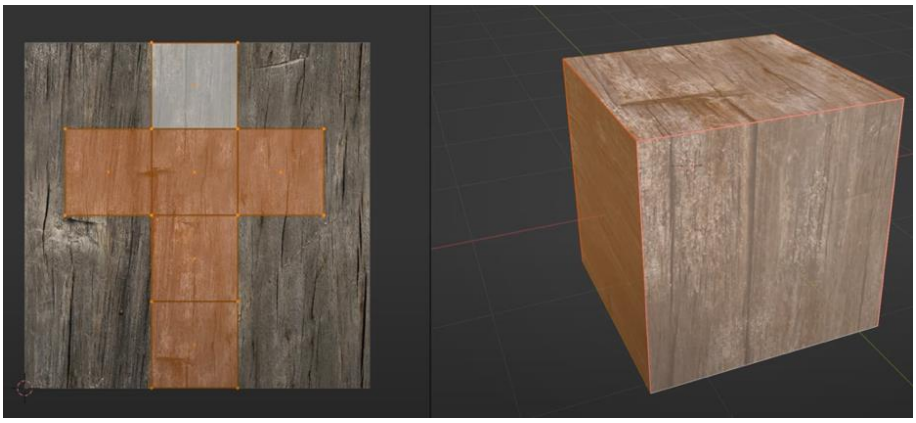
Nos capítulos subsequentes serão apresentadas e explicadas as finalidades de algumas texturas consideradas importantes pelo autor. Todas as texturas apresentadas serão criadas no substance painter e utilizadas no projeto do jogo.

2.2.2.1 Textura de Cor (Albedo Texture):

O albedo desempenha um papel crucial na determinação da cor da sua textura, sendo a base em torno da qual todo o material é construído. Seu padrão pode ser uma cor única ou uma imagem representativa. Por exemplo, na Figura 8, podemos observar como o albedo influencia diretamente a cor predominante da textura.

Ao escolher uma imagem como padrão de albedo, é importante garantir que a iluminação na fotografia seja uniforme, sem sombras pronunciadas. Isso se deve ao fato de que as sombras presentes na imagem de origem podem resultar em irregularidades visuais nas texturas, comprometendo a naturalidade e autenticidade da representação no ambiente do jogo.

Figura 8 - Textura na UV de um cubo



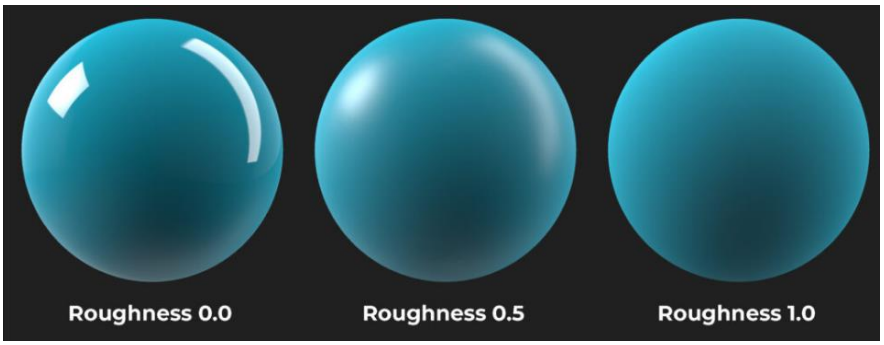
Fonte: Vídeo de RyanKingArt no youtube

2.2.2.2 Textura de Rugosidade (Roughness Texture):

A rugosidade desempenha um papel fundamental na aparência visual de um material, influenciando o quão áspera ou suave uma superfície parece. A faixa de valores de rugosidade varia de 0.0 a 1.0. Na Figura 9, podemos observar claramente essa relação. Na extremidade esquerda da Figura, onde a rugosidade é definida como 0, o material é totalmente reflexivo, resultando em reflexões nítidas e iluminação brilhante. No centro, com uma rugosidade de 0.5, a reflexão torna-se fosca, dispersando a luz de maneira mais difusa. À direita, com rugosidade igual a 1, o material perde completamente seus reflexos, apresentando uma superfície sem brilho.

Para ilustrar, considere a borracha, que tipicamente possui uma rugosidade próxima a 1.0, enquanto o plástico brilhante tende a ter uma rugosidade próxima a zero. Esses valores são representados em escala de cinza nos mapas, onde o branco indica uma superfície mais áspera e o preto indica uma superfície mais brilhante e suave. Essa compreensão detalhada da rugosidade é crucial ao criar texturas no Substance Painter para garantir a fidelidade visual e o realismo desejado nos materiais.

Figura 9 - Mapa roughness com diferentes intensidades



Fonte: a23d, 2023.

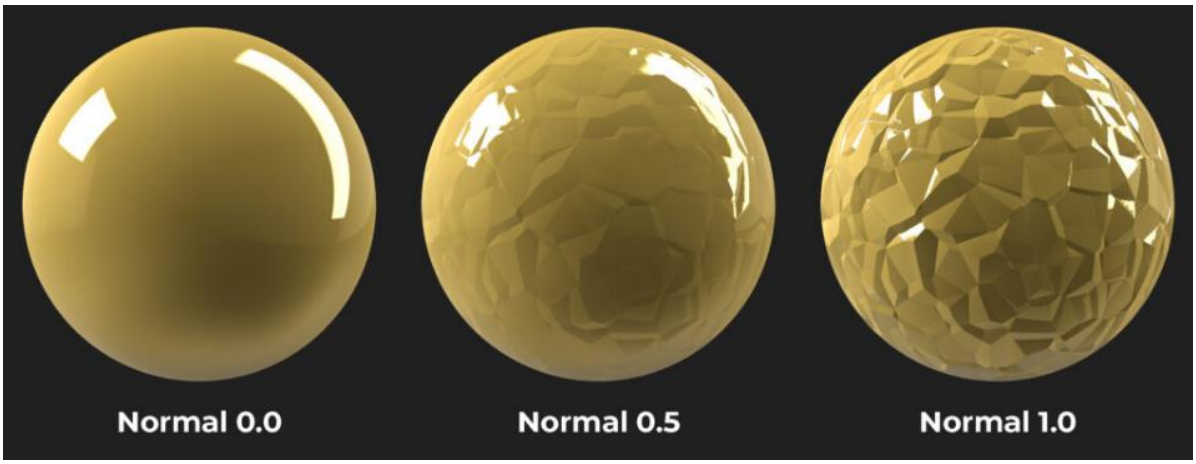
2.2.2.3 Mapa Normal (Normal Map):

Os mapas normais desempenham um papel crucial ao adicionar profundidade e detalhes às texturas. Na Figura 10, podemos observar essa influência: à esquerda, com a força do *normal map* em 0, a superfície aparece plana; no meio, com força em 0.5, pequenos relevos começam a criar sombras e reflexões mais nítidas; à direita, com força em 1, as depressões são mais pronunciadas, proporcionando uma sensação tridimensional mais convincente, destacando não apenas as sombras, mas também reflexões mais aguçadas.

Esses mapas simulam como a luz interage com a superfície de uma substância, utilizando cálculos complexos para criar pequenos relevos e depressões. Embora a geometria básica do modelo não seja alterada, o mapa normal produz o efeito visual de sombras e reflexões mais nítidas, proporcionando uma representação mais rica e realista em ambientes virtuais.

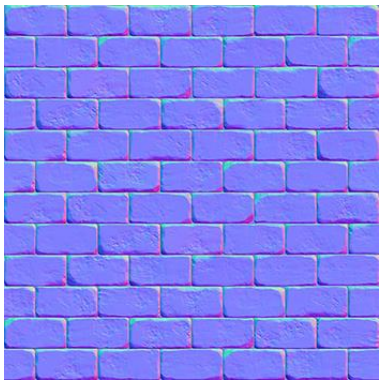
Em contextos de jogos, os mapas normais são de extrema importância, pois oferecem uma maneira eficiente de adicionar detalhes aos modelos sem impactar significativamente no desempenho da renderização. Um exemplo da textura de mapa normal pode ser visto na Figura 11.

Figura 10 - Mapa normal com diferentes intensidades



Fonte: a23d, 2023.

Figura 11 - Mapa normal de parede de tijolos



Fonte: MCREYNOLDS, 2005.

2.2.2.4 Textura Especular (Specular Texture):

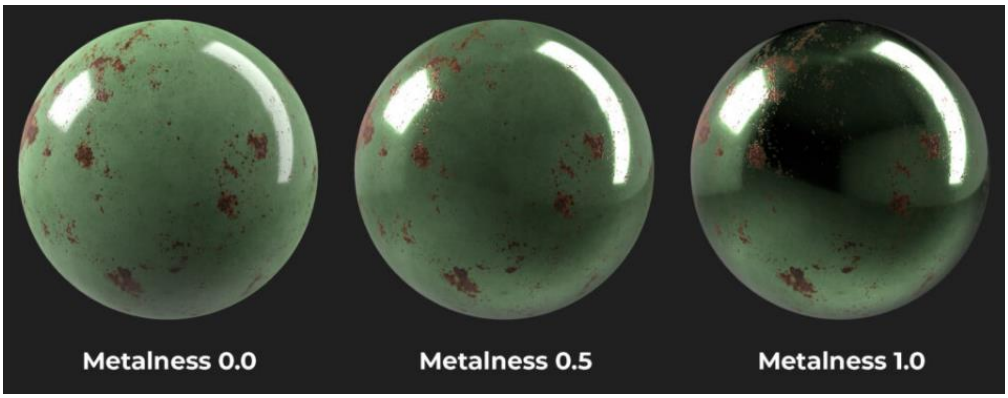
Essa textura tem a função de controlar as áreas que refletem mais luz, destacando regiões suaves e brilhantes em um objeto tridimensional. Ao ajustar a textura especular, é possível criar efeitos visuais que contribuem para a aparência de materiais mais complexos e realistas.

Ao modificar a textura especular, os desenvolvedores e artistas têm o controle preciso sobre como a luz é refletida em diferentes partes do modelo, resultando em destaques mais intensos em áreas desejadas. Essa capacidade de manipular a especularidade contribui significativamente para a representação fiel de materiais, proporcionando um aspecto mais autêntico e visualmente atraente aos objetos em ambientes 3D.

2.2.2.5 Textura de metalicidade (Metalness Texture):

O mapa de metalicidade indica as áreas metálicas e não metálicas em um objeto, permitindo a simulação realista de reflexões e comportamento da luz com base nas propriedades metálicas do material. Na Figura 12, à esquerda com a metalicidade em 0, a cor do albedo é completamente visível, resultando em uma aparência similar a plástico ou cerâmica. No meio, com metalicidade em 0.5, o material sugere ser metal pintado. À direita, com metalicidade em 1.0, a superfície perde praticamente toda a sua cor de albedo, refletindo apenas o ambiente ao redor. Quando a metalicidade é combinada com uma rugosidade de 0.0, por exemplo, a superfície se assemelha a um espelho do mundo real.

Figura 12 - Mapa metalness com diferentes intensidades

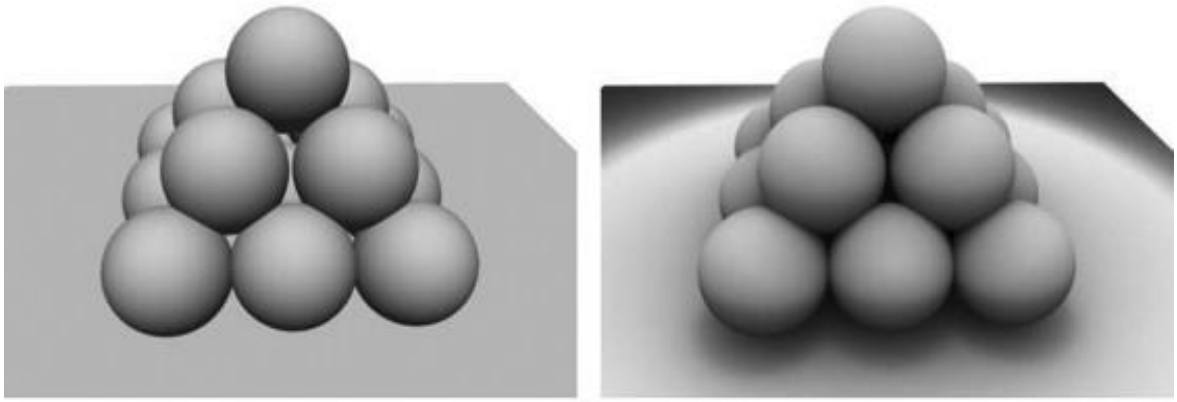


Fonte: a23d, 2023.

2.2.2.6 Textura de Oclusão Ambiental (Ambient Occlusion Texture):

O mapa de oclusão ambiental é um componente fundamental no processo de texturização, representando áreas sombreadas ou menos iluminadas em uma cena. Sua função é aprimorar a sensação de profundidade e realismo ao criar sombras suaves e áreas menos iluminadas nas regiões relevantes do objeto. Na prática, o mapa de oclusão ambiental contribui para definir a interação entre a luz e as superfícies, adicionando nuances visuais que enriquecem a representação tridimensional.

O mapa de oclusão ambiental geralmente criado automaticamente por *softwares* de modelagem ou específicos de texturização. Esses mapas são valiosos para indicar não apenas as porções côncavas de um objeto, mas também superfícies próximas a outros objetos.



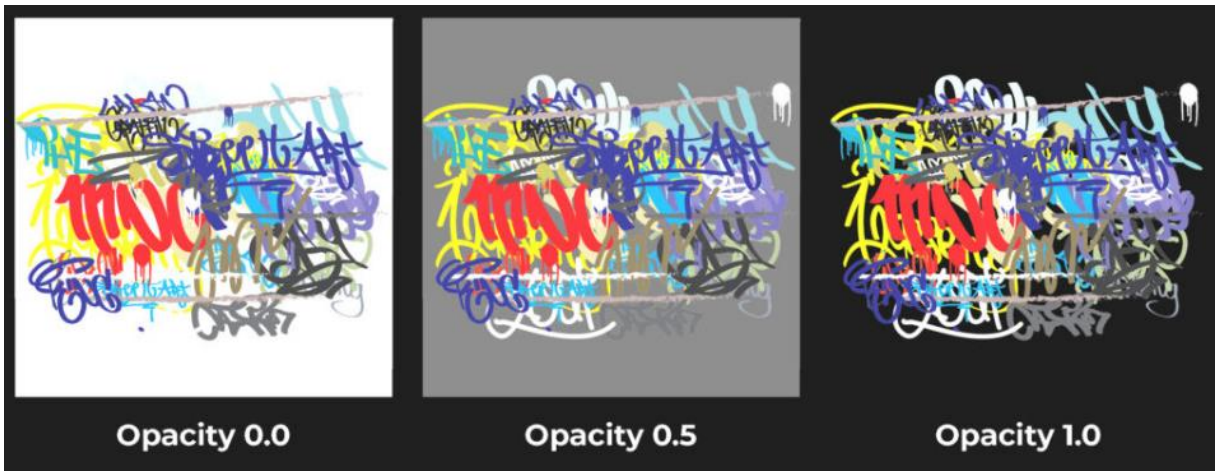
Fonte: MCREYNOLDS, 2005

2.2.2.7 Textura de Opacidade (Opacity Texture):

A textura de opacidade desempenha um papel crucial na definição da transparência ou opacidade de uma superfície, permitindo a criação de objetos transparentes ou semi-transparentes. Na Figura 13, podemos observar a influência dessa textura: à esquerda, com a opacidade em 0, a superfície é totalmente opaca; no meio, com opacidade em 0.5, a transparência é parcial, e à direita, com opacidade em 1, a superfície é completamente transparente.

Essa textura é essencial para incorporar detalhes visuais, como vidro, fumaça ou outros materiais translúcidos, enriquecendo a representação visual de cenas 3D. Um exemplo prático é na texturização da copa de uma árvore, onde é possível construir um conjunto completo de folhas em um único plano poligonal. Utilizando um mapa de opacidade, a visibilidade dessas folhas é controlada. Ao aplicar a textura de opacidade, o excesso do polígono desaparece, proporcionando uma representação realista da folhagem. Esses polígonos podem então ser sobrepostos para criar árvores realistas com baixa demanda de capacidade de processamento, tornando esse método eficiente para a renderização de cenários detalhados.

Figura 13 - Mapa opacity com diferentes intensidades



Fonte: a23d, 2023.

Após passar pelas etapas de modelagem, animação e texturização, o modelo atinge seu estágio final ao ser importado para a *Unreal Engine*. Nesse ambiente, as animações previamente criadas serão executadas, e as cenas renderizadas de acordo com as configurações definidas pelo programador.

2.2.3 Smart Materials

Os *smart materials* no Substance Painter são conjuntos de camadas e configurações predefinidos que simplificam o processo de aplicação de texturas complexas em modelos 3D. Esses materiais inteligentes incorporam uma variedade de texturas, ajustes e efeitos em uma única camada, oferecendo aos artistas uma solução pronta para uso e acelerando o processo de texturização. Os *smart materials* são projetados para serem eficientes, proporcionando resultados visuais complexos com facilidade, enquanto ainda mantêm a flexibilidade para personalização conforme necessário.

É importante destacar que, antes de aplicar *smart materials*, é necessário realizar um processo chamado "bake" de texturas. O bake envolve a geração de mapas de textura adicionais, como mapas de curvatura, cavidade, normal, entre outros. Esses mapas capturam informações essenciais sobre a geometria do modelo, como curvas e cavidades. Os *smart materials* utilizam essas informações geradas durante o *bake* para se adaptarem de maneira mais precisa à superfície do objeto. Por exemplo, ao aplicar um *smart material* que simula ferrugem, as informações de

cavidade provenientes do *bake* ajudarão a determinar onde a ferrugem deve se acumular, criando uma representação mais realista em áreas menos expostas à luz.

2.3 UNREAL ENGINE

A *Unreal Engine* foi criada em 1998 com o advento do jogo de tiro em primeira pessoa Unreal. Esta versão da *Unreal Engine* combinou diversos sistemas, incluindo renderização, detecção de colisões, inteligência artificial, visibilidade, rede, script e gerenciamento de arquivos. A API Glide, no cerne da *engine*, foi desenvolvida especificamente para GPUs 3dfx. Unreal Tournament sucedeu o Unreal e fez grandes avanços para melhorar a renderização e o desempenho em rede. A *Unreal Engine* tornou-se amplamente popular devido ao design modular da arquitetura da *engine* e à inclusão de uma linguagem de script chamada UnrealScript. Baseado principalmente em C++, o UnrealScript permitiu aos usuários criar modificações facilmente (Sanders, 2016).

A *Unreal Engine* está sendo desenvolvida pela empresa Epic Games. Atualmente, é a principal *engine* em visualização realista (Šmíd, 2017). A *Unreal Engine* oferece soluções para o desenvolvimento de ambientes em larga escala e jogos massivos multiplayer. Eles esperam que os desenvolvedores visem o hardware mais recente. Portanto, os requisitos do sistema são elevados tanto para o desenvolvimento quanto para os jogos executáveis finais. A Unreal é considerada a principal plataforma em renderização fotorrealista e visualização arquitetônica (Šmíd, 2017).

Recentemente, a *Unreal Engine* aprimorou ainda mais sua capacidade de atingir níveis de realismo notáveis com a introdução das tecnologias Lumen e Nanite. Lumen é um sistema global de iluminação totalmente dinâmico que permite a criação de ambientes mais realistas, reagindo de maneira instantânea às mudanças de iluminação.

Lumen destaca-se como o sistema dinâmico abrangente de iluminação global e reflexos na Unreal Engine 5, especialmente desenvolvido para as plataformas de próxima geração. Designado como o padrão para iluminação global e reflexos, o Lumen proporciona renderização de inter-reflexões difusas com rebatimentos infinitos e reflexos especulares indiretos em ambientes expansivos e minuciosos. Sua

capacidade abrange escalas que variam de milímetros a quilômetros, oferecendo uma experiência visual imersiva e detalhada. (UNREAL ENGINE, 2022).

Por outro lado, a tecnologia Nanite revolucionou a maneira como a geometria é tratada, permitindo a renderização de detalhes incríveis em modelos 3D sem a necessidade de otimizações manuais. Isso significa que objetos podem ter uma quantidade surpreendente de detalhes, independentemente da distância do observador, proporcionando uma fidelidade visual sem precedentes.

Nanite representa o sistema de geometria virtualizada incorporado à Unreal Engine 5, empregando um novo formato interno de malha e tecnologia de renderização. Esse sistema é projetado para apresentar detalhes em uma escala de pixel elevada e lidar eficientemente com uma grande quantidade de objetos. Sua abordagem é inteligente, concentrando-se apenas nos detalhes perceptíveis, otimizando assim o desempenho. Além disso, o formato de dados do Nanite é altamente comprimido e permite streaming de alta granularidade, ajustando automaticamente os níveis de detalhamento (UNREAL ENGINE, 2022).

Para ilustrar essa capacidade de realismo, considere a Figura 14, que exibe uma floresta com luz e folhagem extraordinariamente realistas.

Figura 14 - Floresta na Unreal Engine

Fonte: Criado por Michal Chojnowski

2.3.1 Programação

Nos próximos subcapítulos serão apresentadas os dois tipos de programação presentes na Unreal Engine. O primeiro é a programação visual originada da Unreal Script e a seguinte será mostrada como é feita a programação em linguagem C++.

2.3.1.1 Programação Visual

A Unreal possui uma poderosa ferramenta de programação visual denominada *Blueprints*. Na *Unreal Engine* são uma ferramenta inovadora e poderosa para desenvolvedores, designers e artistas, proporcionando uma abordagem visual e intuitiva na criação de lógica de jogo sem a necessidade de programação convencional. Esses diagramas visuais, compostos por nodos interconectados, representam diferentes ações, eventos e lógica de programação, permitindo a construção de sequências complexas de maneira eficiente e acessível.

O poder das *Blueprints* vai além da simplicidade, abrangendo uma variedade de elementos, desde o movimento de personagens até a implementação de sistemas de jogo completos. Essa versatilidade democratiza o processo de desenvolvimento, permitindo que membros da equipe, mesmo sem experiência em programação, contribuam ativamente para a criação e prototipagem de funcionalidades (SEWELL, 2015).

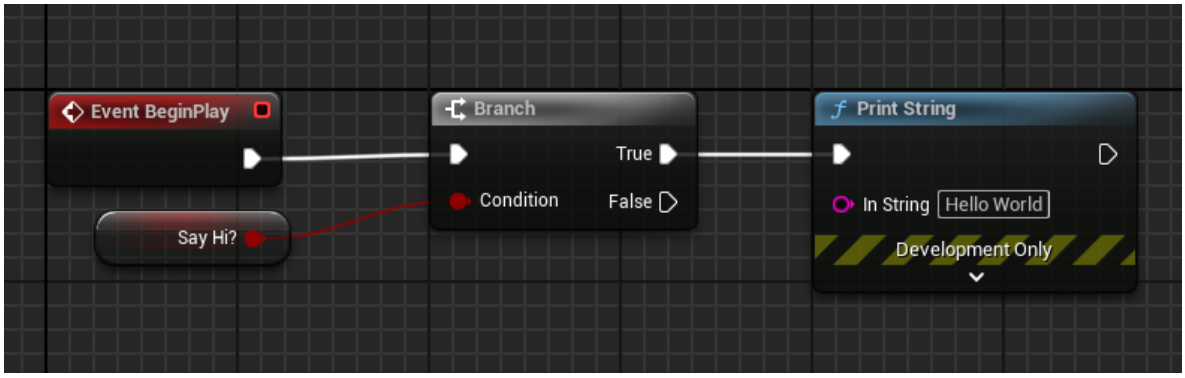
Como exemplo prático, na Figura 15, temos uma ilustração de como *Blueprints* podem ser utilizadas para criar um simples *Hello World*. O evento *Begin Play* dispara um acionador para um nó do tipo *Branch* (homólogo à um operador condicional), que verifica se a variável *Say Hi* é verdadeira antes de imprimir a mensagem.

