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

DESENVOLVENDO JOGOS UTILIZANDO A ENGENHARIA DE SOFTWARE E A UNITY3D

NAEL BARRETO MARQUES

GOIÂNIA
2024

NAEL BARRETO MARQUES

DESENVOLVENDO JOGOS UTILIZANDO A ENGENHARIA DE SOFTWARE E A UNITY3D

Trabalho de Conclusão de Curso apresentado à Escola Politécnica e de Artes, da Pontifícia Universidade de Goiás, como parte dos requisitospara obtenção do título de Bacharel em Ciência daComputação

Orientador (a):

Profª. Dr. Solange da Silva.

Banca examinadora:

Prof. Msc. Fernando Gonçalves Abadia.

Prof. Esp. Anibal Vicente Vieira

GOIÂNIA
2024

NAEL BARRETO MARQUES

DESENVOLVENDO JOGOS UTILIZANDO A ENGENHARIA DE SOFTWARE E A UNITY3D

Trabalho de Conclusão de Curso aprovado em sua forma final pela Escola Politécnica e de Artes, da Pontifícia Universidade Católica de Goiás, para obtenção do título de Bacharel emCiências da Computação, em:_____/_____/_____.

_____

Orientador (a): Profª. Dr. Solange da Silva.

_____

Prof. Msc. Fernando Gonçalves Abadia

_____

Prof. Esp. Anibal Vicente Vieira

Este trabalho e dedicado à minha família, Ivone, Jorge e Elder.

AGRADECIMENTOS

Agradeço imensamente à minha mãe, por sua inabalável fé em mim e por ser a luz orientadora em minha jornada. Sua força e amor foram fundamentais na minha formação e são a base do meu sucesso.

Ao meu pai, estendo minha mais sincera gratidão. Obrigado por estar sempre presente, por cada sacrifício e por acreditar em mim.

Ao meu irmão, meu eterno parceiro e amigo, expresso minha profunda gratidão. Seu apoio constante e sua cumplicidade foram essenciais para superar os desafios e celebrar as conquistas.

À minha orientadora, Solange da Silva, minha sincera gratidão por sua dedicação e paciência. Sua orientação sábia e seu incentivo foram cruciais para a realização do meu TCC.

Aos professores que me ensinaram durante o curso, minha gratidão é imensa. Vocês não apenas compartilharam conhecimento, mas também inspiraram e moldaram o profissional que me tornei.

Aos meus colegas de faculdade, agradeço por todos os momentos compartilhados, pelas discussões enriquecedoras e pelo companheirismo que tornaram essa jornada acadêmica mais agradável.

E ao universo, agradeço por conspirar a meu favor, permitindo-me alcançar este estágio da minha vida.

RESUMO

O objetivo deste trabalho foi o de desenvolver um jogo de videogame (*game*), aplicando os conhecimentos da Engenharia de *Software* (ES). Foi realizada uma revisão bibliográfica com autores que abordam a temática Engenharia de *Software* e desenvolvimento de jogos e foi desenvolvido um jogo. Os resultados obtidos permitiram concluir que o projeto de desenvolvimento de jogos utilizando o conhecimento da área de ES foi essencial para alcançar alguns avanços no desenvolvimento de jogos. Notou-se que a ES permite um panorama geral acerca de soluções, tarefas e maneiras de implementar o jogo. A aplicação, mesmo que informal da ES no processo de desenvolvimento de jogo propiciou um norte para que todas as atividades vinculadas ao *design*, arte, implementação e busca de soluções técnicas redundassem em resultado satisfatório.

Palavras-chaves**:** Engenharia de *Software.* Desenvolvimento de Jogos. Entretenimento. Ferramenta UNITY.

ABSTRACT

The overall objective of this paper is to develop a video game, applying Software Engineering (SE) knowledge. A literature review was conducted with authors who address Software Engineering and game development. In addition, a simple game was developed. The results led to the conclusion that the game development project, utilizing SE expertise, was crucial in achieving advancements. It was observed that SE provides a comprehensive view of solutions, tasks, and ways to implement the game. Even the informal application of SE in the game development process provided guidance for all activities related to design, art, implementation, and seeking technical solutions, resulting in a satisfactory outcome.

Keywords: Software Engineering. Game Development. Entertainment. UNITY.

LISTA DE FIGURAS

LISTA DE SIGLAS

AI	*Artificial Intelligence*
CD	*Continuous Delivery*
CI	*Continuous Integration*
CPU	*Central Processing Unit*
ES	Engenharia de *Software*
FSM	*Finite State Machine*
GDD	*Game Design Document*
GDP	*Game Development Projects*
IDE	*Integrated Development Enviroment*
IK	*Inverse Kinematics*
LoS	*Line of Sight*
MDA	*Mechanics, Dynamics, Aesthetics*
NPC	*Non Playable Character*
RPG	*Role Playing Game*
RUP	*Rational Unified Process*
SE	*Software Engineering*
TCC	Trabalho de Conclusão de Curso
UML	*Unified Modeling Language*

SUMÁRIO

1 INTRODUÇÃO

A proposição do projeto "*Desenvolvendo Jogos utilizando a Engenharia de Software e a Unity3d*" parte do interesse pessoal por jogos e pela suposição de que é possível aplicar os conhecimentos da disciplina Engenharia de *Software* (ES) em uma experiência empírica que resulte no desenvolvimento de jogos por um estudante/amador.

Fez-se necessário consultar alguns autores essenciais na área de ES que tratam do tema para que norteassem o projeto e colaborassem com conceitos e procedimentos para o desenvolvimento de jogos estruturando o trabalho, abordando conceituação e características de *software* e jogos, aplicação da ES no desenvolvimento de jogos, apresentando trabalhos relacionados ao tema e mostrando os resultados da tentativa inicial e o estágio em que se encontra o projeto pessoal de desenvolvimento do jogo. Dentre os referenciais teóricos encontram-se: Pressman (2021), Sommerville (2019), Huizinga (2019), Caillois (2017), dentre outros autores.

Justifica-se estudar este tema porque conjugar os conhecimentos da ES ao desenvolvimento de jogos pois, os jogos são um fenômeno cultural popular, entretenimento que despertam interesse e fascinam, representam um mercado consolidado, lucrativo e em expansão. Este trabalho pretende apresentar uma experiência empírica de como o conhecimento da área de ES, tais como *design*, testes, gerenciamento de requisitos e outras disciplinas podem ser adaptadas e aplicadas no desenvolvimento de jogos amadores.

Diante deste contexto este trabalho visa responder a seguinte questão de pesquisa: **Seria possível desenvolver jogos aplicando conhecimentos da Engenharia de *Software* e a ferramenta *Unity3d*?**

Este trabalho tem o objetivo geral de desenvolver um jogo de videogame (*game*), aplicando conhecimentos da ES e utilizando a ferramenta *Unity3d*.

E os objetivos específicos são:

- Recorrer as bibliografias de ES e metodologias adequadas e relevantes para desenvolvimento de jogos.
- Expressar as insuficiências, impasses no decorrer do projeto.
- Criar protótipo do jogo, testar e validar a estética e a mecânica.
- Mostrar os resultados obtidos.

Espera-se que os resultados deste trabalho possam contribuir:

- Na percepção mais ampla do que significa desenvolver um jogo.
- No uso e adaptação de conhecimentos da ES aplicado ao desenvolvimento de jogos.
- Despertar o interesse de outros alunos pela área de desenvolvimento