A motivação por trás da adoção massiva de *Blueprints* na *Unreal Engine* reside na busca por eficiência e acessibilidade no desenvolvimento de jogos. Essa abordagem visual intuitiva acelera o processo de design, simplificando a implementação de ideias e iteração durante a fase de desenvolvimento. Além disso, a visualização direta da lógica do jogo e a facilidade de manutenção são características essenciais, facilitando ajustes rápidos nas funcionalidades conforme necessário.

A flexibilidade e o poder oferecidos pelas *Blueprints* também são notáveis, garantindo que, apesar de sua acessibilidade para não programadores, elas possam

oferecer funcionalidades avançadas para programadores experientes. Esse equilíbrio entre simplicidade e capacidade avançada torna as *Blueprints* uma peça fundamental e eficaz no toolkit de desenvolvimento de jogos na *Unreal Engine*.

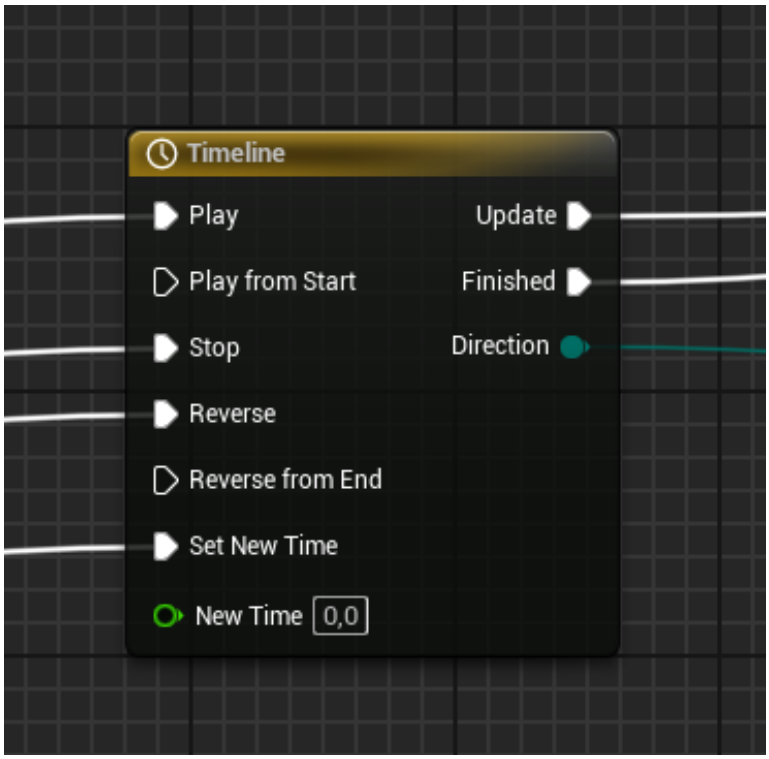
Figura 15 - Unreal Blueprint



Fonte: Autoria própria

É bastante comum que blocos tenham uma única entrada e uma saída de fluxo; no entanto, há blocos mais complexos que apresentam diversas funcionalidades e fluxos abrangentes de ativação e saída. Um exemplo disso pode ser observado na Figura 16, onde o bloco possui várias entradas e duas saídas, demonstrando a diversidade de configurações possíveis.

Figura 16 - Timeline node



Fonte: Autoria própria

2.3.1.2 Programação C++

Embora a *Unreal Engine* seja conhecida pelo seu poderoso sistema de *Blueprints*, a programação em C++ desempenha um papel crucial no desenvolvimento de jogos, proporcionando uma flexibilidade e controle mais profundos sobre a lógica do jogo. Na verdade, o sistema de *Blueprints* é uma camada visual que simplifica a interação com classes e funções escritas em C++.

A utilização de macros em C++, como *ufunction, uproperty* e *generated_body* é uma prática comum na *Unreal Engine*. Essas macros fornecem metadados essenciais para o sistema, permitindo que as funções e propriedades escritas em C++ sejam acessadas e manipuladas visualmente nas *Blueprints*. Por exemplo, *ufunction* expõe funções para as *Blueprints*, *uproperty* gerencia propriedades e *generated_body* é responsável por gerar o código associado.

Essas macros desempenham um papel vital na comunicação entre C++ e *Blueprints*, possibilitando que qualquer programador faça modificações na interface das *Blueprints* sem a necessidade de acessar diretamente o código-fonte em C++. Isso promove a colaboração eficiente entre programadores e designers, permitindo ajustes rápidos e prototipagem visual.

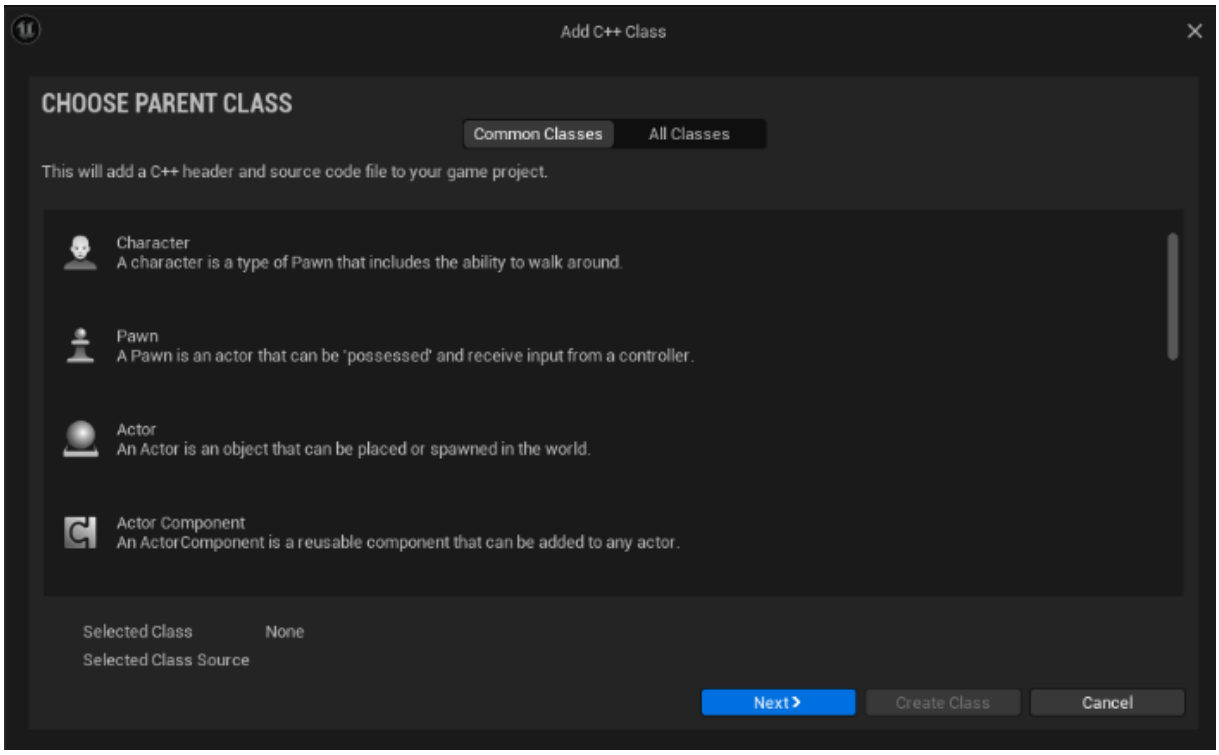
É importante destacar que, apesar da aparência visual das *Blueprints*, cada *Blueprint* é, na verdade, uma instância de uma classe em C++. Isso significa que, por trás da interface visual, a lógica do jogo é suportada por robustas implementações em C++, proporcionando o melhor dos dois mundos: a facilidade de uso do sistema de *Blueprints* e o poder e controle da programação em C++.

Para utilizar o C++ na *Unreal Engine*, o primeiro passo é criar uma classe. A Unreal facilita esse processo oferecendo opções para criar classes filhas de classes já existentes. Isso é evidenciado na Figura 17 (a), que apresenta a interface de criação de classe com as opções mais comuns, e na Figura 17 (b), que lista todas as classes disponíveis.

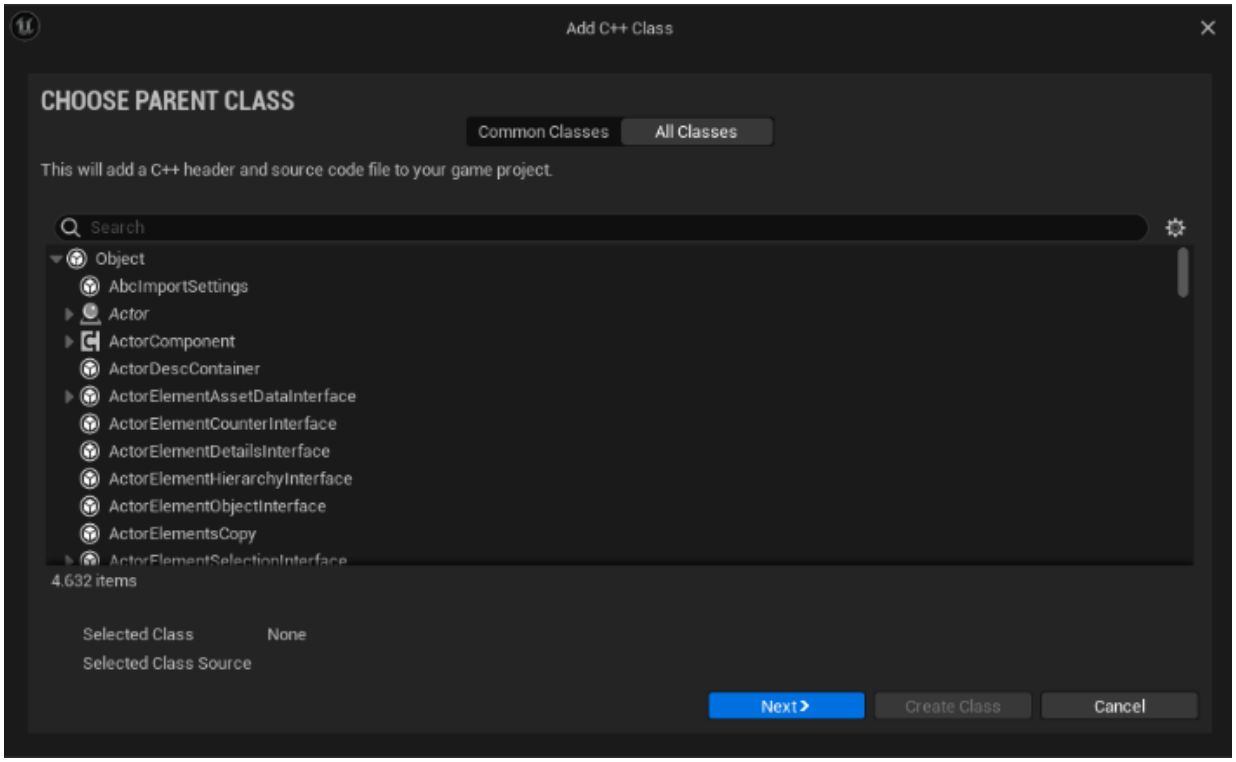
A oportunidade de criar classes baseadas em outras já existentes na própria *Engine* é providencial para promover a reutilização de código e seguir práticas de desenvolvimento eficientes. Ao criar uma classe derivada de uma classe existente, os desenvolvedores herdam as funcionalidades e características já implementadas, economizando tempo e esforço. Além disso, essa abordagem mantém a consistência e padronização dentro da *Engine*, algo muito importante em vista da grande complexidade do software.

Após a criação de classes na *Unreal Engine*, algumas delas já vêm com funções escritas, como a *BeginPlay* e *Tick*. A função *BeginPlay* é chamada no início do jogo ou quando um objeto é instanciado, sendo frequentemente utilizada para inicializar variáveis e configurar o estado inicial do objeto. Por sua vez, a função *Tick* é executada a cada quadro do jogo, sendo crucial para lógicas que precisam ser atualizadas continuamente, como movimento e interações. Essas funções são apenas 2 entre diversas outras que são implementadas em classes superiores e que podem ser sobrecarregadas, assim destacando sempre a importância de chamar a função na classe superior.

Figura 17 – Criação de classes na Unreal



a)



b)

Fonte: Autoria própria

Um exemplo prático pode ser visualizado na Figura 18, que apresenta o arquivo *header* recém-criado de uma classe derivada de *ACharacter*. Nesse exemplo, as funções *Tick, BeginPlay* e *SetupPlayerInputComponent* já estão declaradas no código. A função *SetupPlayerInputComponent* é frequentemente usada para configurar a entrada do jogador.

Figura 18 - Arquivo header de classe filha de Character

```
1   // Fill out your copyright notice in the Description page of Project Settings.
2
3   #pragma once
4
5   #include "CoreMinimal.h"
6   #include "GameFramework/Character.h"
7   #include "MyCharacter.generated.h"
8
9   UCLASS()
10  class MYPROJECT2_API AMyCharacter : public ACharacter
11  {
12      GENERATED_BODY()
13
14  public:
15      // Sets default values for this character's properties
16      AMyCharacter();
17
18  protected:
19      // Called when the game starts or when spawned
20      virtual void BeginPlay() override;
21
22  public:
23      // Called every frame
24      virtual void Tick(float DeltaTime) override;
25
26      // Called to bind functionality to input
27      virtual void SetupPlayerInputComponent(class UInputComponent* PlayerInputComponent) override;
28
29  };
30
```

Fonte: Autoria própria

2.3.2 Classes

Aqui estão apresentadas algumas classes consideradas importantes ou amplamente utilizadas pelo autor. Muitas dessas classes possuem uma extensa variedade de funções já implementadas, e algumas apresentam interfaces significativamente diferentes das usualmente encontradas na Unreal. Por exemplo, uma *Blueprint* do tipo Animation *Blueprint* possui uma aparência visual totalmente distinta de uma *Blueprint* comum, uma vez que é projetada para o controle específico de animações de personagens. Essa diversidade de classes oferece aos desenvolvedores uma gama abrangente de ferramentas e funcionalidades para a criação e aprimoramento de seus projetos na *Unreal Engine*.

2.3.2.1 World

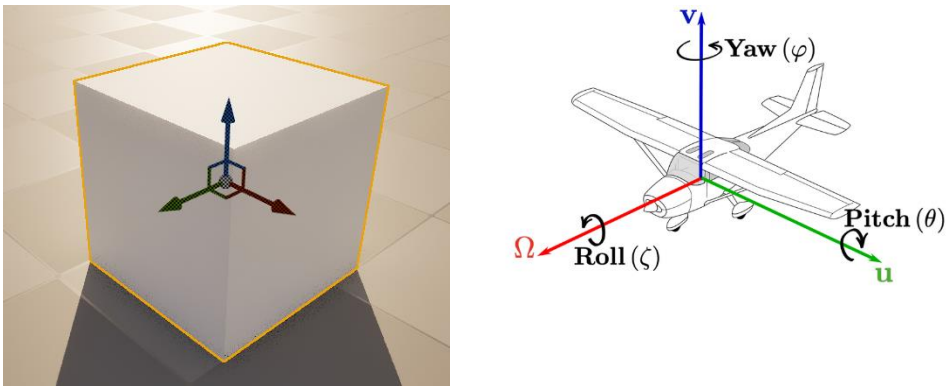
Na arquitetura da *Unreal Engine*, assim como em muitas outras *engine*s de jogos, a organização dos elementos dentro de uma cena ou jogo ocorre dentro de um contexto específico. Enquanto algumas *engine*s usam o termo *Scene* para descrever esse contexto, como é o caso da Unity e da Godot, na Unreal, essa entidade é denominada *World*, sendo uma classe derivada de *UWorld*.

O objeto *World* desempenha uma função crucial na gestão de todos os elementos presentes no contexto atual do jogo na *Unreal Engine*. Além disso, sua importância se destaca na determinação das coordenadas dos objetos no mundo virtual, já que a posição de cada elemento é calculada em relação à posição do *World*.

No contexto da Unreal, para posicionar objetos e personagens, faz-se uso de uma estrutura chamada *Transform*. Essa estrutura, ou *struct*, contém informações fundamentais sobre a posição, rotação e escala de um objeto físico. As posições são determinadas pelos eixos X, Y e Z, enquanto as rotações são definidas pelos ângulos de *Pitch, Yaw* e *Roll*. Quanto à escala, esta é representada pela razão de tamanho nos eixos X, Y e Z. Uma escala de (1, 1, 1) indica um objeto com tamanho normal em relação ao modelo importado.

Para ilustrar a aplicação prática desses conceitos, considere a Figura 19 (a), onde um cubo é representado no ambiente Unreal, acompanhado pelo *gizmo* de coordenadas (X, Y, Z). Um *gizmo*, nesse contexto, é uma ferramenta visual que representa os eixos tridimensionais e facilita a compreensão da orientação e posicionamento do objeto no espaço. Na Figura 19 (b), um avião é exibido, juntamente com um *gizmo* que indica os eixos relacionados a *Roll, Pitch* e *Yaw.* Já a Figura 20 apresenta a estrutura de um *Transform* dentro da *Unreal Engine*.

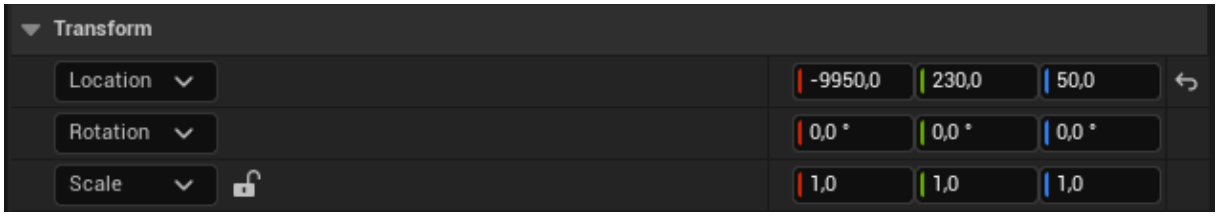
Figura 19 - Unreal Gizmos



a) b)

Fonte: DEGOND ET AL, 2021.

Figura 20 - Transform na Unreal



Fonte: Autoria própria

2.3.2.2 Actor

A classe *Actor* na *Unreal Engine* desempenha um papel fundamental e versátil no desenvolvimento de jogos. Ela serve como a base para todos os elementos interativos no mundo do jogo, representando desde personagens e inimigos até objetos estáticos e pontos de interação.

O *Actor* na *Unreal Engine* destaca-se pela sua característica de ter uma presença física no mundo do jogo, proporcionando informações sobre posição, rotação e escala.

Além disso, a classe *Actor* é a base para muitas outras classes importantes na *Unreal Engine*, como *ACharacter* para personagens jogáveis, *AProjectile* para projéteis e *APawn* para objetos controláveis.

A classe *Actor* é tão central dentro da *Unreal Engine* que, por convenção, qualquer classe filha de um *Actor* recebe o prefixo "A". Essa prática reforça a importância da hierarquia de classes baseada em *Actor* na *Engine*, proporcionando uma padronização que contribui para a clareza e a compreensão do código.

2.3.2.3 Controller

Na *Unreal Engine*, a classe *Controller* possibilita que atores recebam inputs no contexto do jogo, tanto provenientes de Inteligências Artificiais quanto de Usuários. O *Controller* assume a responsabilidade de capturar inputs do usuário ou de IA e transmitir comandos ao ator que está sob sua posse. Dentre as subclasses relevantes de *Controllers*, destacam-se a *AIControlle*r, designada para receber inputs de Inteligências Artificiais, e o *PlayerController*, responsável por receber e encaminhar

inputs do usuário. Uma analogia pertinente seria comparar o *Controller* à função de manipulação de uma marionete, sendo o *Actor* a entidade controlada, ilustrando assim a interação vital entre essas entidades no ambiente de jogo.

2.3.2.4 Componente

Na *Unreal Engine*, um Componente é uma unidade modular e reutilizável que pode ser anexada a um ator para conferir funcionalidades específicas. Esses componentes são blocos de construção importantes que contribuem para a construção e extensão das características de um ator no ambiente de desenvolvimento de jogos. Cada componente tem sua própria lógica e propriedades, permitindo a composição flexível e eficiente de elementos no design do jogo.

2.3.2.5 Character

A classe Character é uma extensão da classe Actor na *Unreal Engine*, projetada para representar personagens jogáveis. Essa classe permite que o personagem seja controlado por um *Controller* e receba *inputs* de movimento para se locomover no ambiente do jogo. Além disso, a *Unreal Engine* proporciona recursos integrados que facilitam a replicação do movimento em ambientes de rede, corrigindo perdas e desconexões. Em resumo, a classe *Character* é comumente empregada para modelar e controlar personagens em jogos, oferecendo uma estrutura robusta e eficiente para essas entidades.

2.3.2.6 Animation

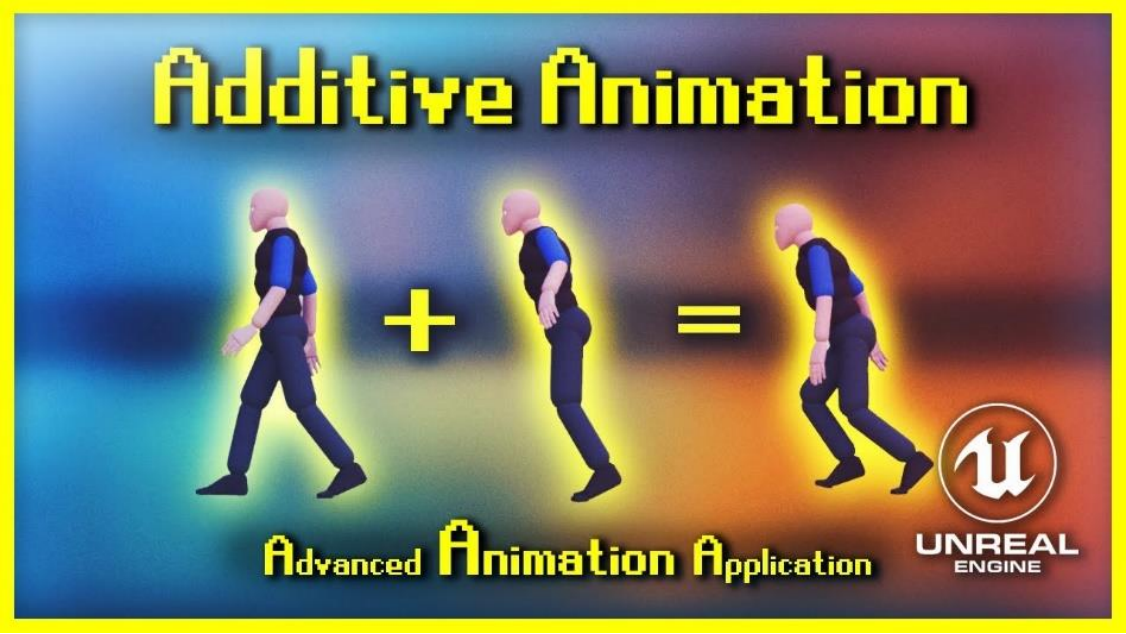
Uma classe do tipo Animation na *Unreal Engine* refere-se a um recurso utilizado para representar e controlar animações dentro do jogo. Esse tipo de *asset* armazena dados e informações necessárias para dar vida aos personagens, objetos ou elementos interativos no ambiente virtual. *Animations assets* contêm dados como quadros-chave (*keyframes*), transições entre poses, informações sobre movimentos e outras propriedades específicas da animação.

2.3.2.6.1 Additive Animation

A Additive Animation, ou animação aditiva, é um conceito na animação 3D onde uma pose adicional é sobreposta a uma pose base, resultando em uma combinação final. Na Figura 21, à esquerda, temos uma pose base, representando o estado inicial ou padrão de um personagem ou objeto. No meio, encontra-se uma pose aditiva, que descreve as mudanças específicas desejadas em relação à pose base. Essas mudanças podem incluir movimentos adicionais, variações de expressão ou ajustes específicos.

Ao aplicar a animação aditiva, a pose aditiva é somada à pose base, resultando na pose final à direita. Essa soma cria uma representação composta que incorpora tanto os elementos da pose base quanto os da pose aditiva. Esse conceito é frequentemente utilizado para realizar ajustes refinados ou sobreposições em animações existentes, permitindo maior flexibilidade e precisão no controle das expressões e movimentos dos personagens ou objetos durante a animação 3D.

Figura 21 - Additive Animation



Fonte: PRISMATICADEV, 2021

2.3.2.6.2 Physical Animation

A Physical Animation na *Unreal Engine* refere-se a uma técnica que incorpora física aos elementos de animação, proporcionando movimentos mais realistas e dinâmicos para personagens ou objetos no jogo. Em vez de depender exclusivamente

de animações predefinidas, a *Physical Animation* permite que as forças físicas influenciem as animações em tempo real.

Essa técnica é frequentemente utilizada para simular movimentos naturais, como a interação de um personagem com o ambiente, colisões com objetos ou a resposta a forças externas. Um exemplo comum é a simulação realista do movimento de cabelos, roupas ou membros do personagem com base em forças físicas como vento, gravidade ou interações com outros objetos.

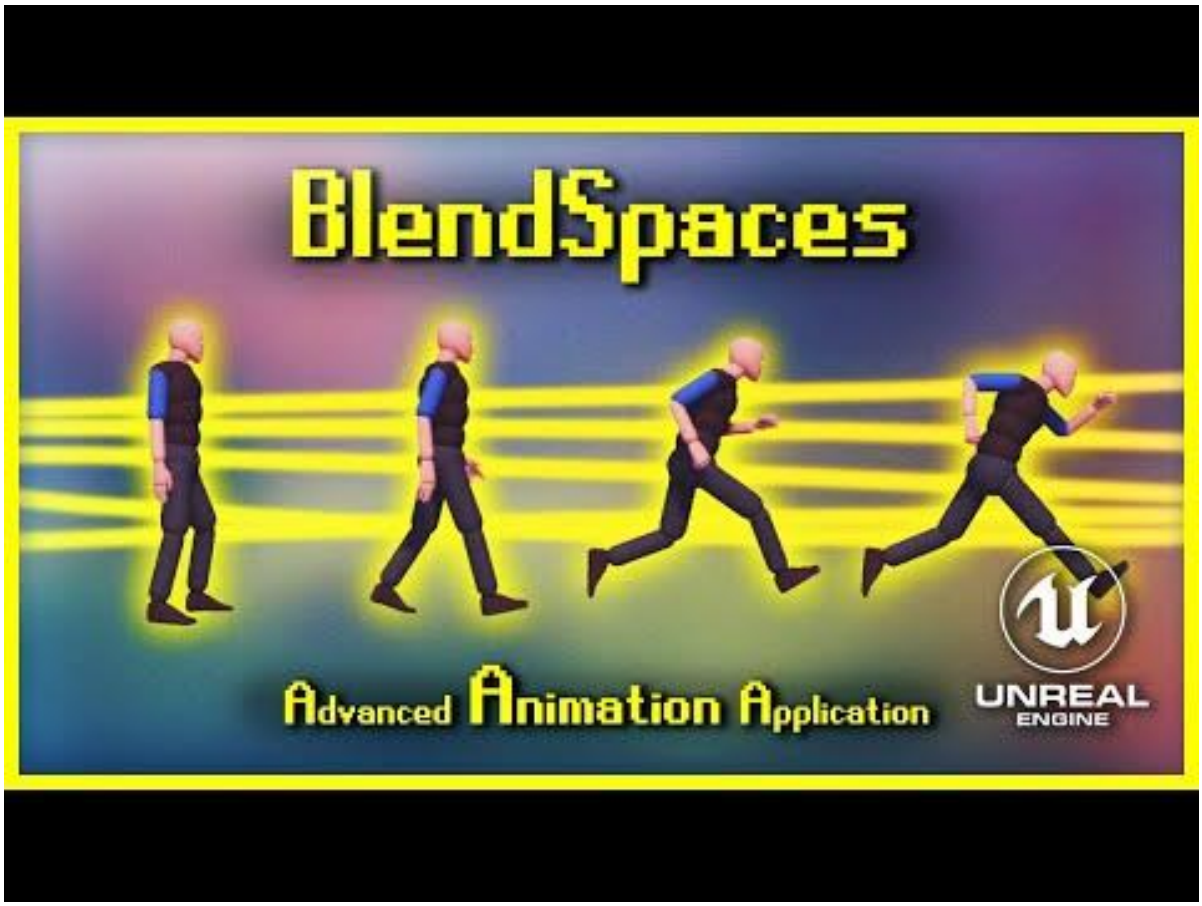
A *Physical Animation* pode ser implementada por meio de componentes como o Physical Animation Component na *Unreal Engine*. Esse componente permite ajustar a influência da física nas animações, proporcionando um equilíbrio entre o controle artístico desejado e os efeitos realistas gerados pela simulação física. Ao incorporar a *Physical Animation*, os desenvolvedores podem alcançar um nível mais elevado de imersão e autenticidade nas animações de seus jogos.

2.3.2.7 Blendspace

O *Blendspace* na *Unreal Engine* é um recurso que permite transições fluidas entre diferentes poses ou animações, proporcionando uma animação contínua e realista para personagens em movimento. Na Figura 22, à esquerda, temos uma pose de descanso, enquanto à direita, temos uma pose de corrida. Entre essas duas poses, existem duas poses intermediárias que representam transições graduais.

O *Blendspace* permite criar uma representação visual dessas transições em uma única interface. Em um eixo, geralmente horizontal ou vertical, temos a variação entre as poses de descanso e corrida. À medida que movemos ao longo desse eixo, a animação do personagem muda suavemente, criando uma transição contínua. Esse recurso é particularmente útil para ajustar a velocidade ou intensidade de uma animação com base em variáveis dinâmicas, como a velocidade do personagem em um jogo.

Figura 22 - Blendspace



Fonte: PRISMATICADEV, 2021.

2.3.2.8 Montage

Na *Unreal Engine*, uma *Montage* é um recurso que permite criar sequências de animações encapsuladas, conhecidas como "montagens de animação". Essas montagens são especialmente úteis para gerenciar transições suaves entre diferentes animações, como ataques, ações ou movimentos específicos do personagem.

Uma *Montage* pode incluir vários segmentos de animação, cada um representando uma parte específica do movimento. Por exemplo, em um jogo de luta, uma *Montage* pode abranger animações para começar um soco, realizar o golpe principal e concluir o movimento de forma fluida. Esses segmentos podem ser reproduzidos sequencialmente ou em resposta a eventos específicos no jogo.

Além disso, as Montagens oferecem controle sobre a temporização, permitindo ajustar a velocidade de reprodução, interromper ou misturar transições entre animações.

2.3.2.9 Animation Blueprint

O Animation *Blueprint* na *Unreal Engine* é um componente crucial para o desenvolvimento de animações complexas e interativas. Ele representa uma parte do sistema de animação da *Unreal Engine* e é utilizado para controlar e gerenciar as animações de personagens ou objetos no jogo.

O Animation *Blueprint* contém lógica visual baseada em nós (*Blueprint* Graph) que permite definir como as animações respondem a diferentes eventos, estados ou condições do jogo. Ele é frequentemente usado para criar transições suaves entre diferentes animações, como movimentos de caminhada, corrida, ataques, entre outros.

Essa abordagem baseada em *Blueprint* oferece aos desenvolvedores a capacidade de criar animações complexas sem a necessidade de programação extensiva. Os nós no *Blueprint Graph* representam lógica visual que pode ser conectada para controlar o fluxo da animação com base em variáveis, entradas do jogador ou eventos do jogo.

Em resumo, o Animation *Blueprint* na *Unreal Engine* é uma ferramenta poderosa para criar e gerenciar animações de forma flexível e eficiente, proporcionando aos desenvolvedores controle total sobre o comportamento animado de personagens e objetos no jogo.

2.3.2.10 Behavior Tree

Uma *Behavior Tree* (Árvore de Comportamento) na *Unreal Engine* é uma estrutura gráfica usada para definir e gerenciar a lógica de comportamento de Inteligência Artificial (IA) em jogos. Ela oferece uma abordagem visual para representar as decisões e ações que uma entidade controlada pela IA deve realizar em diferentes situações.

A *Behavior Tree* organiza o comportamento da IA em uma hierarquia de nodos, onde cada nodo representa uma ação, uma condição ou uma lógica de tomada de decisão. Esses nodos são interconectados para formar uma árvore que define o fluxo de comportamento da IA. A estrutura da árvore permite que diferentes partes do comportamento sejam modificadas, expandidas ou ajustadas de forma modular.

Cada nodo na *Behavior Tree* pode representar uma ação específica, como atacar, se movimentar, buscar cobertura, entre outras, ou uma condição para avaliar o ambiente e determinar qual ação executar em seguida. A árvore é percorrida de cima para baixo, seguindo as condições e ações de acordo com as regras definidas.

As *Behavior Trees* são amplamente utilizadas para controlar o comportamento de NPCs (Personagens Não Jogáveis) e outros elementos controlados por IA em jogos. Elas proporcionam uma maneira intuitiva e eficiente de projetar, modificar e depurar o comportamento da IA

3 O JOGO

Neste capítulo, adentramos na fase prática do trabalho, delineando o passo a passo que culminou na concepção e desenvolvimento de um jogo teórico e genérico. Este capítulo não apenas apresenta o resultado tangível do projeto, mas também serve como um guia instrutivo para aqueles que desejam explorar a criação de jogos semelhantes.

Ao longo deste capítulo, desvendaremos a seleção criteriosa de técnicas e algoritmos aplicados no projeto, destacando sua aplicação prática e os resultados almejados.

3.1 GAME DESIGN DOCUMENT

Game Design Document (GDD), ou documento de Design de Jogo, refere-se a um texto documental, geralmente ricamente ilustrado, elaborado por um designer de jogos. Esse documento abrange diversos elementos do jogo, como estética, narrativa e mecânicas, servindo como meio de comunicação e orientação para os participantes no processo de desenvolvimento do jogo. A falta de um padrão definido para esse artefato resulta em várias discussões, pois cabe ao designer de jogos escolher um modelo de documento que melhor atenda às exigências do jogo.

O GDD é considerado um componente crucial no desenvolvimento de jogos, fornecendo uma estrutura sólida para a concepção e execução de um projeto. Este documento abrangente detalha as características essenciais, mecânicas, narrativa, arte, som e todos os aspectos pertinentes do jogo que está sendo desenvolvido (MOTTA & JUNIOR, 2013).

Para que se possa dar continuidade ao trabalho, deve-se primeiramente ser estabelecido um jogo exemplo onde técnicas e algoritmos poderão ser aplicados.

3.1.1 Ideia Geral

A ideia genérica para o jogo será de um jogo do gênero *First Person Shooter* (FPS) de terror, ambientado em intrincados sistemas de cavernas. Dentro da caverna existe um monstro que está constantemente caçando os jogadores.

A premissa do jogo consiste na experiência dos jogadores que se encontram perdidos em um ambiente de cavernas complexo, onde um monstro, notoriamente

mais forte e ágil que os jogadores, impõe uma vantagem desafiadora. Cada possível confronto com o monstro exige precisão estratégica, ressaltando a opção recuar como uma alternativa viável.

3.1.2 Considerações Artísticas

A ênfase no desenvolvimento recai na busca por uma imersão completa dos jogadores. O jogo proposto visa incorporar cenários e animações de elevado realismo e atratividade, com o intuito de cativar o jogador e proporcionar uma experiência envolvente. Esta imersão tem o propósito de intensificar o sentimento de medo ao se deparar com o monstro, tornando a experiência mais visceral e impactante. A utilização de contrastes entre claro e escuro no cenário também será implementada, visando criar um ambiente misterioso e perigoso, contribuindo para a atmosfera de suspense do jogo.

3.1.3 Objetivo

A vitória do jogo é alcançada quando o jogador, mediante estratégia, elimina a ameaça monstro. Em contrapartida, caso o monstro vença, resultando na eliminação do jogador, a partida é considerada perdida.

3.1.4 Trabalhos relacionados

Dentre os trabalhos relacionados à dinâmica entre Monstro e Jogador, destacam-se Alien: Isolation e Evolve. No contexto de Alien: Isolation, o monstro Xenomorph ganha notoriedade por sua inteligência artificial complexa, proporcionando experiências incríveis ao jogador. O jogo mantém um constante clima de suspense, permitindo que o jogador supere desafios e conclua os níveis. Por outro lado, em Evolve, quatro jogadores exploram um ambiente desconhecido com o objetivo de caçar e eliminar um monstro notavelmente mais poderoso. Nesse cenário, o monstro, também controlado por um jogador, precisa fugir, consumir outros animais do planeta para evoluir e se tornar progressivamente mais formidável, culminando na inversão do papel, onde ele passa a caçar e eliminar os jogadores.

Quanto à concepção do mapa, referências foram extraídas de jogos como GTFO e *Deep Rock Galactic. Deep Rock Galactic* destaca-se pela criação de mundos

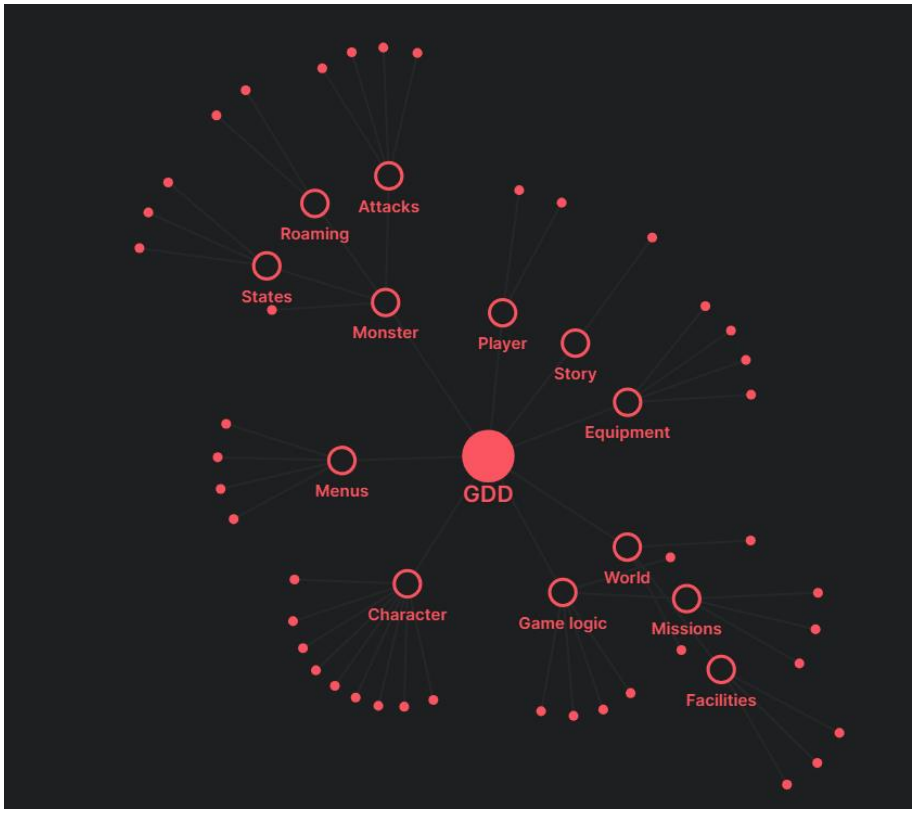
totalmente procedurais e organicamente dinâmicos a cada partida, oferecendo desafios e experiências novas a cada jogada. Por outro lado, GTFO adota uma abordagem procedural na geração de mapas, construindo ambientes por meio de salas posicionadas de forma randômica. Essa estratégia, ao reutilizar salas previamente criadas, contribui para a formação de ambientes inéditos e cativantes a cada incursão no jogo.

3.1.5 Processo de construção

A fase inicial no desenvolvimento do jogo envolve a elaboração do GDD que detalha de maneira abrangente o funcionamento do jogo, seus requisitos funcionais e não funcionais. Isso inclui definições para objetivos, condições de vitória e derrota, além de abranger aspectos que envolvem o início e o término do jogo.

Conforme destacado por Motta e Junior (2013), a abordagem na elaboração do GDD varia entre diferentes jogos. No âmbito deste trabalho, a escolha recaiu sobre a utilização da plataforma online denominada Nuclino, reconhecida por suas diversas opções de visualização para o GDD, exemplificadas na figura 23, a qual apresenta uma representação gráfica organizada dos elementos do GDD em formato de grafo.

Figura 23 - Game Design Document



Fonte: Autoria própria

Após a definição do tema do jogo, estabelecimento dos objetivos e elaboração do GDD, avança-se para a crucial etapa de determinar o processo de construção do software. Ao analisar os requisitos essenciais do jogo genérico, tomou-se a decisão estratégica de iniciar o desenvolvimento pelas armas do jogo. Essa escolha se fundamenta na necessidade primordial de criar animações para o personagem segurando as armas, estabelecer um sistema de dano causado por essas armas antes de conceber o monstro, e até mesmo considerar o mapa em relação às partículas e interações que as armas podem realizar com o ambiente. Dessa maneira, a sequência de construção foi estabelecida da seguinte forma:

1. Armas
2. Personagem
3. Monstro
4. Mapa

É relevante ressaltar que, embora a locomoção do personagem e do monstro esteja intrinsecamente ligada à concepção do mapa, foi possível adotar *placeholders* iniciais como medida temporária. Essa abordagem permite a continuidade do desenvolvimento, facilitando a posterior integração de movimentação personalizada conforme o mapa é aprimorado.

3.2 ARMAS

O plano de jogo adotado destaca a importância da fidelidade visual aos ambientes da vida real, um princípio que se estende à escolha das armas do personagem. Em contraste com muitos jogos que apresentam armamentos futuristas e ferramentas inovadoras, a decisão foi orientada pela busca de fidelidade visual, optando por armas contemporâneas que os jogadores possam reconhecer facilmente. Para alcançar esse objetivo, escolheu-se trazer armas existentes para o jogo, conferindo-lhes um visual modernizado.

Este trabalho oferecerá um guia detalhado do processo de construção da arma *Winchester*", destacando que o mesmo procedimento será aplicado a todas as outras armas do jogo, com ajustes mínimos. O primeiro passo para modelar um *asset* para o jogo consiste em realizar uma pesquisa extensiva por referências, explorando

imagens e trabalhos similares para obter uma compreensão abrangente do objeto que será modelado.

Ao buscar referências para a *Winchester*, foram identificados e analisados os elementos essenciais que definem sua aparência e características distintivas. A figura 24, representando uma Winchester padrão, servirá como exemplo visual ao longo do processo de modelagem, oferecendo uma referência concreta para orientar a construção do *asset* com precisão e autenticidade. Adicionalmente, a figura 25 apresenta uma versão da Winchester com visual modernizado, ampliando as opções de referência e inspiração para o desenvolvimento das armas do jogo.

Figura 24 - Winchester

Fonte: NSFBLOG, 2023

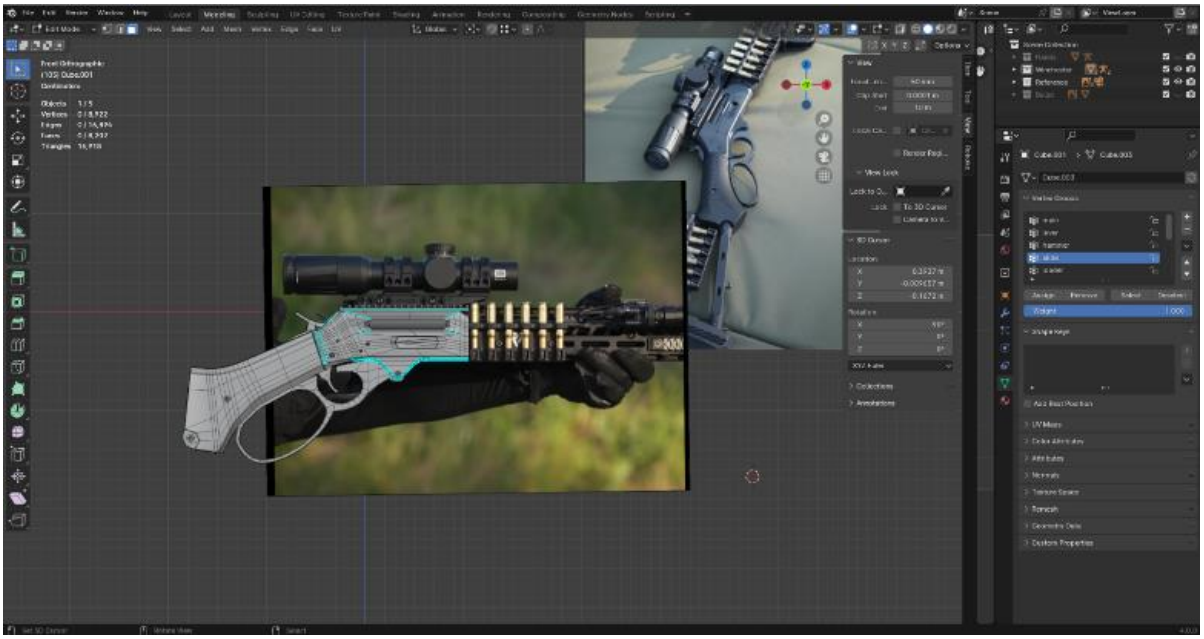
Figura 25 - Winchester Moderna

Fonte: JAEGERZ999

3.2.1 Modelo e Animações

Após reunir as referências necessárias, o processo de modelagem teve início, sendo adotada a técnica de alinhar a modelagem da *Winchester* com referências laterais da arma. Essa abordagem visa minimizar as chances de desalinhamento de escala e assegurar proporções adequadas à arma, contribuindo para um resultado mais preciso. A Figura 26 ilustra esse processo, evidenciando a modelagem sobreposta à referência.

Figura 26 - Modelagem sobreposta em referência



Fonte: Autoria própria

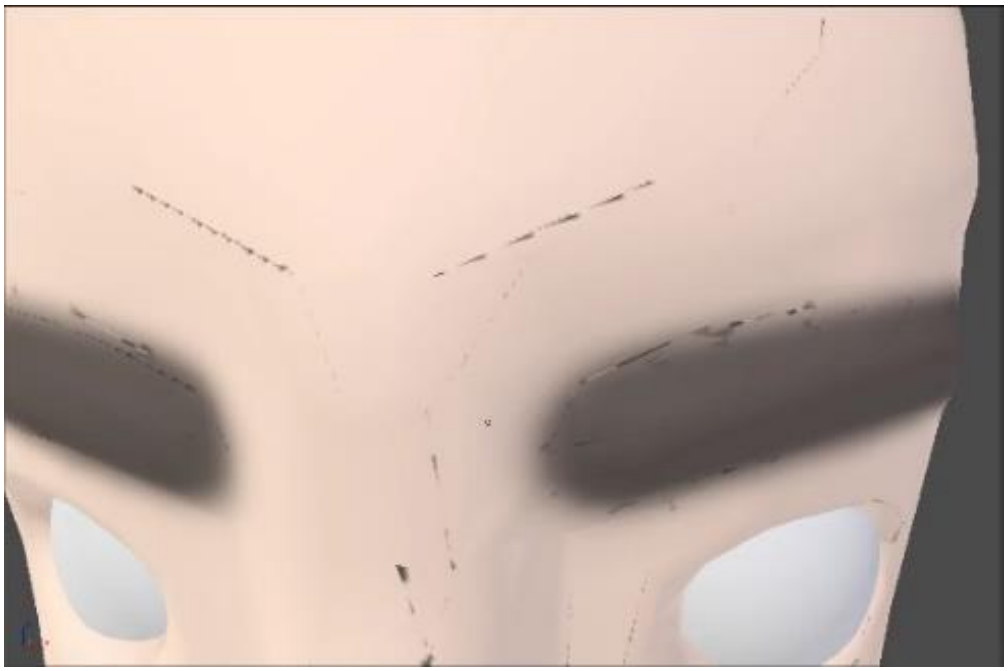
Uma escolha estratégica foi separar a arma dos acessórios, permitindo a união posterior na Unreal. Essa abordagem oferece flexibilidade ao jogador, que pode personalizar a arma com diferentes acessórios (*attachments*), cada um com vantagens e desvantagens. A Figura 27 apresenta a modelagem finalizada da arma *Winchester*, incluindo todos os *attachments* relevantes.

Figura 27 - Modelo da Winchester

Fonte: Autoria própria

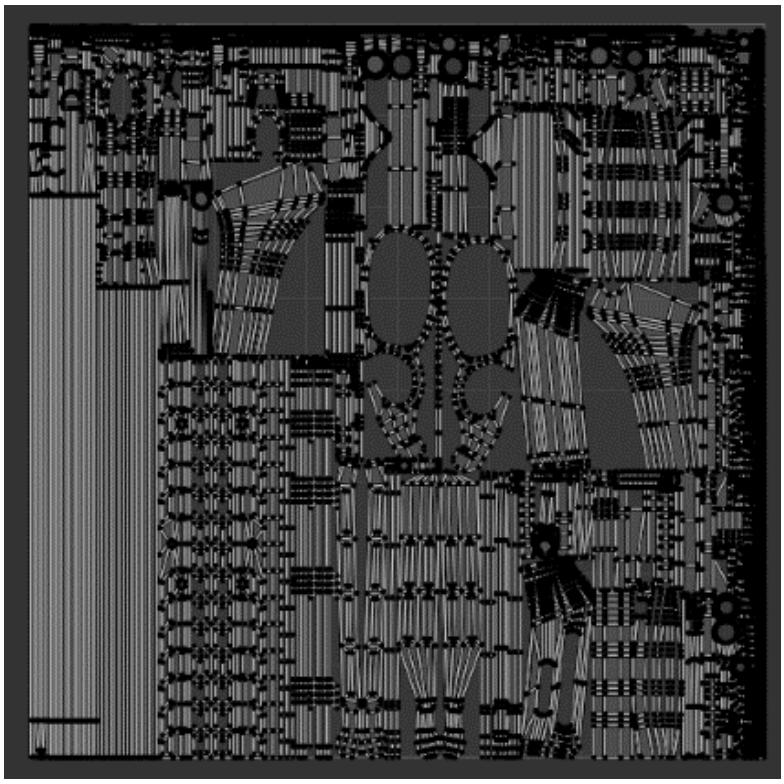
Devido à natureza inorgânica das armas, a UV do modelo foi desdobrada sem grande ênfase na preservação de movimento. O foco principal foi evitar a proximidade excessiva entre as faces para prevenir vazamento de texturas entre elas, o que poderia resultar em artefatos durante a renderização, como pode ser evidenciado no exemplo da Figura 28. A Figura 29 exibe o UV *unwrap* da arma modelada.

Figura 28 - Falha em razão de UV



Fonte: XOREALIS, 2023

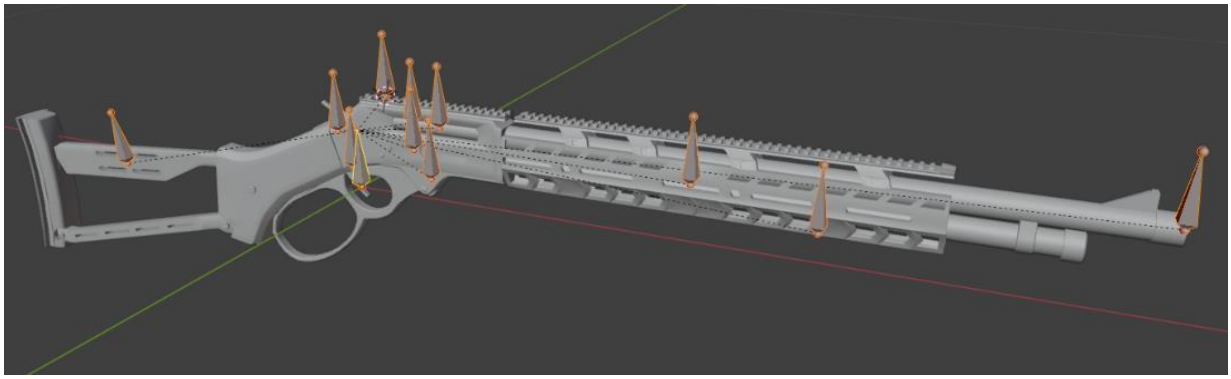
Figura 29 - UV da winchester



Fonte: Autoria própria

Com o modelo finalizado, passou-se à etapa de *rigging*. Apesar da ausência de um esqueleto análogo ao humano, a criação de um esqueleto abstrato para os movimentos da *Winchester* foi essencial para a animação da arma, incluindo ações como atirar e recarregar. O *Weight Paint* do modelo sólido foi simplificado, uma vez que ossos têm influência total ou nula sobre partes específicas da arma. A definição de ossos em locais estratégicos, como o bico do cano e pontos de conexão para acessórios, facilita a manipulação posterior dentro da *Engine*. A Figura 30 destaca a configuração dos ossos de movimento e auxiliares no modelo da *Winchester*.

Figura 30 - Armadura da Winchester



Fonte: Autoria própria

Com o modelo exportado, o processo de texturização ocorreu no Substance Painter, seguindo o padrão PBR. Camadas de albedo, *roughness, normals* e *metaliness* foram criadas, resultando em texturas de alta qualidade e fidelidade à realidade. Texturas em tamanho 4k foram geradas para proporcionar um nível de detalhamento mais elevado. A Figura 31 exemplifica o nível de texturização alcançado na *Winchester*.

Figura 31 - Textura no Substance Painter

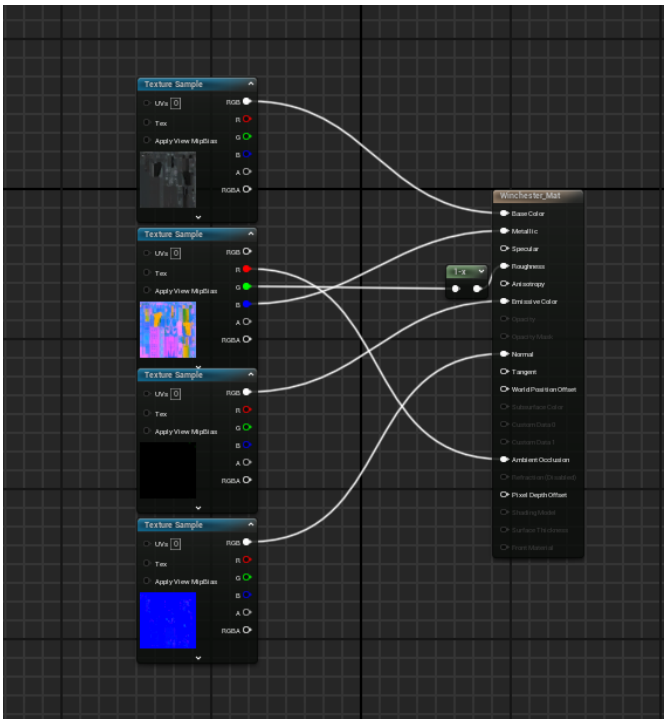
Fonte: Autoria própria

A integração do modelo na Unreal começou pela importação do arquivo FBX do Blender e das texturas do Substance Painter. Antes de visualizar o resultado, a criação do Material foi essencial. Este define as propriedades do PBR para o objeto, como cor, reflexão, transparência, normais, entre outras. A Figura 32 mostra o resultado do modelo texturizado dentro da Unreal, exibindo um nível de detalhe satisfatório. Por fim, a Figura 33 destaca a configuração do material utilizando as texturas PBR na *Unreal Engine*.

Figura 32 - Winchester dentro da Unreal

Fonte: Autoria própria

Figura 33 - Material da Winchester



Fonte: Autoria própria

3.2.2 Programação

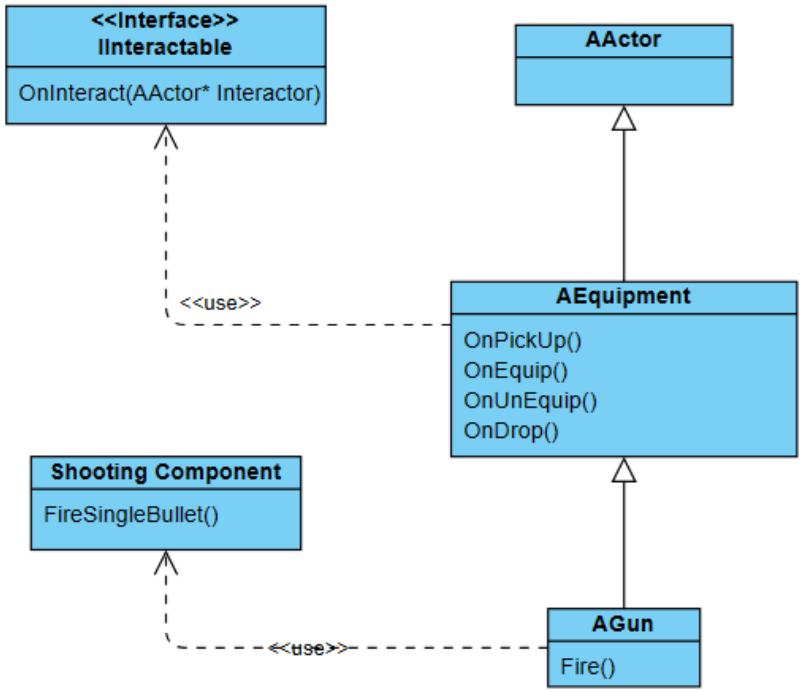
Concluída a fase de desenvolvimento e importação do modelo da *Winchester,* torna-se imperativa a abordagem da programação associada à arma.

Considerando a possibilidade de o personagem possuir diversos equipamentos, como lanternas, granadas, armadilhas e armas, optou-se por uma abordagem estruturada. Foi desenvolvida uma classe principal denominada *Equipment*, que contém funções relacionadas ao equipamento e à interação do jogador com o objeto. A partir desta classe mãe, foi derivada a classe *Gun*, que herda

algumas funcionalidades da classe *Equipment* e, além disso, sobrecarrega determinados métodos específicos.

A classe *Gun* tem a capacidade de invocar a função *ShootSingleBullet*, a qual está implementada no componente *ShootingComponent*. Esse componente é acessível pela *Gun* e tem como função executar o disparo de balas. A forma como as classes foram estruturadas pode ser evidenciada na Figura 34.

Figura 34 - Diagrama de classe simplificado da Winchester

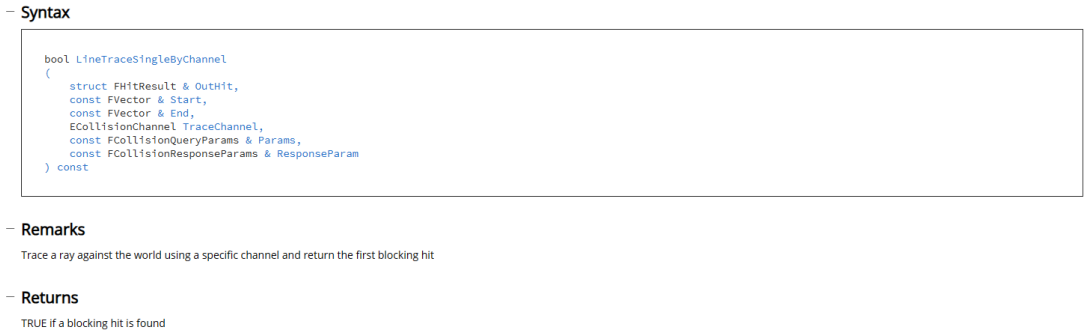


Fonte: Autoria própria

Para a implementação da função de tiro no componente *ShootingComponent*, faz-se uso da função *LineTraceSingleByChannel* da *Unreal Engine*. Essa função realiza um traçado de raio a partir de um ponto de origem até um ponto de destino, identificando e retornando qualquer objeto que o raio encontrar durante seu percurso. Na estruturação da função, são necessários parâmetros específicos, como um *struct* do tipo *FHitResult*, que, por referência, armazenará os dados da colisão; o ponto de início e o ponto final do raio; e o canal de colisão apropriado.

A Figura 35, extraída da documentação oficial da *Unreal Engine*, apresenta de maneira sintática a estrutura e o retorno da função *LineTraceSingleByChannel*.

Figura 35 - Documentação da função LineTraceSingleByChannel



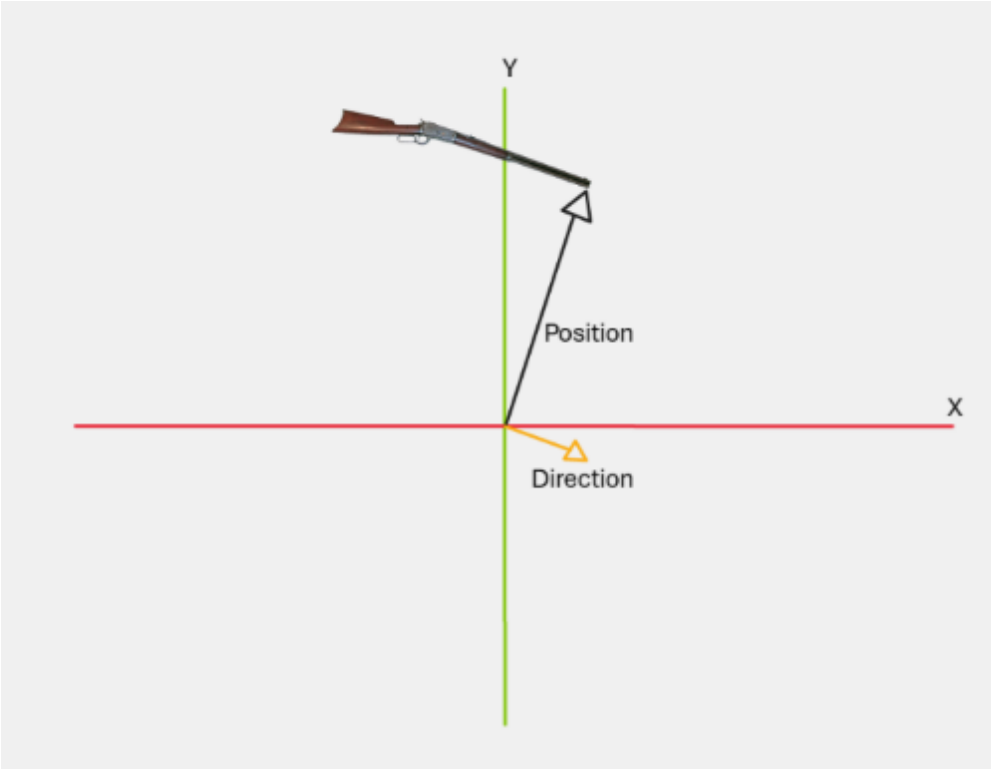
Fonte: Documentação da Unreal

Na execução da função *LineTraceSingleByChannel*, é essencial configurar adequadamente os parâmetros para garantir um traçado de raio preciso e eficaz.

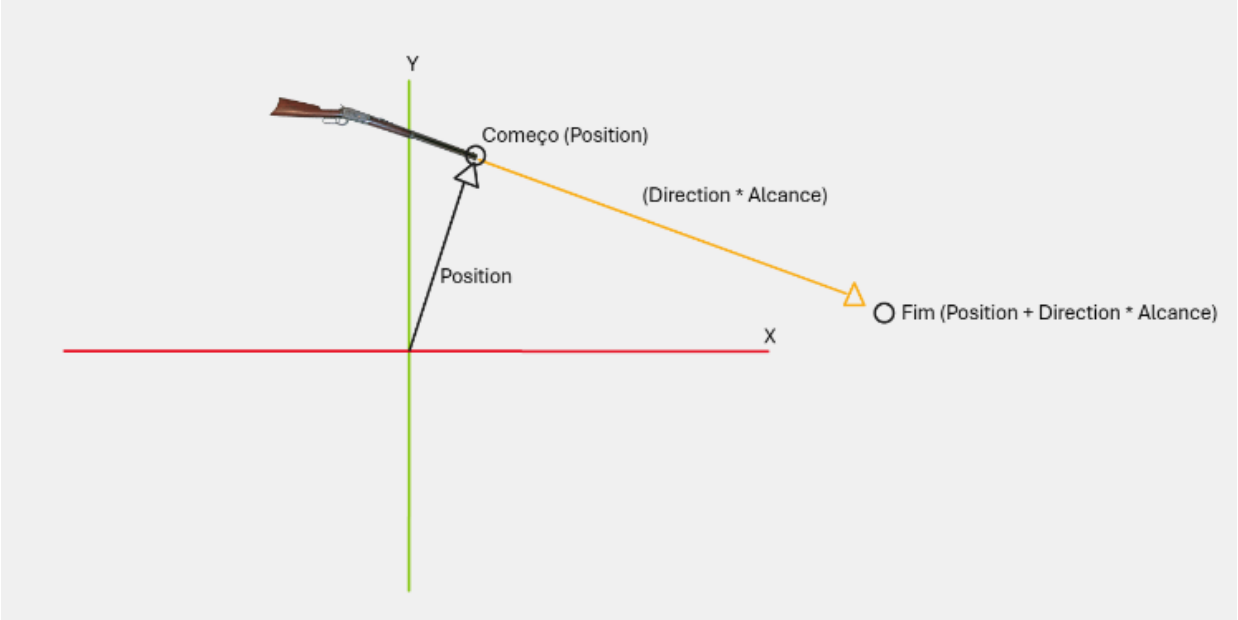
Para o canal de colisão, optamos pelo *ECollisionChannel::ECC_Visibility*, permitindo que qualquer entidade visível seja considerada para a colisão. Os pontos de início e fim do raio são calculados por meio de vetores. O ponto de início corresponde à posição do cano da arma, enquanto o vetor unitário de direção é obtido utilizando a função *GetActorForwardVector* do objeto arma. Multiplicando esse vetor unitário pelo alcance desejado para a bala e somando-o ao vetor da posição do cano, alcançamos o ponto de destino do raio.

As Figuras 36 (a) e (b) oferecem uma representação visual do processo de cálculo vetorial para a implementação bem-sucedida da função LineTraceSingleByChannel no contexto do jogo.

Figura 36 - Cálculo vetorial da trajetória da bala



a)



b)

Fonte: Autoria própria

Após a execução bem-sucedida da função *LineTraceSingleByChannel*, e identificando uma colisão com um ator, o método *GetActor* do *struct FHitResult*

oferece um ponteiro para o ator envolvido. Utilizando a Figura 37 como exemplo, que ilustra a chamada da função *TakeDamage*, podemos entender como esse processo é aplicado.

Ao detectar uma colisão, a função *TakeDamage*, nativa da *Unreal Engine*, é acionada no ator identificado pela colisão. Esse processo cria um evento de dano no ator, especificando informações adicionais, como a quantidade de dano a ser aplicada, o Controlador que desencadeou o dano e o próprio ator responsável pela causa do dano.

Figura 37 - Código de aplicação de dano

```
if (OutHit.GetActor())
{
    //GEngine->AddOnScreenDebugMessage(INDEX_NONE, 5.f, FColor::Green, FString(("Actor Hit: " + OutHit.GetActor()->GetName())));

    // Apply damage
    FPointDamageEvent PointDamageEvent(Damage, OutHit, Direction, UDamageType::StaticClass());
    OutHit.GetActor()->TakeDamage(Damage, PointDamageEvent, GetOwner() ? GetOwner()->GetInstigatorController() : nullptr, GetOwner());
}
```

Fonte: Autoria própria

3.3 PERSONAGEM

Esta fase do processo foi dedicada a uma etapa intricada. Com o objetivo de atingir gráficos de alta fidelidade e animações orgânicas e realistas, foi empreendido um esforço considerável na criação de um extenso conjunto de animações, assim como foi escolhido um aspecto visual específico para os modelos do jogador.

3.3.1 Modelo e animações

A representação visual do personagem no jogo adota uma abordagem específica, utilizando apenas um par de braços flutuantes, sem a presença do corpo, como pode ser visualizado na Figura 38. Essa escolha é motivada pela perspectiva em primeira pessoa adotada no jogo, eliminando a necessidade de renderizar um corpo completo. Essa estratégia não apenas otimiza o processo de desenvolvimento, economizando tempo, mas também elimina a exigência de animações complexas para o corpo inteiro do personagem. Essa técnica direciona o foco para os membros superiores, proporcionando uma visão clara das ações do jogador, sem comprometer a experiência visual do jogo.

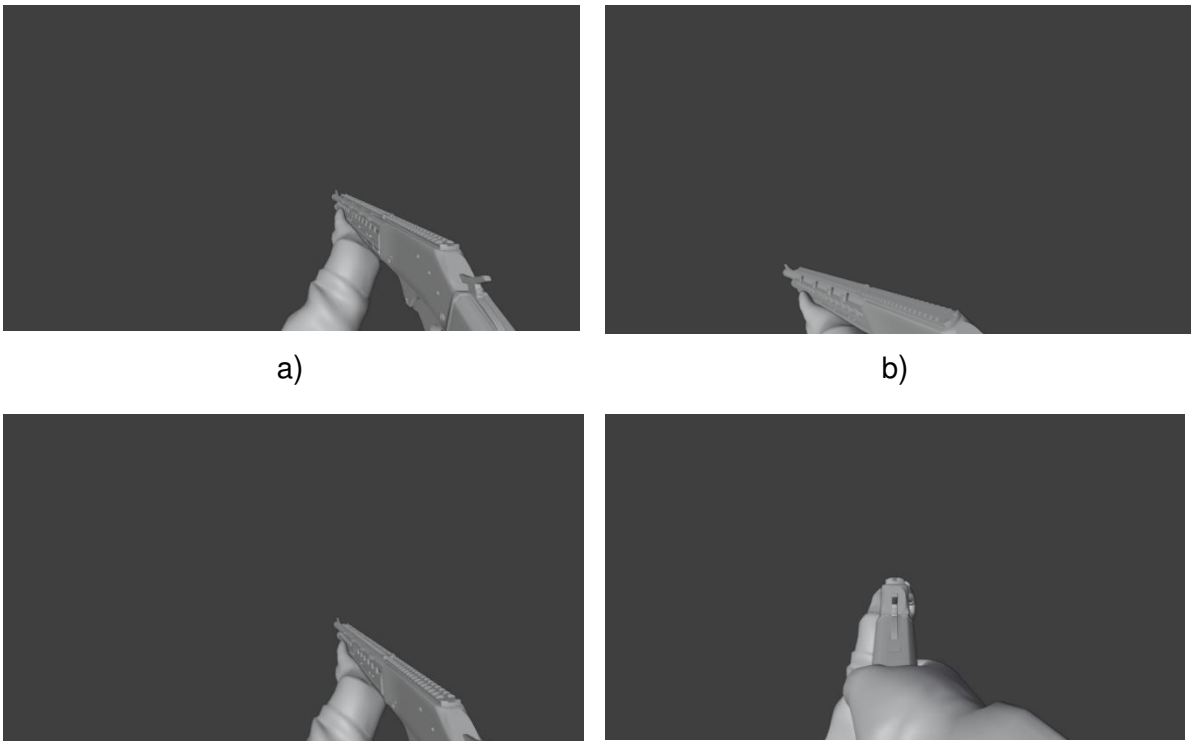
Figura 38 - Braços do jogador dentro da Unreal

Fonte: Autoria própria

Inicialmente, foram identificados diversos tipos de movimentos que o jogador poderia executar, incluindo andar, correr, mirar e agachar. Para cada categoria de movimento, foram desenvolvidas animações para o estado estacionário e em movimento.

A Figura 39 (a), (b), (c) e (d) ilustram algumas das poses dessas animações, representando respectivamente as animações de andar, correr, mirar e agachar. Utilizando o sistema de *blendspace* da Unreal, permitiu-se uma interpolação suave entre as animações estacionárias e as de movimento. Associando esse processo a uma variável, é possível controlar o grau de transição entre as animações. Quando a variável é ajustada para 0, apenas a animação estacionária é exibida. Quando definida como 1, a única animação visível é a de movimento. Em valores intermediários, como 0.5, as animações são mescladas, criando a impressão de um movimento mais suave e lento. Essa técnica proporciona uma experiência visual coesa e dinâmica para o jogador, contribuindo para a imersão no ambiente do jogo.

Figura 39 - Animações do personagem



a) b)

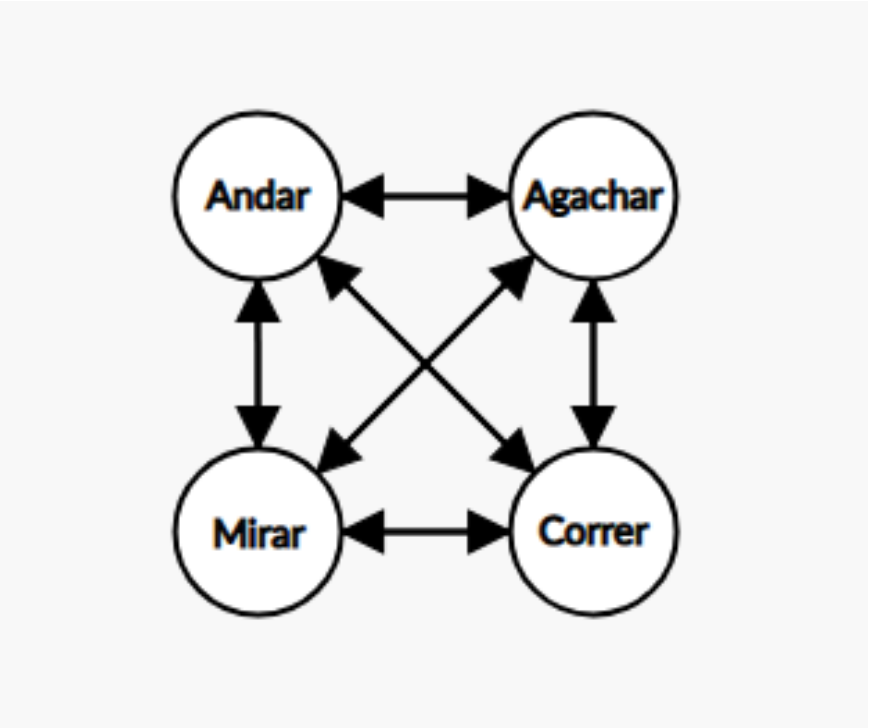
c) d)

Fonte: Autoria própria

Após a criação das animações para os quatro estados fundamentais, é crucial estabelecer transições fluidas entre esses estados. A abordagem padrão da *Unreal Engine* envolve uma interpolação linear das poses entre os estados, porém, essa técnica pode resultar em transições visualmente artificiais. Optei por uma abordagem mais meticulosa, que consiste em exaustivamente animar manualmente cada uma das transições possíveis entre os quatro estados de animação.

A Figura 40, representando um grafo com nós que indicam os estados possíveis de animação e arestas que denotam todas as transições entre esses estados, ilustra essa complexa rede de interconexões. Analogamente, se considerarmos os estados como nós em um grafo, todas as animações relativas às arestas de um grafo completo foram desenvolvidas. Esse cuidadoso processo resultou em um conjunto de 12 animações de transição, garantindo que as mudanças entre os estados de animação sejam visualmente naturais e realistas.

Figura 40 - Estados de animações

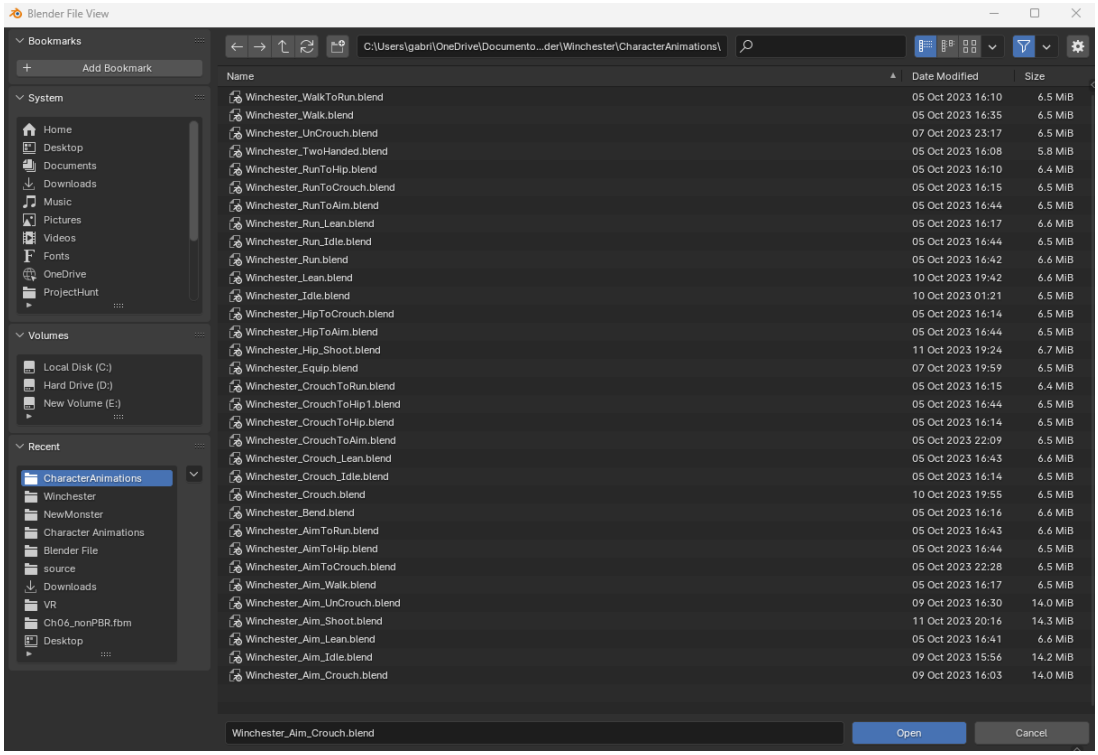


Fonte: Autoria própria

Após o minucioso processo de animação, as transições entre estados e movimentos da arma Winchester foram refinadas para garantir uma experiência visualmente autêntica e fiel à movimentação humana. A Figura 41 detalha todas as 33 animações, abrangendo transições entre estados, animações de estado e ações específicas, como pular e atirar.

O resultado reflete a atenção aos detalhes e a busca pela organicidade nos movimentos. Além disso, para refinar ao máximo e introduzir um nível adicional de realismo, foram empregadas técnicas de animações aditivas, contribuindo para uma representação mais natural e fluida das ações do jogador no jogo.

Figura 41 - Animações da Winchester

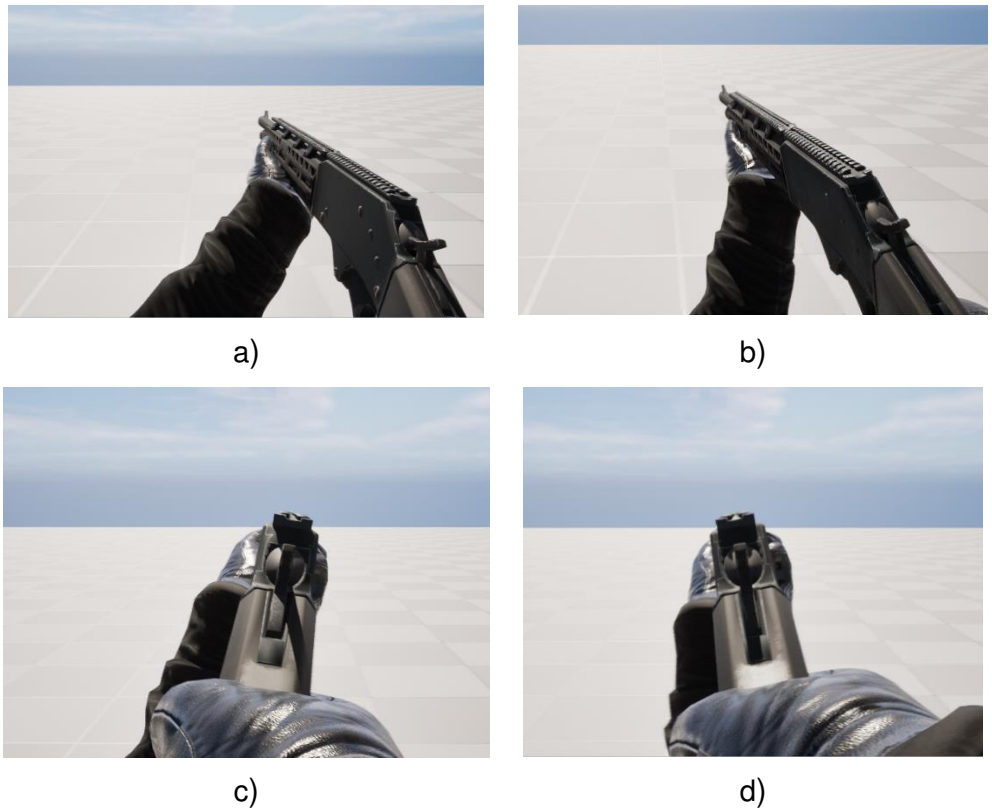


Fonte: Autoria própria

No refinamento das animações, as Figuras 42 (a) e (b) ilustram poses específicas da animação de andar, destacando a inclinação da arma para a direita e esquerda, respectivamente. Essa distinção é resultado da aplicação de poses aditivas que respondem à movimentação atual do personagem. Da mesma forma, as Figuras 42 (c) e (d) representam as animações de mirar, onde a arma se inclina para a direita e esquerda, ajustando-se dinamicamente às nuances do movimento.

A aplicação estratégica de animações aditivas visa enfatizar a movimentação do personagem, proporcionando uma diferenciação clara entre os movimentos para direita, esquerda, frente e trás, mesmo quando a mesma animação está em execução. Essa técnica assegura uma resposta ágil aos inputs do jogador, tornando evidente a dinâmica do personagem e elevando a organicidade dos movimentos. Animações aditivas foram incorporadas em todas as poses para os quatro ângulos de movimento (frente, direita, esquerda, trás), contribuindo para uma experiência de jogo mais imersiva e fluida.

Figura 42 - Animações aditivas



a) b)

c) d)

Fonte: Autoria própria

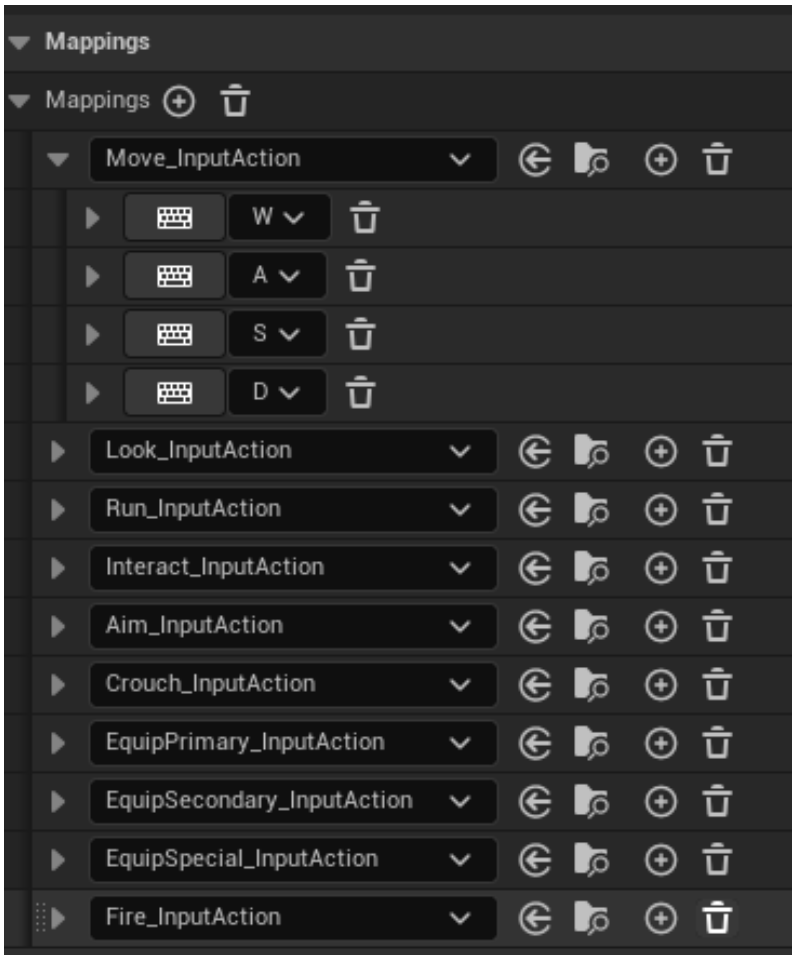
As Montagens na *Unreal Engine* são usadas para gerenciar e reproduzir animações de forma dinâmica, permitindo transições suaves entre diferentes estados. No contexto do jogo, Montagens foram implementadas para controlar as animações de tiro e recarregamento da arma. Isso oferece uma integração eficiente entre os eventos do jogo, como atirar e recarregar, e as animações correspondentes do personagem.

3.3.2 Programação

Na criação do personagem, o processo se inicia desenvolvendo uma classe derivada de *Character*. A primeira etapa envolve a configuração da captura de inputs do jogador. É importante destacar que a *Unreal Engine* suporta tanto inputs digitais, como teclas do teclado, quanto inputs analógicos, como os joysticks de controles de consoles. Para capturar inputs, é necessário configurar uma classe chamada *InputAction* na *Unreal Engine*. Essa classe armazena informações sobre os inputs a serem observados e pode realizar transformações nos valores recebidos. Por

exemplo, é possível remapear o input de um joystick de uma escala de 0 a 1 para -1 a 0, proporcionando flexibilidade na manipulação dos controles (Figura 43).

Figura 43 - Input Actions



Fonte: Autoria própria

Na etapa subsequente, é viável associar funções específicas às *Input Actions*, de modo que essas funções sejam acionadas sempre que ocorrer um input registrado. Por exemplo, para lidar com a movimentação e rotação da câmera do personagem, são vinculadas funções que são chamadas a cada frame em resposta aos inputs capturados (Figura 44). Essa vinculação entre *Input Actions* e funções contribui para a integração eficaz da interatividade do jogador com o personagem no jogo como pode ser visto na (Figura 45) mostrando a definição das funções chamadas a partir dos *inputs* do jogador.

Figura 44 - Configuração de callbacks

```
if (!InputLookAction.IsNull())
    Input->BindAction(InputLookAction.LoadSynchronous(), ETriggerEvent::Triggered, this, &APHCharacter::Look);

if (!InputMoveAction.IsNull())
    Input->BindAction(InputMoveAction.LoadSynchronous(), ETriggerEvent::Triggered, this, &APHCharacter::Move);
```

Fonte: Autoria própria

Figura 45 - Definição dos callbacks

```
void APHCharacter::Look(const FInputActionValue& value)
{
    AddControllerYawInput(value[0]);
    AddControllerPitchInput(-value[1]);
}

void APHCharacter::Move(const FInputActionValue& value)
{
    AddMovementInput(GetActorForwardVector(), value[0]);
    AddMovementInput(GetActorRightVector(), value[1]);
}
```

Fonte: Autoria própria

3.4 MONSTRO

Essa etapa do trabalho é dedicada à criação, configuração e programação de um monstro com aparência e movimentação realistas e convincentes. Aqui é uma etapa onde grande parte dos resultados foram alcançados a partir de muitos testes e experimentações, tanto com animações quanto com a movimentação do monstro, deixando claro a enorme necessidade de testes e retestes durante o processo de desenvolvimento de um jogo.

3.4.1 Modelo e animações

Ao conceber o monstro, a intenção era evocar uma sensação de amedrontamento, inspirada no Xenomorph de Alien: Isolation. Optei por adquirir um modelo tridimensional e texturas já elaboradas, disponíveis em um mercado online chamado TurboSquid (Figura 46). Este site facilita a compra e venda de *assets* 3D, proporcionando uma abordagem eficiente para desenvolvedores, especialmente em equipes de tamanho reduzido.

Embora a escolha de *assets* prontos acelere o desenvolvimento, o modelo escolhido para o monstro não incluía animações, demandando a criação de todas as sequências animadas necessárias para conferir vida ao personagem monstruoso. Este processo de combinar ativos prontos com animações customizadas permitiu alcançar o visual desejado para o monstro no contexto do jogo.

Figura 46 - Monstro

Fonte: CECOALIENSA, 2017

Na concepção do monstro, a primeira etapa foi estabelecer os tipos de movimento que ele seria capaz de executar, determinando, assim, as animações necessárias. Optou-se por definir três estados principais para o monstro: descanso, andar e correr. Ao contrário do jogador, cujas animações se concentram principalmente nos membros superiores devido à visão em primeira pessoa, as animações do monstro englobam o corpo inteiro, tornando-as consideravelmente mais complexas.

Para gerenciar a transição suave entre essas animações, foi aplicada a interpolação linear padrão da Unreal, uma escolha coerente dada a complexidade das

animações (Figura 47). Este processo, embora mais desafiador, permitiu alcançar animações realistas e orgânicas para o monstro, contribuindo para a atmosfera tensa e imersiva do jogo.

Figura 47 - Frame de animação de andar

Fonte: Autoria própria

3.4.2 Programação

Para conferir dinamismo e autenticidade ao monstro, foi implementado um comportamento básico utilizando o sistema *Behavior Tree* da *Unreal Engine*.

O *Behavior Tree* é uma estrutura de dados que organiza o fluxo de decisões em jogos, comumente usado para criar lógicas de IA (Inteligência Artificial). Nesse contexto, ele permite definir uma série de ações que o monstro realizará dependendo de certas condições.

Quando o monstro não tem visão do jogador, seu comportamento segue um padrão: escolher um ponto aleatório no mapa e movimentar-se em direção a esse ponto. Essa abordagem garante que o monstro esteja constantemente explorando o ambiente, aumentando as chances de encontrar o jogador e iniciar um ataque.

Quando o monstro avista o jogador, sua primeira ação é executar uma animação de rugido, proporcionando feedback visual e auditivo ao jogador, indicando que foi detectado. Além disso, esse rugido serve como um alerta, concedendo ao

jogador a oportunidade de fugir antes de ser atacado. Em seguida, o monstro inicia uma corrida em direção ao jogador, e ao alcançá-lo, executa uma animação de ataque que causa dano ao jogador.

Essa abordagem cria uma dinâmica envolvente, instigando ações estratégicas por parte do jogador e intensificando a imersão no ambiente tenso do jogo.

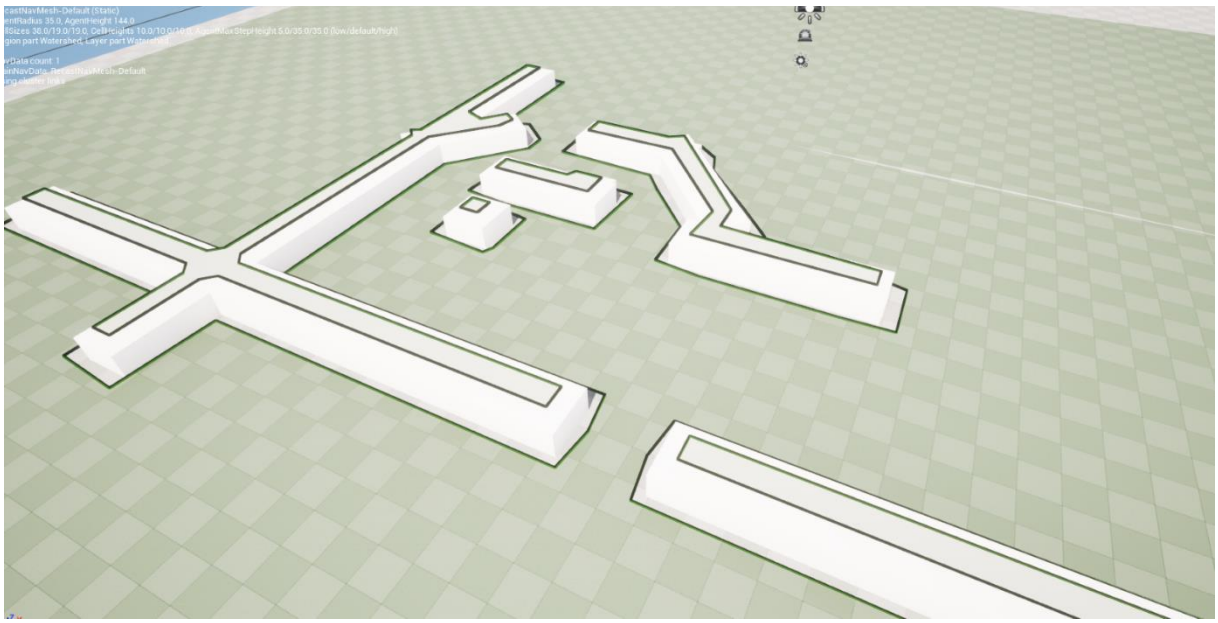
3.4.2.1 Movimentação

A implementação da movimentação do monstro foi realizada por meio do sistema de *Pathfinding* (Procura de Caminhos) integrado à *Unreal Engine*. Essa técnica é essencial para permitir que o monstro se desloque de maneira eficiente pelo ambiente do jogo, evitando obstáculos e alcançando diferentes pontos.

O primeiro passo foi adicionar um *Navigation Volume* ao mapa. Esse volume é fundamental para o processamento do *NavMesh*, uma malha que define as áreas acessíveis pelo monstro. O *NavMesh* é crucial para o cálculo de trajetórias, proporcionando ao monstro a capacidade de navegar pelo ambiente de forma inteligente, desviando de obstáculos e alcançando qualquer ponto válido dentro do *NavMesh*.

A Figura 48 ilustra a representação visual do *NavMesh* em um mapa específico, destacando as áreas navegáveis pelo monstro.

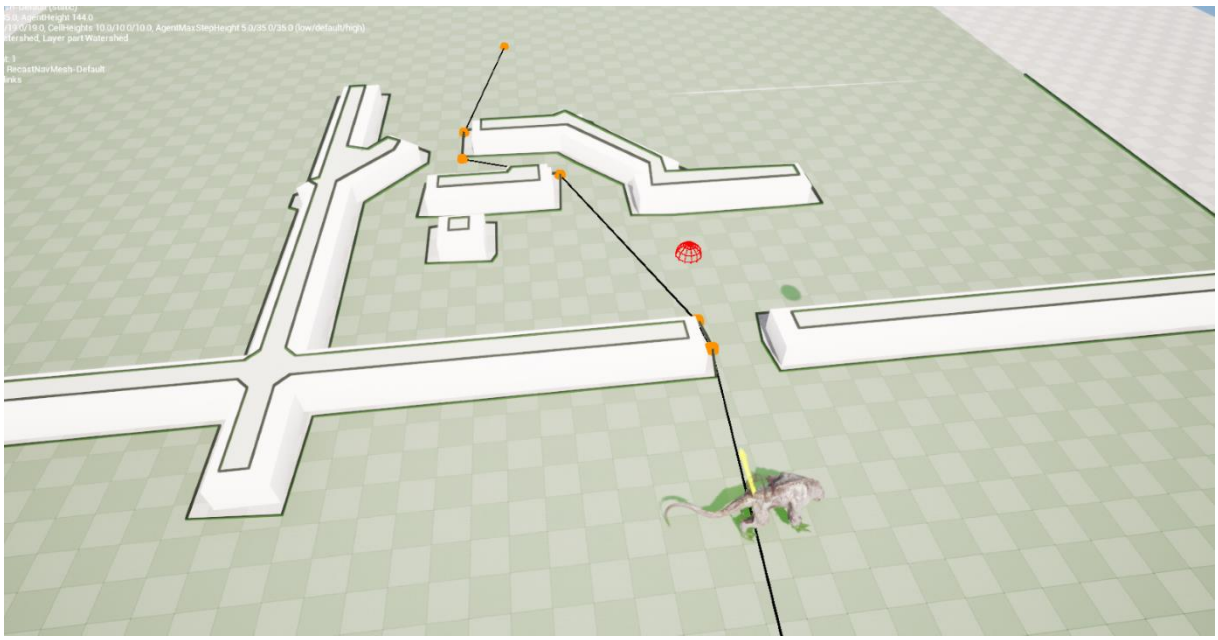
Figura 48 - NavMesh



Fonte: Autoria própria

Com o *NavMesh* configurado, o processo de movimentação do monstro torna-se simplificado. Ao utilizar a função *MoveTo* dentro de uma *Behavior Tree*, o monstro é capaz de seguir o caminho previamente determinado pelo *Path Component* (Figura 49).

Figura 49 - Caminho gerado pelo monstro



Fonte: Autoria própria

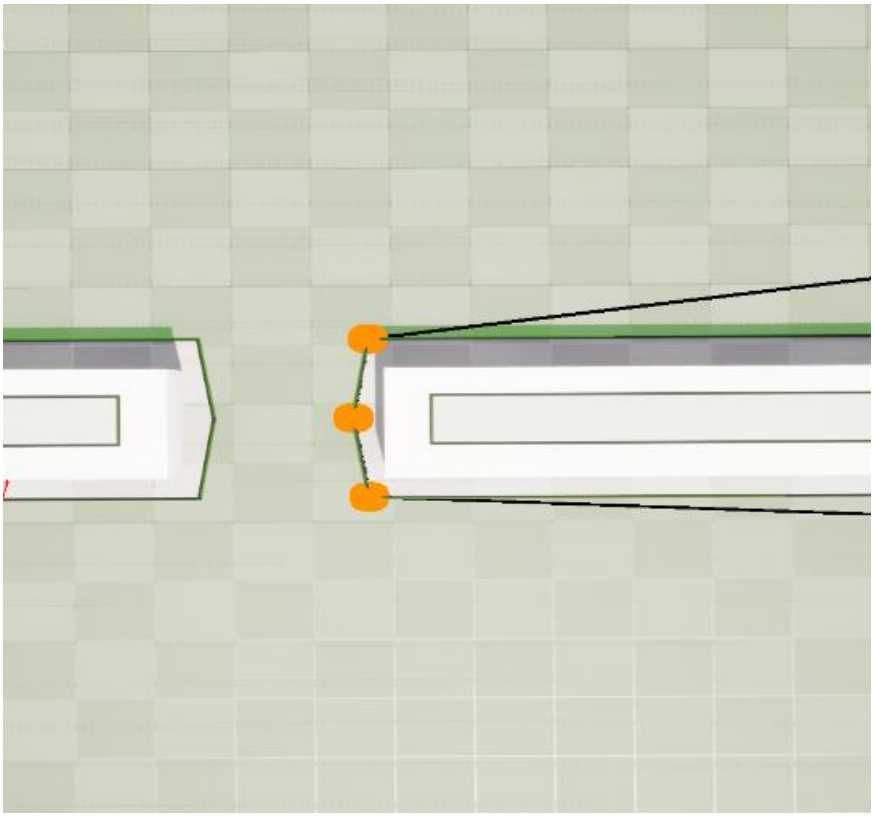
Ao analisar a interação do monstro com o sistema de *PathFinding* implementado na *Unreal Engine*, percebe-se que há desafios relacionados à suavidade e naturalidade de seu movimento, como ilustrado na Figura 49, onde o monstro está andando de lado. Uma abordagem inicial foi tentar alinhar a rotação do monstro com o caminho determinado pelo *PathFinding*. No entanto, essa estratégia resultava em rotações excessivamente rápidas do monstro, especialmente em curvas fechadas, devido à representação do caminho por uma quantidade limitada de pontos, tornando as curvas abruptas (Figura 50).

Uma alternativa foi orientar o monstro na direção do movimento, proporcionando uma transição mais suave entre os pontos do caminho. No entanto, mesmo com essa abordagem, situações de rápida rotação ainda eram observadas, dependendo das configurações de movimentação, aceleração e atrito do monstro com o ambiente.

Esses desafios evidenciam a necessidade de ajustes refinados na integração entre o monstro e o sistema de *PathFinding,* visando um movimento mais orgânico e

realista, especialmente em curvas e transições de terreno. Essas melhorias são cruciais para aprimorar a experiência do jogador e garantir que o comportamento do monstro seja coerente e envolvente dentro do jogo.

Figura 50 - Caminho com curvas abruptas



Fonte: Autoria própria

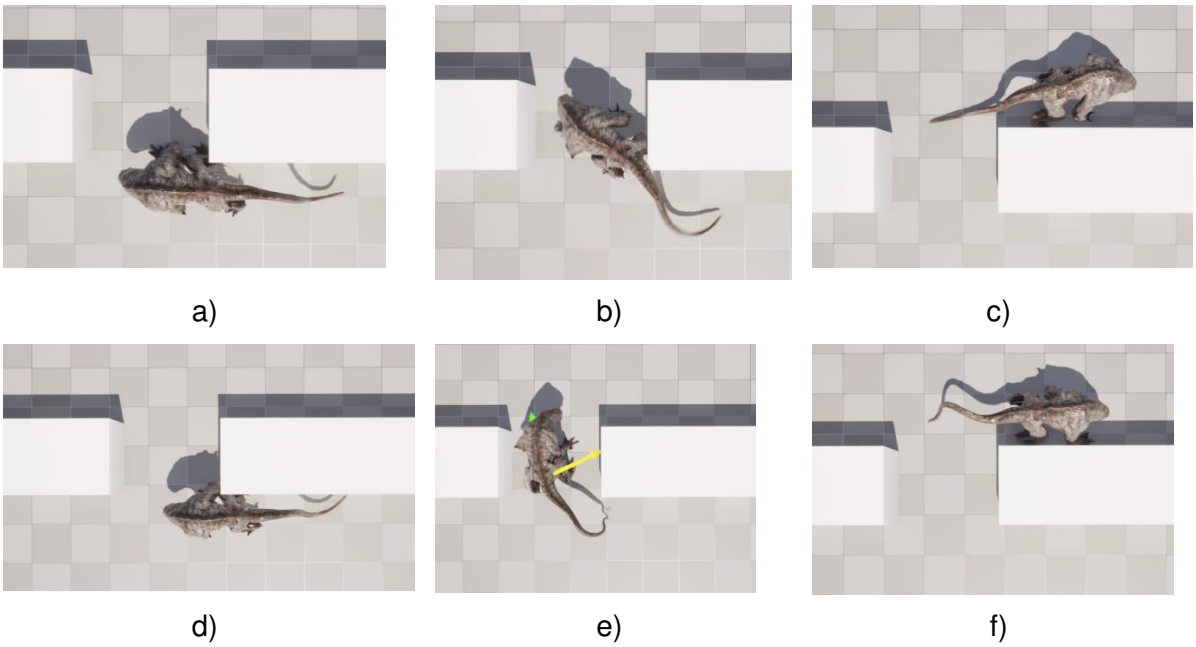
Para abordar o desafio anteriormente mencionado, foi implementada uma estratégia em que o monstro gradualmente rotaciona em direção ao seu movimento. Essa abordagem mitigou a questão da rotação excessiva e começou a conferir ao movimento uma aparência mais natural. Contudo, essa solução revelou um novo problema: o monstro girava com uma velocidade constante em direção à rotação desejada, resultando em uma parada abrupta quando finalmente apontava na direção do movimento. Para superar essa limitação, introduziu-se uma curva de velocidade para a rotação, permitindo que o monstro girasse mais lentamente à medida que se aproximava da rotação desejada. Essa implementação contribuiu significativamente para tornar a movimentação mais fluida e orgânica, já que o monstro ajustava suavemente sua rotação quando se aproximava do ângulo desejado.

Apesar dessas melhorias, persistia um problema quando a velocidade de rotação do monstro não era suficientemente rápida. Em alguns casos, o monstro acabava movendo-se lateralmente por um curto período, devido à natureza procedural da rotação, que levava algum tempo para se alinhar com a direção desejada. Aumentar a velocidade de rotação não era uma solução viável, pois resultaria em animações artificiais. Assim, optou-se por mais uma vez empregar a técnica de animações aditivas. Desta vez, a animação aditiva induz o monstro a curvar totalmente sua coluna (Figura 51). Ajustando a força dessa animação aditiva, foi possível fazer com que o monstro se curvasse consideravelmente quando a rotação desejada estava distante da rotação atual, e menos quando já estava próxima da direção desejada. Esse refinamento proporcionou uma transição mais suave e natural no movimento do monstro, mesmo em situações em que a rotação exigia ajustes rápidos, como pode ser visto na comparação entre as Figuras 52 (a), (b) e (c) que demonstram o movimento do monstro antes de aplicar animações aditivas e as Figuras 52 (d), (e) e (f), depois de aplicadas as animações aditivas.

Figura 51 - Animação aditiva do monstro

Fonte: Autoria própria

Figura 52 - Movimentação do monstro



Fonte: Autoria própria

Como evidenciado na segunda imagem de ambos os casos, a implementação da rotação do monstro em direção ao seu movimento, aliada ao uso de poses aditivas para curvar a coluna, confere à animação uma naturalidade e fidelidade ao movimento real de um animal. Em contrapartida, na ausência dessa técnica, o monstro realiza a rotação (mesmo que suavizada) de maneira linear, acompanhada por considerável deslizamento nos pés.

Para atenuar ainda mais o deslizamento dos pés do monstro durante curvas, uma abordagem eficaz consiste na criação de um *blendspace* entre animações de andar com curva para a esquerda, andar para frente e andar com curva para a direita. Esse *blendspace* possibilita a transição fluida entre animações, de acordo com a intensidade da curva realizada pelo monstro. Consequentemente, à medida que o monstro curva, animações mais adequadas ao movimento são acionadas, resultando em um deslizamento reduzido nos pés e um comportamento mais autêntico durante a movimentação.

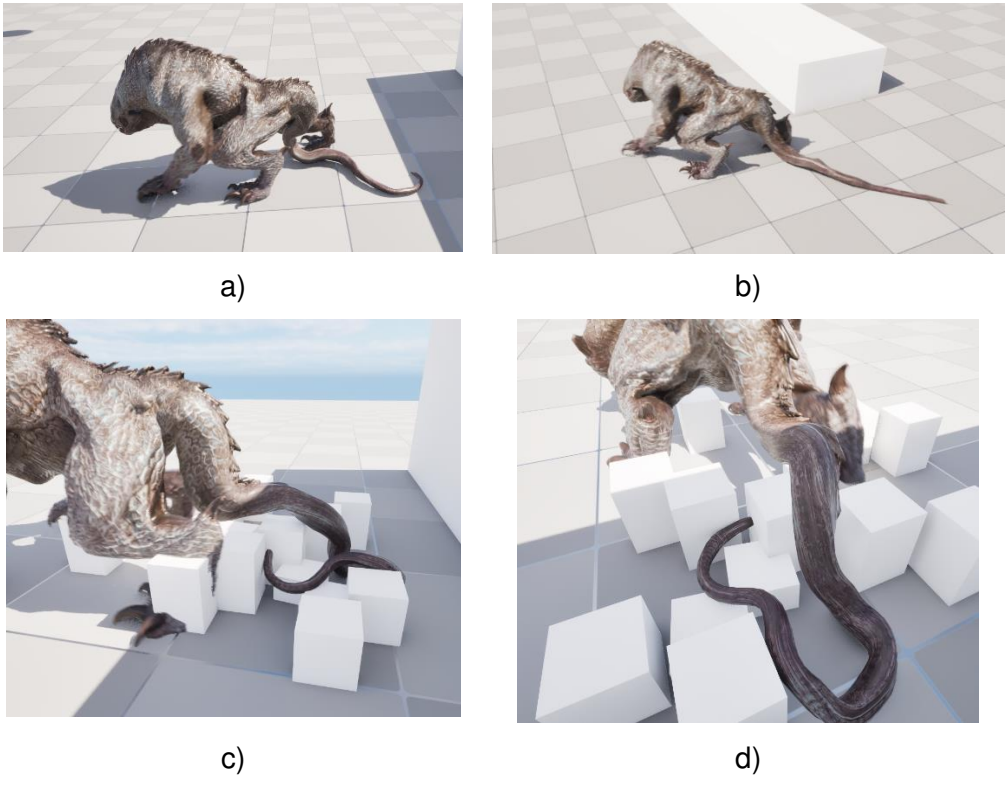
Durante os testes de movimentação com o monstro, uma observação relevante surgiu: a ausência de configuração de colisões na malha do monstro resultava em partes, como a cabeça e o rabo, atravessando elementos do cenário de teste. Na *Unreal Engine*, a gestão de colisões para personagens é frequentemente realizada

por meio de uma cápsula de colisão. Inicialmente, as malhas dos personagens não apresentam colisões e são associadas à cápsula.

Para solucionar a questão da cabeça do monstro atravessando paredes, uma solução direta envolve o aumento do tamanho da cápsula do personagem, envolvendo adequadamente a cabeça do monstro.

No entanto, para o rabo, uma abordagem mais eficaz foi empregada. Em vez de tentar englobá-lo com a cápsula, optou-se por ativar as colisões nos ossos do rabo e simular a física nesses ossos. Essa medida confere ao rabo do monstro uma presença mais substancial no mundo virtual, permitindo colisões mais realistas com objetos e o ambiente circundante, como ilustrado na Figura 53, que demonstram o rabo do monstro interagindo com o ambiente de forma mais autêntica.

Figura 53 - Simulação do rabo



a) b)

c) d)

Fonte: Autoria própria

A incorporação das colisões nos ossos do rabo proporcionou uma interação mais satisfatória com o ambiente. Contudo, essa implementação gerou uma percepção de que o rabo do monstro carecia de musculatura ou força própria, o que, anatomicamente, não fazia sentido para a criatura. Para resolver essa questão sem

sacrificar a interação realista do rabo com o ambiente, recorreu-se à técnica de animações físicas, conhecida como Physical Animation.

A *Physical Animation* permite manter as animações originais do monstro enquanto proporciona a interação do rabo com o ambiente circundante. Essa abordagem resultou em um monstro que não apenas se movimentava de maneira realista, mas também interagia de forma orgânica com o terreno conferindo ao monstro uma presença mais impactante no cenário, como evidenciado na Figura 54.

Figura 54 - Simulação física

a) b)

Fonte: Autoria própria

Com as animações do monstro concluídas e integradas ao algoritmo de *pathfinding*, resta agora implementar a programação para a percepção do jogador e a lógica de morte do monstro. Para a percepção do jogador, uma solução sugerida é a utilização do componente *AI Perception*, que oferece um sistema de detecção por cones de visão. A configuração do tamanho do cone e os tipos de atores a serem detectados são ajustáveis, iniciando o processo de perseguição sempre que o cone detecta um jogador.

A robustez do sistema *AI Perception* possibilita uma detecção eficiente do jogador, permitindo a ativação de comportamentos específicos, como a transição para o estado de perseguição, quando o monstro identifica a presença do jogador em seu campo de visão.

Quanto ao sistema de vida do monstro, a implementação em C++ envolve a criação de uma variável para determinar a quantidade de vida do monstro, juntamente com uma função que registra o dano causado pelos tiros do jogador. Esta abordagem não apenas gerencia a integridade do monstro, mas também permite a customização

de eventos, como a reprodução de uma animação de morte quando a vida do monstro é reduzida abaixo de zero.

Ao desconectar o *AIController*, evita-se que o monstro continue a receber inputs, garantindo uma resposta visual e lógica à derrota do monstro no jogo.

3.5 MAPA

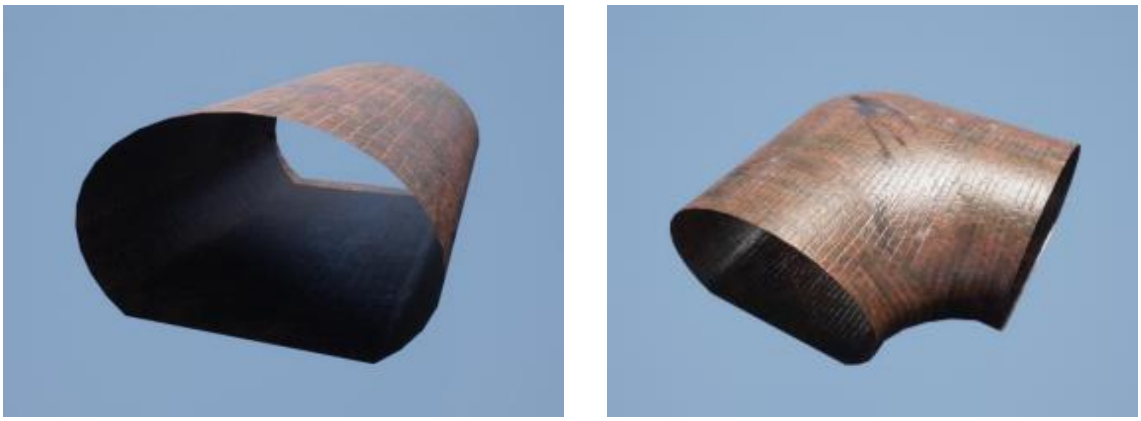
Como etapa final do projeto, há a necessidade de criação de um ambiente orgânico e divertido, onde o jogador possa explorar, atacar e se esconder do monstro. Com isso em mente, os próximos capítulos têm como objetivo explorar técnicas e algoritmos para a criação de um mapa aleatório, mas que permite um certo controle na criação pelo programador.

3.5.1 Geração procedural

Para desenvolver um mapa procedural, optou-se por uma abordagem baseada na criação de módulos modulares, que podem ser combinados para formar um labirinto, assemelhando-se a um quebra-cabeça Lego. Cada módulo é composto por uma malha e informações sobre suas saídas.

A lógica por trás dessa estratégia é assegurar que as saídas especificadas sejam coesas com a geometria da malha do módulo. Por exemplo, a Figura 55 (a) ilustra uma malha adequada para um módulo com uma entrada e uma saída em lados opostos, enquanto a Figura 55 (b) mostra uma malha para entrada e saídas perpendiculares. Este enfoque proporciona flexibilidade na construção do mapa, permitindo a combinação de módulos de maneira coesa e gerando uma variedade de ambientes de jogo de forma procedural.

Figura 55 - Módulos

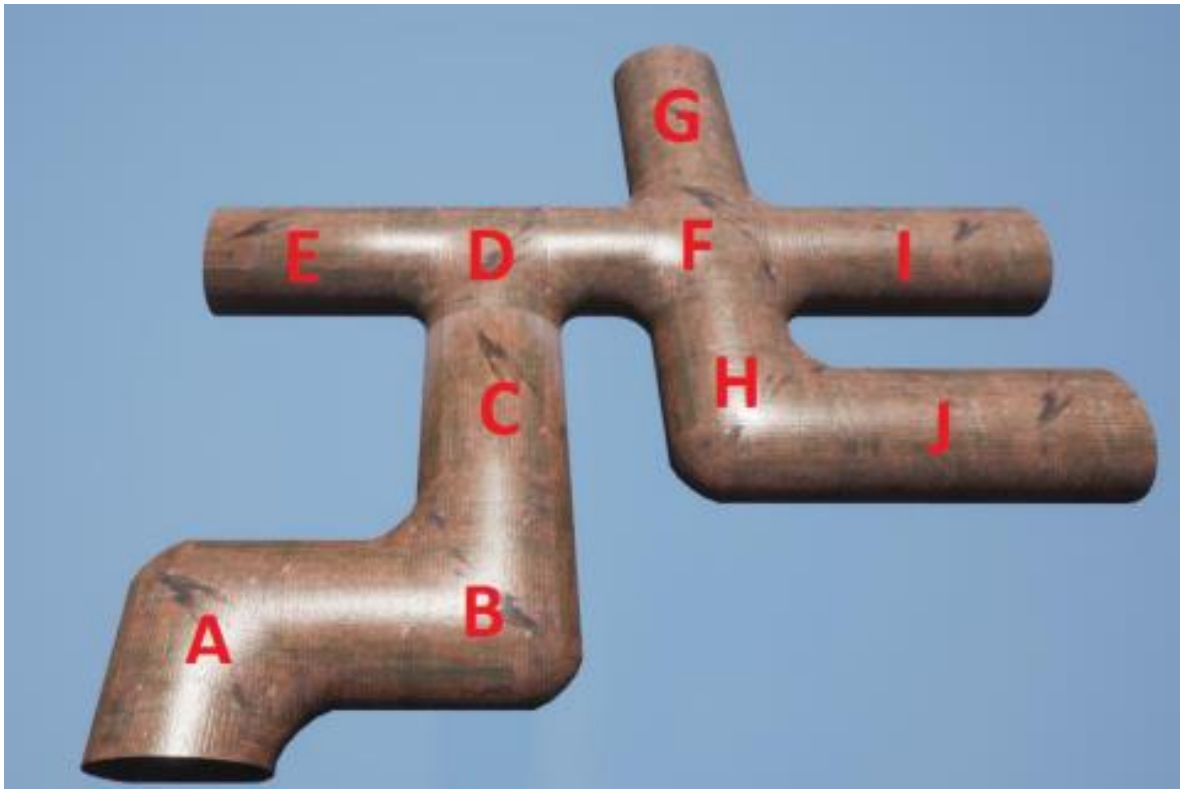


a) b)

Fonte: Autoria própria

Com um labirinto simplificado representado por uma matriz, onde é possível mover-se para cima, para baixo e para os lados, um conjunto de módulos é capaz de criar qualquer variação de labirintos. A montagem ocorre conectando os módulos conforme suas saídas, fazendo rotações quando necessário. Um exemplo de labirinto montado seria pode ser visualizado na Figura 56.

Figura 56 - Mapa modular



Fonte: Autoria própria

Com base na representação do labirinto, composto por 10 módulos, cada um pertencendo a uma das 5 classes possíveis, é evidente que, com um total de 5 classes, é possível criar toda a estrutura do labirinto. As classes são as seguintes:

1. Linha reta
2. Curva
3. Beco
4. Cruzamento em T
5. Cruzamento em X

Segue a legenda dos módulos utilizados na Figura 56 de acordo com os tipos de módulos existentes:

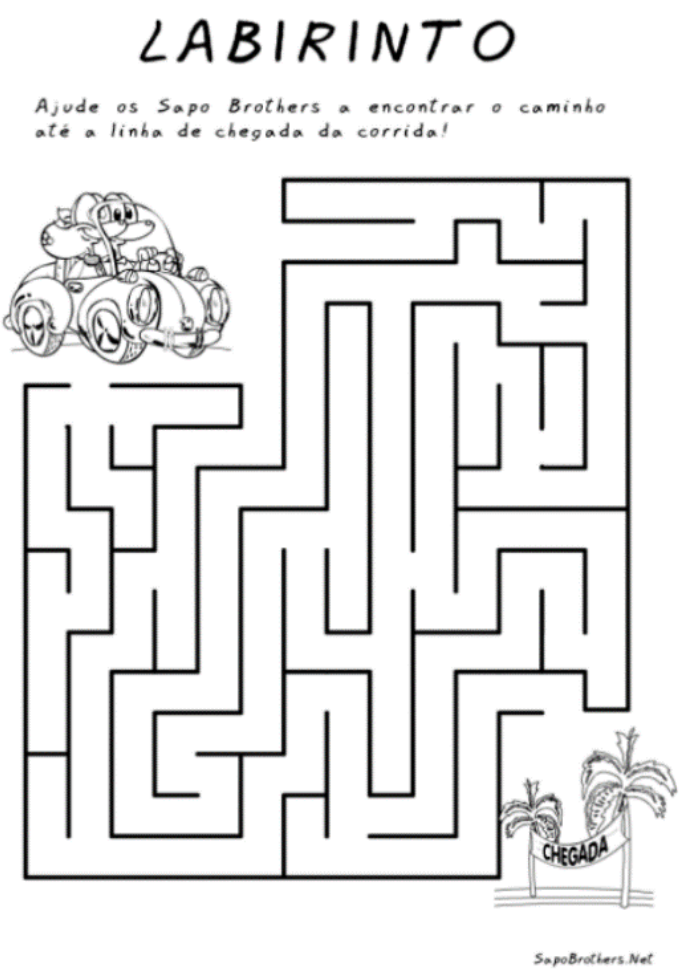
A. Curva
B. Curva (rotacionada)
C. Linha reta
D. Cruzamento em T
E. Beco
F. Cruzamento em X
G. Beco
H. Curva (rotacionada)
I. Beco
J. Beco

Demonstrando que é viável criar um labirinto utilizando esses módulos, agora vamos explorar a lógica para a montagem do labirinto.

Com certeza, a experiência de jogabilidade é fundamental, e labirintos excessivamente confusos podem frustrar os jogadores. Evitar estruturas muito intricadas é uma prática sensata, pois os jogadores geralmente preferem desafios que sejam envolventes e estimulantes, mas não excessivamente complexos. Neste sentido, é importante criar labirintos que proporcionem uma experiência divertida e desafiadora, sem comprometer o aproveitamento do jogo.

Um exemplo de labirinto que deve ser evitado pode ser evidenciado na Figura 57, que consiste em um labirinto extremamente complexo com vários caminhos que levam à becos sem saída; extremamente frustrante para um jogador, especialmente um jogador que não vê o labirinto por cima e, por causa do limite da velocidade de locomoção, está limitado à explorar o labirinto em alta velocidade.

Figura 57 - Labirinto 2D



Fonte: Retirado de banco de imagens

Em primeiro lugar, é essencial destacar que o labirinto apresentado possui um início e um fim, algo que não se alinha à proposta do meu jogo. No meu jogo, toda a ambientação é um labirinto em si, sem um ponto de partida ou destino específico. O jogador começa em algum lugar, mas não é obrigado a alcançar um destino para vencer. A jornada é livre, permitindo que o jogador explore o labirinto enquanto cumpre seus objetivos.

Outro aspecto crucial a ser considerado é a saturação da matriz completa no puzzle apresentado. Isso implica em excesso de opções de caminhos, muitos dos quais podem levar a becos sem saída. Essa abordagem pode se tornar frustrante, especialmente considerando o tempo de movimentação do personagem no jogo. Ao contrário de uma folha de papel, onde é possível voltar rapidamente ao ponto de desvio, o jogo exige que o jogador percorra todo o caminho de volta, aumentando a frustração.

Além disso, o desafio apresentado pelo *puzzle* em questão oferece uma diversão mínima ao jogador que o resolve. Vale ressaltar que o meu jogo não é exclusivamente um jogo de labirinto; o labirinto é apenas um elemento utilizado para proporcionar mais diversão. O labirinto mencionado poderia ser gerado por uma simples função DFS, percorrendo aleatoriamente a matriz, mas essa abordagem não se adequa ao meu jogo e não é uma função facilmente modificável para atender às suas necessidades.

Assim, a ideia é criar um labirinto simples e envolvente, minimizando a confusão para os jogadores. O objetivo é proporcionar situações em que o jogador pode se perder ocasionalmente, mas sem que isso ocorra com frequência.

3.5.1.1 Algoritmo

A primeira consideração a ser feita é que o labirinto será montado em formato de matriz, uma questão de simplicidade e facilidade na implementação, utilizando estruturas nativas da linguagem de programação para representar um labirinto. Outro detalhe importante a se considerar é que a matriz do labirinto não deve ser totalmente preenchida. Essa decisão visa evitar que o jogador fique constantemente se deparando com becos sem saída, o que poderia causar cansaço e frustração ao tentar navegar pelo jogo.

3.5.1.1.1 Pontos de interesse

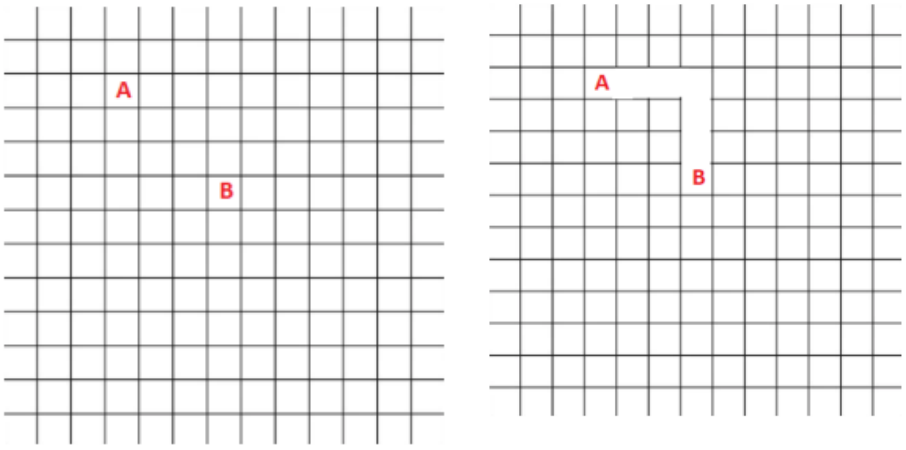
Outro detalhe importante é que o labirinto deve ser composto por salas. Essas salas são então interligadas por corredores que juntas formam um labirinto. Essas salas serão denominadas Pontos de Interesse. Os pontos de interesse podem consistir em mais de um módulo, podendo formar grandes salas que ocupam mais de uma posição na matriz do labirinto. Portanto, as informações importantes que devem ser guardadas nos pontos de interesse são, além de quantos e quais as posições que ocupam na matriz, onde estão as posições de saída daquele ponto de interesse (onde o jogador pode entrar naquela sala ou sair dela).

3.5.1.1.2 Conectando 2 pontos de interesse

Explicado o que é um ponto de interesse, é preciso definir como duas salas serão interligadas. Para isso, utiliza-se uma matriz e a perspectiva do labirinto a partir de uma visão superior. A representação da matriz também assume a forma de um grafo, onde cada célula é tratada como um nó. Para cada nó (excluindo os da borda), há quatro arestas conectando-o: uma que conduz ao nó imediatamente acima, outra ao nó imediatamente à direita, uma terceira ao nó imediatamente à esquerda e, por fim, uma que se dirige ao nó imediatamente abaixo.

Uma abordagem eficiente para conectar os pontos de interesse é implementar um algoritmo de *Dijkstra*, onde cada aresta possui peso 1. Ao aplicar esse algoritmo, obtemos uma conexão direta entre os pontos na matriz, como ilustrado na Figura 58 após a execução.

Figura 58 - Conexão de pontos de interesse

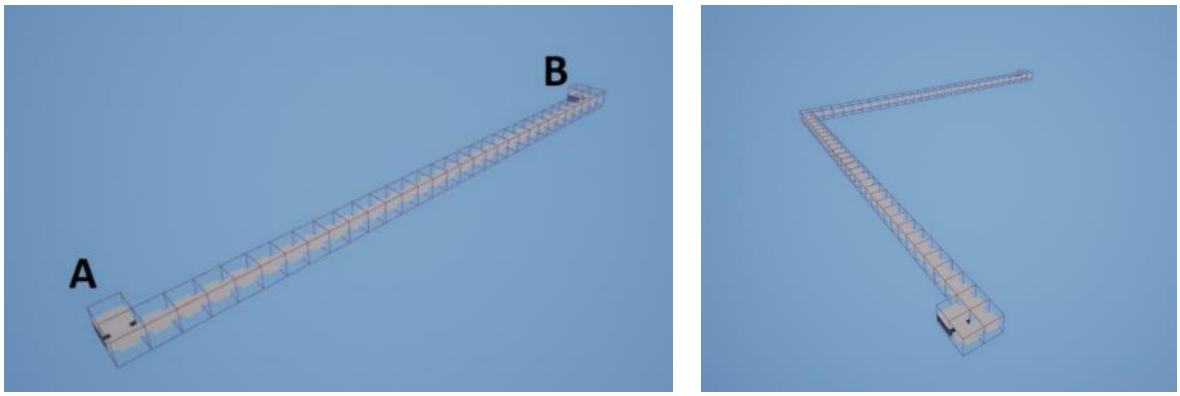


a) b)

Fonte: Autoria própria

O algoritmo de *Dijkstra* aplicado ao grafo, embora eficiente, apresenta um problema significativo: os caminhos resultantes são diretos, revelando padrões previsíveis. Isso se traduz em trajetórias com distâncias de Manhattan, sem contornos ou curvas inesperadas. Tal previsibilidade pode comprometer a experiência do jogador, tornando os caminhos entre pontos demasiadamente óbvios como evidenciado na Figura 59, que foram resultados da implementação de tal algoritmo dentro da *Unreal Engine*.

Figura 59 - Conexão de pontos de interesse na Unreal

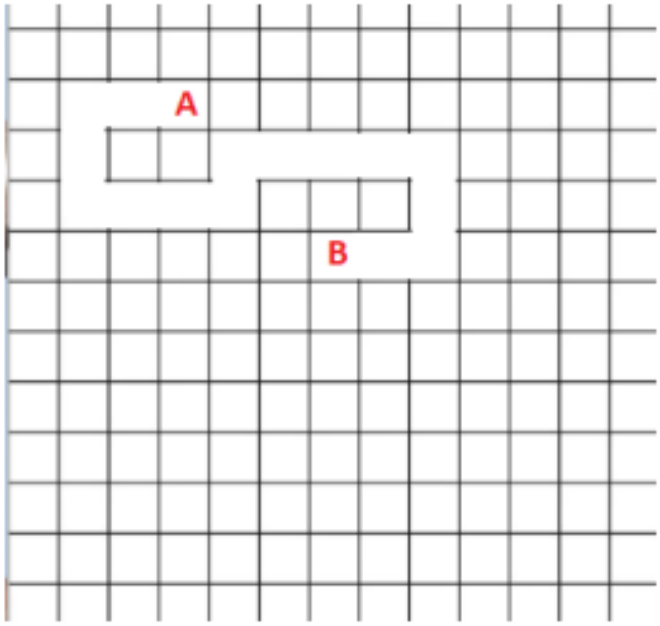


a) b)

Fonte: Autoria própria

Um exemplo de caminho não previsível, desejado para proporcionar uma experiência mais desafiadora ao jogador, é ilustrado na Figura 60. Nesse caso, a trajetória não segue diretamente em direção ao ponto final 'B', exigindo que o jogador se afaste inicialmente de 'B' em termos de distância euclidiana antes de se aproximar. Para obter esse tipo de resultado, é necessário aplicar o algoritmo de *Dijkstra* e atribuir pesos aleatórios aos vértices.

Figura 60 - Caminho desejado

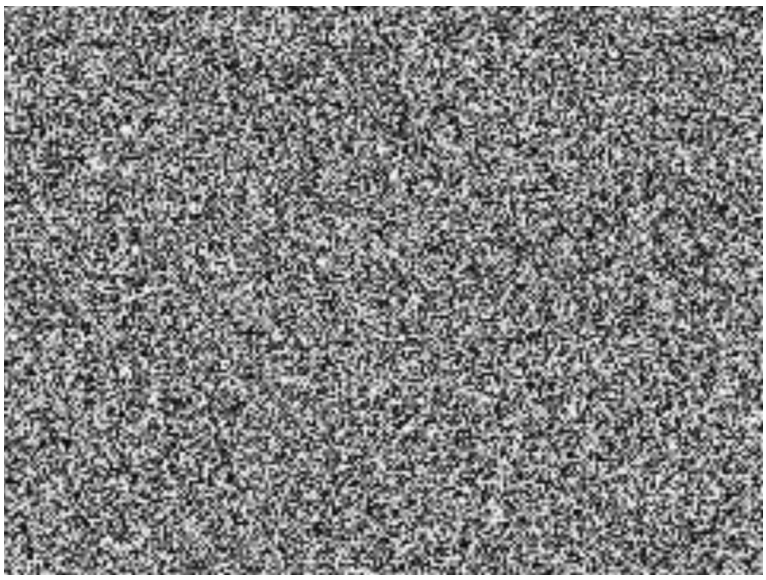


Fonte: Autoria própria

Na busca por uma solução, o autor inicialmente optou por atribuir pesos completamente aleatórios a todos os nós (Figura 61), resultando em um padrão inesperado. Ao utilizar pesos totalmente aleatórios, observou-se que os caminhos

gerados eram praticamente idênticos aos produzidos pela distância de Manhattan. Isso provavelmente ocorre devido à probabilidade equitativa de todos os pesos no mapa. Durante a busca por um caminho, qualquer movimento que se afaste do objetivo reduz a probabilidade de encontrar um caminho mínimo, levando a escolhas frequentes em direção ao destino.

Figura 61 - Noise randômico

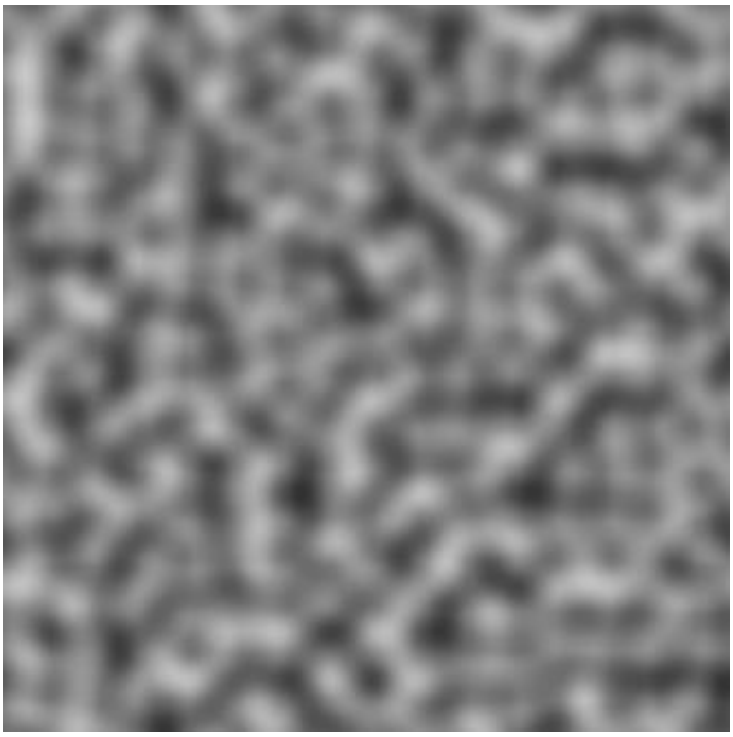


Fonte: WIKIPEDIA

Uma alternativa sugerida é a aplicação de valores randômicos através do *Perlin Noise* (Figura 62). O *Perlin Noise* é uma função que produz padrões de ruído pseudoaleatórios, comumente utilizada para criar efeitos visuais naturais e orgânicos em gráficos computacionais.

Essa abordagem permite a existência de extensas regiões na matriz com valores de peso significativamente elevados, induzindo o algoritmo de *Dijkstra* a evitar essas áreas, resultando em trajetos mais longos em busca de caminhos mais eficientes.

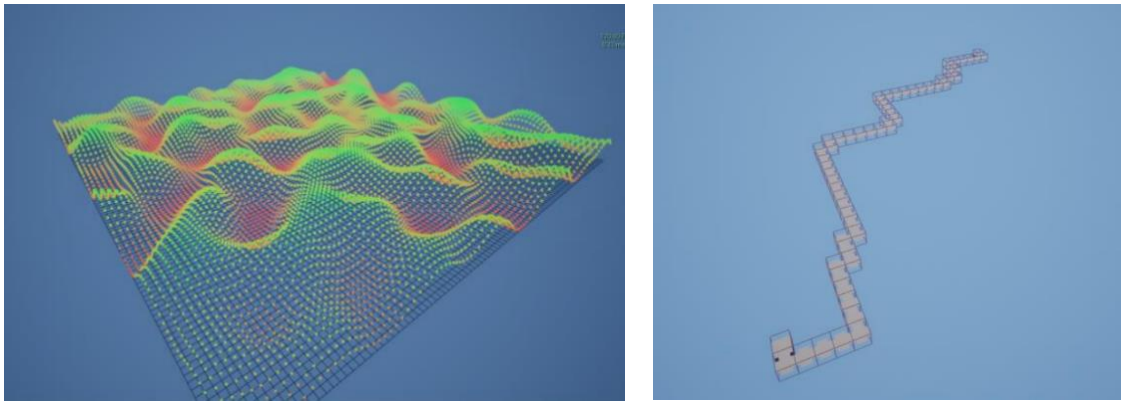
Figura 62 - Perlin Noise



Fonte: Wikipedia

Na Figura 63 (a), apresenta-se o primeiro resultado da visualização da malha *Perlin*, onde a altura de cada célula indica o peso para alcançá-la. A adição de cores visa facilitar a compreensão visual, sendo que áreas verdes representam caminhos com custo mais baixo, enquanto áreas vermelhas indicam custos mais elevados. Já na Figura 63 (b), é apresentado um resultado de um labirinto utilizando o *Perlin Noise* como peso. Nessa representação, a influência do *Perlin Noise* nos pesos das células contribui para a criação de caminhos menos previsíveis, proporcionando uma experiência mais desafiadora e interessante para o jogador.

Figura 63 - Caminho gerado com perlin noise



a) b)

Fonte: Autoria própria

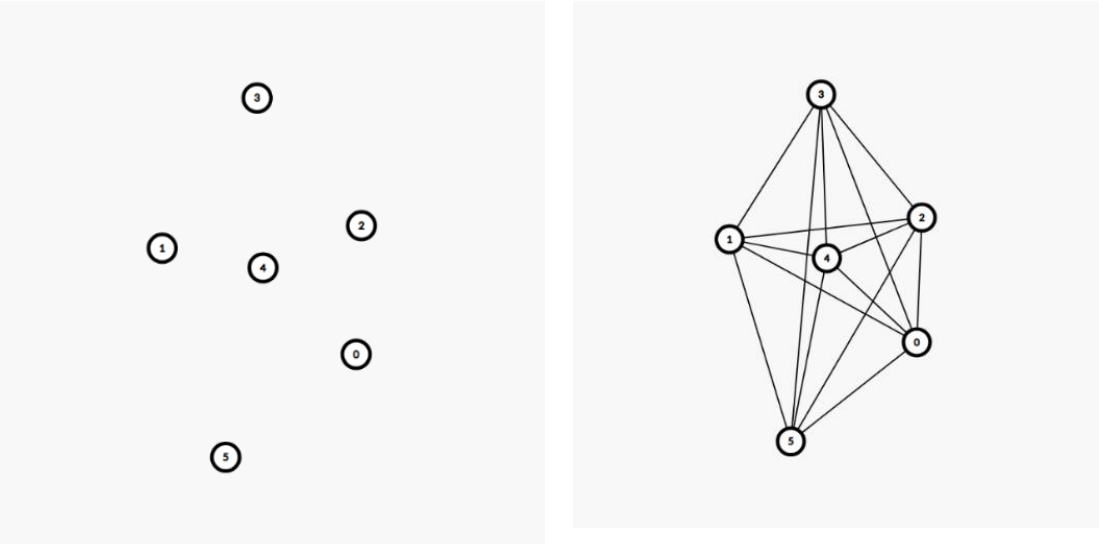
Outro aspecto a ser enfatizado é que o algoritmo desenvolvido, além de fornecer e possibilitar a criação do caminho encontrado, também apresenta o valor do caminho mínimo entre os dois pontos de interesse fornecidos, uma informação que será útil no próximo subcapítulo.

3.5.1.1.3 Conectando todos os pontos de interesse

Com o problema de conectar os pontos de interesse resolvido, ainda não temos um labirinto verdadeiro. Até agora, apenas estabelecemos uma maneira de conectar dois caminhos. Não há confusão para o jogador, nem ramificações no mapa, nada além de um único caminho entre dois pontos.

Uma solução simples para conectar mais de dois pontos de interesse é pensar nos pontos como nós em um grafo. Basta criar caminhos correspondendo às arestas de um grafo completo, como pode ser visto na Figura 64 (a), que mostra o grafo sem nenhuma conexão, e na Figura 64 (b), que mostra o grafo completamente conectado.

Figura 64 - Conexão de pontos de interesse em grafo



a) b)

Fonte: Autoria própria

O mapa pode se tornar uma verdadeira bagunça, com vários caminhos se sobrepondo uns aos outros, resultando em um grafo cheio de ciclos e múltiplas formas de chegar ao mesmo lugar. Essa complexidade pode se tornar cansativa para os jogadores, tornando essencial o controle da complexidade do mapa.

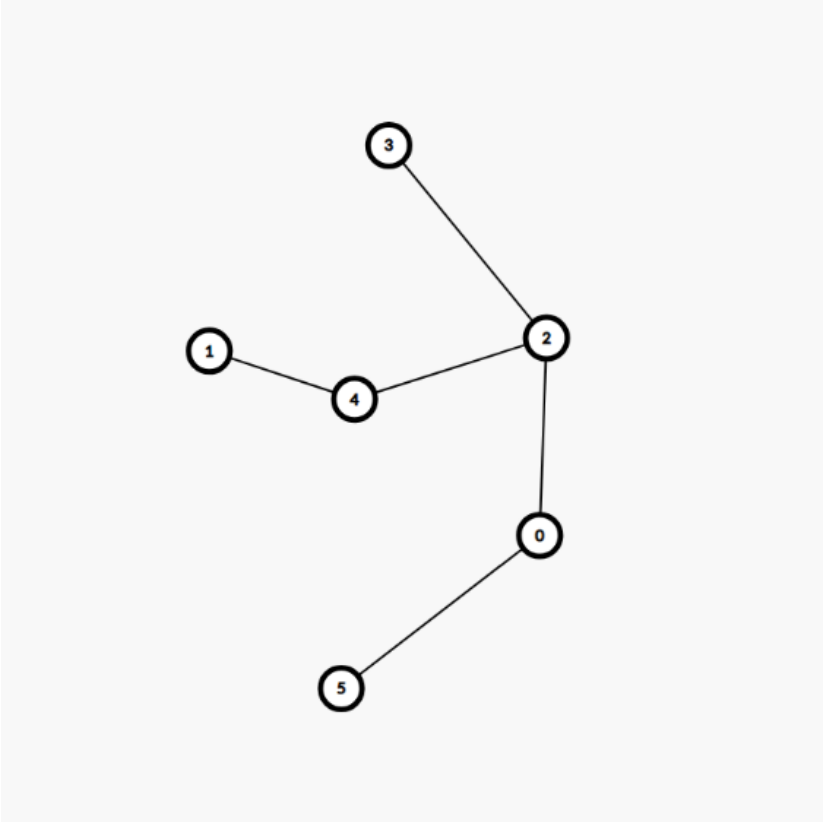
Um conceito fundamental para alcançar essa organização é a utilização de uma árvore geradora, garantindo que pelo menos exista um caminho direto ou indireto para cada ponto de interesse. Em teoria dos grafos, uma árvore geradora é um subgrafo que contém todos os vértices do grafo original e forma uma árvore, garantindo conectividade entre os pontos. Essa árvore pode ser minimizada para criar uma árvore geradora mínima, que, em termos simples, é a árvore que conecta todos os pontos com o menor custo total possível.

O algoritmo de *Kruskal* é uma escolha adequada para encontrar a árvore geradora mínima, especialmente porque o autor possui familiaridade com esse algoritmo devido às maratonas de programação. O *Kruskal* opera de maneira eficiente, selecionando arestas em ordem crescente de peso e adicionando-as à árvore geradora mínima, garantindo uma conectividade mínima entre os pontos de interesse.

O processo começa com a criação de um grafo completo, onde todas as arestas são ponderadas usando o algoritmo de *Dijkstra* previamente implementado. Os pesos das arestas são calculados com base no custo retornado pelo algoritmo de *Dijkstra*,

com foco apenas no custo. Em seguida, o algoritmo de *Kruskal* é aplicado para determinar quais pontos de interesse devem ser conectados, formando assim a árvore geradora mínima desejada. Esse método não apenas evita a sobreposição de caminhos, mas também prioriza a conexão de nós próximos, contribuindo para a organização e eficácia do mapa.

Figura 65 - Árvore geradora mínima



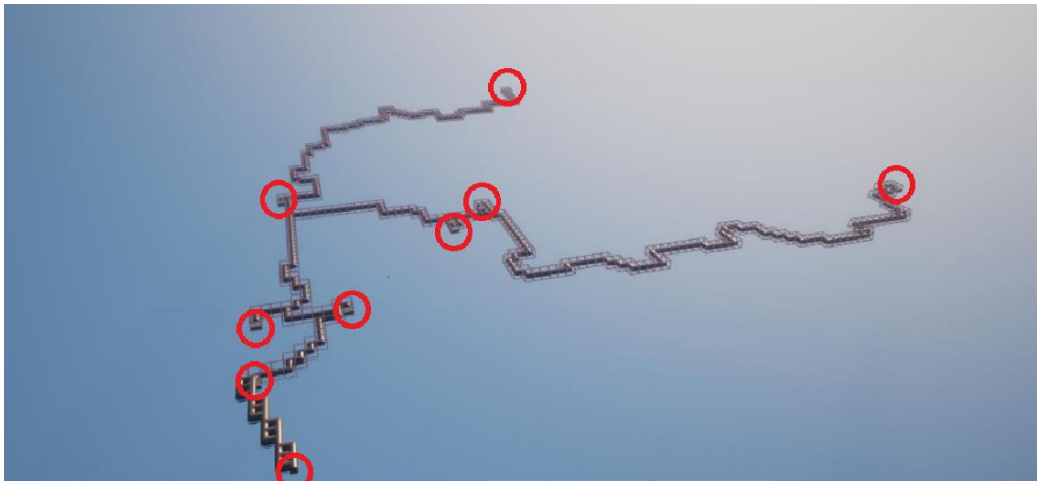
Fonte: Autoria própria

Novamente, o algoritmo de *Dijkstra*, previamente implementado, desempenha um papel crucial na etapa final do processo, em que os caminhos são efetivamente gerados e posicionados no ambiente do jogo. Esse passo resulta na criação de caminhos orgânicos, repletos de curvas, mas ainda assim mantendo uma certa simplicidade para facilitar a orientação dos jogadores.

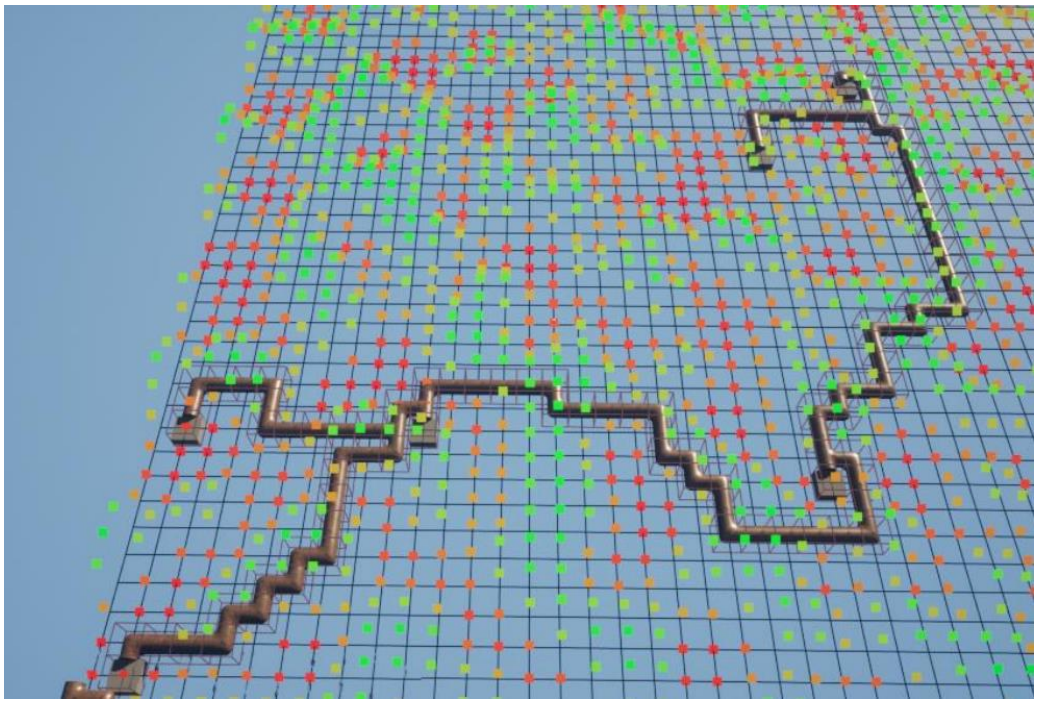
Ao utilizar o algoritmo de *Dijkstra*, os caminhos são estrategicamente posicionados no mundo, proporcionando um ambiente expansivo, orgânico e, ao mesmo tempo, evitando a complexidade e a confusão excessiva. A Figura 66 (a) ilustra como, mesmo com uma grande quantidade de pontos de interesse (marcados para evidência), os mapas podem ser extensos, orgânicos e desafiadores, sem perder a clareza. A Figura 66 (b) complementa essa visão ao apresentar outro mapa gerado

com o algoritmo proposto, destacando simultaneamente a malha de *Perlin Noise* utilizada para orientar a geração desse mapa. Também é evidente que os caminhos criados pelo algoritmo tendem a percorrer áreas do mapa onde o custo é mais baixo.

Figura 66 - Mapa com vários pontos de interesse na Unreal



a)



b)

Fonte: Autoria própria

Duas considerações finais merecem destaque: Primeiramente, a estrutura montada garante que os mapas gerados nunca apresentem ciclos, um efeito que pode ser intencional ou não. Dado que o algoritmo de *Kruskal* fornece as arestas utilizadas

na Árvore Geradora Mínima (*Minimum Spanning Tree* - MST), a inserção de qualquer outra aresta não utilizada resultará na criação de ciclos. Portanto, se a intenção for incluir ciclos, basta selecionar algumas arestas não presentes na MST e adicionar seus caminhos ao mapa.

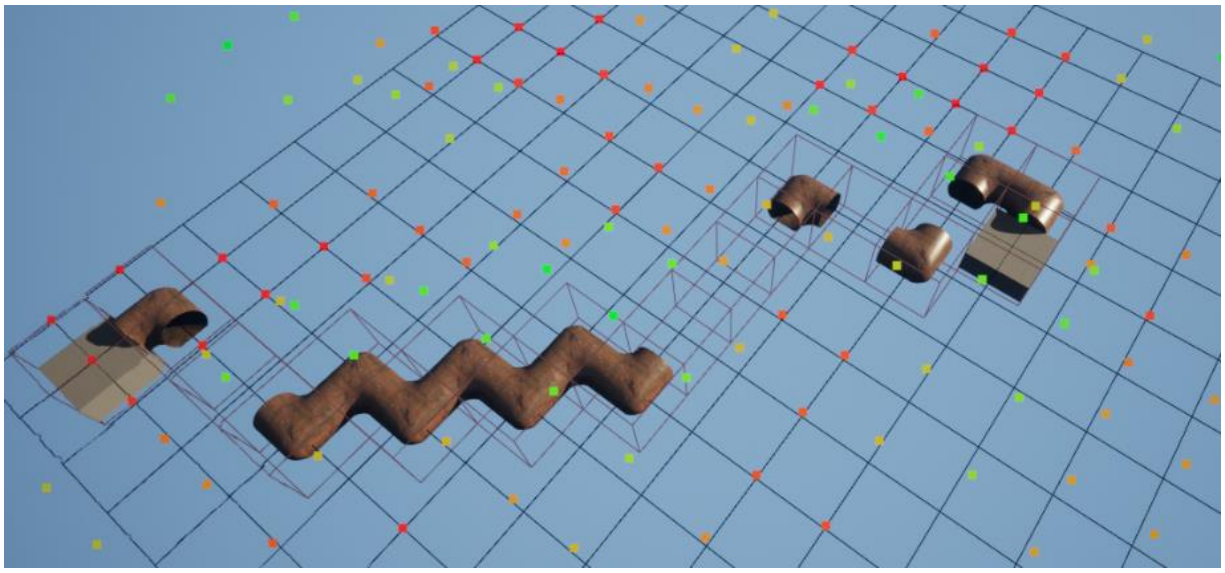
A segunda consideração é que, dado o modelo atual, todos os caminhos criados pelas arestas ligam um ponto de interesse a outro, eliminando a possibilidade de existirem caminhos que levem a becos sem saída. Para incorporar becos sem saída ao labirinto, basta incluí-los como pontos de interesse antes da geração da MST pelo algoritmo de *Kruskal*. Dessa forma, os becos sem saída tornam-se pontos de interesse e pelo menos um caminho (os caminhos originais da MST) conduzirá a esses becos sem saída.

A escolha de quais arestas adicionar como ciclos (mais caras ou mais leves) e a decisão de quantos becos sem saída adicionar, bem como onde posicioná-los, são configurações a serem escolhidas por meio de experimentação, considerando os resultados desejados para o labirinto. Para labirintos mais complexos, sugere-se a inclusão de vários becos sem saída e a adição de arestas-ciclo mais caras, proporcionando caminhos desafiadores e aumentando a chance de sobreposição com outros caminhos já existentes no mapa.

3.5.2 Algoritmo de escolha de módulos

Além da geração do mapa, é imperativo desenvolver um algoritmo que, dado os caminhos selecionados pelo algoritmo de *Kruskal* e pelas modificações do algoritmo de *Dijkstra*, assegure que os caminhos no mapa estejam alinhados com a quantidade de interseções naquela célula da matriz. Além disso, a rotação desses caminhos deve ser ajustada de acordo com as entradas e saídas da célula específica. Essa abordagem é essencial para evitar problemas de rotação, garantindo que uma mesma malha seja rotacionada de acordo com a necessidade, como ilustrado na Figura 67, onde os caminhos, embora com a mesma malha, apresentam rotações diferentes para adequar ao caminho escolhido. Essa correção garantirá uma representação precisa da rede de caminhos no mapa.

Figura 67 - Módulos em caminho



Fonte: Autoria própria

3.5.3 Melhoria visual

Agora que os algoritmos para a construção dos mapas estão finalmente prontos, a ferramenta revela um potencial significativo, permitindo a criação de mapas extremamente orgânicos com visuais diversificados. Isso ocorre porque a malha dos módulos precisa se adaptar às informações específicas de cada tipo de módulo, garantindo entradas e saídas condizentes com as informações fornecidas. Um exemplo do potencial dessa ferramenta pode ser observado na elaboração de um mapa utilizando *assets* do Quixel, modelos 3D de alta qualidade disponíveis no Unreal Marketplace, um espaço onde criadores podem comprar, vender e baixar ferramentas, modelos e outros recursos para auxiliar no desenvolvimento. A Figura 68 ilustra a criação de um mapa de exemplo para o jogo, utilizando módulos e malhas de alta qualidade:

Figura 68 - Melhoria visual de módulos

a)

b)

c)

Fonte: Autoria própria

Quando os módulos estão separados, eles parecem isolados e artificiais. No entanto, ao unir os módulos de acordo com as especificações do algoritmo desenvolvido, torna-se possível criar cavernas procedurais e totalmente aleatórias, como exemplificado na Figura 69 (a) (módulo 'T' conectado a módulos do tipo 'L') e na Figura 69 (b) (módulo "corredor" conectado ao módulo do tipo 'T').

Figura 69 - Mapa modular melhorado

a)

b)

Fonte: Autoria própria

4 CONCLUSÕES

Na conclusão desta monografia, evidencia-se os esforços dedicados à exploração e implementação de técnicas e algoritmos fundamentais para a criação de jogos realistas e imersivos. O escopo abrangente deste trabalho abordou aspectos cruciais, desde a configuração detalhada de armas e animações até a programação sofisticada do comportamento do monstro e a geração procedural de mapas orgânicos.

O cerne desta pesquisa reside na compreensão aprofundada das nuances que contribuem para o realismo em jogos. A análise e implementação do *NavMesh*, aliadas à simulação física do rabo do monstro, revelam não apenas uma busca por realismo, mas também uma atenção meticulosa aos detalhes que potencializam a imersão do jogador.

A aplicação estratégica de algoritmos, como o *Dijkstra* e o *Kruskal* para a criação de labirintos e o *Perlin Noise* para a geração de mapas, destaca a sofisticação técnica do projeto. A decisão de adotar a árvore geradora mínima, proporcionando uma experiência de jogo mais controlada e envolvente, revela a preocupação com a qualidade da jogabilidade.

Ao finalizar este trabalho, é reforçado que a integração harmoniosa de todos esses elementos técnicos e conceituais buscam não apenas oferecer um repertório sólido para o desenvolvimento de jogos realistas, mas também refletem a busca

incessante por inovação e qualidade na criação de experiências memoráveis para os jogadores. Este estudo não é apenas um registro técnico, mas um testemunho do compromisso com a excelência na criação de jogos imersivos e impactantes.

Para futuros desenvolvimentos relacionados à esta monografia, considera-se a implementação de um modo multiplayer como uma extensão promissora. Nessa proposta, múltiplos jogadores teriam a oportunidade de colaborar na batalha contra o monstro, formando estratégias coletivas de defesa e ataque para alcançar o objetivo da missão. A introdução de um modo cooperativo (COOP) pode enriquecer significativamente a experiência do jogo, seguindo a tendência popular observada em títulos recentes de jogos de terror, como Phasmophobia e Lethal Company. A análise e incorporação de elementos de sucesso desses jogos podem contribuir para aprimorar a jogabilidade e atrair um público mais amplo, levando a um enriquecimento da proposta inicial da monografia.

5 REFERÊNCIAS

ABRAGAMES, 2023: Disponivel em: https://www.abragames.org/uploads/5/6/8/0/56805537/2023_fact_sheet_v1.6.2_-_ptbr_version.pdf

BLAIN, J.M. **The Complete Guide to Blender Graphics: Computer Modeling & Animation**, Fifth Edition. A K Peters/CRC Press. 2021

CECOALIENSA. Lizard Pinky Creature Model. 2017. Disponível em: https://www.turbosquid.com/3d-models/rigging-animation-model-1157701. Acessado em: 9 de dezembro de 2023

CHOJNOWSKI, M. Forest Environment - Unreal Engine 5. 2022. Disponível em: https://vladimir_pl.artstation.com/projects/JeVk3D. Acessado em: 9 de dezembro de 2023

DEGOND, P. & Diez, A. & Na, M. **Bulk topological states in a new collective dynamics model**. 2021.

FOWLER, Sean. New! Substance Painter All Levels Volume 1. 2020. Disponível em: https://www.udemy.com/course/learning-substance-painter-all-levels-volume-1-sci-fi-theme/. Acessado em: 9 de dezembro de 2023

JAEGERZ999. The Gun That Won The West (But Modernized). 2023. Disponível em: www.youtube.com/watch?v=REBvNOVSD3k. Acessado em: 9 de dezembro de 2023

JANTUNEN, J. Creating Procedural Textures for Games: With Substance Designer. 2017. Disponível em https://urn.fi/URN:NBN:fi:amk-2017062013867. Acessado em: 9 de dezembro de 2023

KUMAR, A. **Beginning PBR texturing: Learn physically based rendering with allegorithmic's substance painter.** 2020.

MCREYNOLDS, T., BLYTHE, D. **Advanced Graphics Programming Using OpenGL.** 2005.

MEHRSHAD. Mark Seem in UV Unwrapping + Good Texturing Guide (Step-by-Step) + Photos. 2022. Disponível em: www.steamcommunity.com/sharedfiles/filedetails/?id=2885101442. Acessado em: 9 de dezembro de 2023

MOTTA, R. L., & JUNIOR, J. T. **Short game design document (SGDD).** 2013.

PERRON, B. **Horror video games: Essays on the fusion of fear and play.** 2009.

PHARR, M., JAKOB, W., & HUMPHREYS, G. **Physically Based Rendering,** fourth edition: From Theory to Implementation. 2023. Disponível em: https://books.google.com.br/books?id=i9d2EAAAQBAJ. Acessado em: 9 de dezembro de 2023

PRESSANTO, B. P., **Jogos Independentes de Terror: Uma Proposta para Boas Experiências com Baixo Custo de Produção.** 2021

PRISMATICADEV. Additive Animations | Adv. Anim Application [UE4]. 2021. Disponível em: https://www.youtube.com/watch?app=desktop&v=flHL3qJB3_I&ab_channel=PrismaticaDev Acessado em: 9 de dezembro de 2023

PRISMATICADEV. Introducing: the Blendspace | Adv. Anim Application [UE4]. 2021. Disponível em: https://www.youtube.com/watch?app=desktop&v=flHL3qJB3_I&ab_channel=PrismaticaDevAcessado em: 9 de dezembro de 2023

RYANKINGART. UV Unwrapping for Beginners (Blender Tutorial). 2022. Disponível em: https://www.youtube.com/watch?v=qa_1LjeWsJg&ab_channel=RyanKingArt. Acessado em: 9 de dezembro de 2023

SANDERS, A. (2016). **An Introduction to Unreal Engine 4.** CRC Press. Disponível em https://books.google.com.br/books?id=2i8NDgAAQBAJ. Acessado em: 9 de dezembro de 2023

SEWELL, B. **Blueprints visual scripting for Unreal Engine.** 2015.

ŠMÍD, A. **Comparison of unity and Unreal Engine.** Czech Technical University in Prague. 2017.

UNREAL ENGINE. Unreal Engine 5.0 Documentation. 2022. Disponível em: https://docs.unrealengine.com/5.0/en-US/. Acessado em: 9 de dezembro de 2023

VRIES, J., Learn OpenGL. 2015. Disponível em: https://learnopengl.com/. Acessado em: 9 de dezembro de 2023

WIKIPEDIA. PERLIN NOISE. 2023. Disponível em: https://en.wikipedia.org/wiki/Perlin_noiser Acessado em: 9 de dezembro de 2023.

WIKIPEDIA. WHITE NOISE. 2023. Disponível em: https://en.wikipedia.org/wiki/White_noise Acessado em: 9 de dezembro de 2023.

WITKOWSKI, W. **Videogames are a bigger industry than movies and North American sports combined, thanks to the pandemic.** 2020.

WYMA, T. L. Help with Weight Painting Shoulders. 2020. Disponível em: https://blenderartists.org/t/help-with-weight-painting-shoulders/1236604. Acessado em: 9 de dezembro de 2023

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE GOIÁS
GABINETE DO REITOR

Av. Universitária, 1069 ● Setor Universitário
Caixa Postal 86 ● CEP 74605-010
Goiânia ● Goiás ● Brasil
Fone: (62) 3946.1000
www.pucgoias.edu.br ● reitoria@pucgoias.edu.br

RESOLUÇÃO n° 038/2020 – CEPE

ANEXO I

APÊNDICE ao TCC

Termo de autorização de publicação de produção acadêmica

O(A) estudante GABRIEL HÚGLIO PINELI SIMÕES do Curso de Ciência da Computação, matrícula 2019.1.0028.0057-7, telefone: (62) 99143-9883 e-mail gabrielhuglio@me.com, na qualidade de titular dos direitos autorais, em consonância com a Lei nº 9.610/98 (Lei dos Direitos do autor), autoriza a Pontifícia Universidade Católica de Goiás (PUC Goiás) a disponibilizar o Trabalho de Conclusão de Curso intitulado EMPREGO DE TÉCNICAS E ALGORITMOS PARA DESENVOLVIMENTO DE JOGOS REALISTAS, gratuitamente, sem ressarcimento dos direitos autorais, por 5 (cinco) anos, conforme permissões do documento, em meio eletrônico, na rede mundial de computadores, no formato especificado (Texto (PDF); Imagem (GIF ou JPEG); Som (WAVE, MPEG, AIFF, SND); Vídeo (MPEG, MWV, AVI, QT); outros, específicos da área; para fins de leitura e/ou impressão pela internet, a título de divulgação da produção científica gerada nos cursos de graduação da PUC Goiás.

Goiânia, 20 de dezembro de 2023.

Assinatura do(s) autor(es): Gabriel Húglio Pineli Simões

Nome completo do autor: GABRIEL HÚGLIO PINELI SIMÕES

Assinatura do professor-orientador: Max Gontijo

Nome completo do professor-orientador: MAX GONTIJO DE OLIVEIRA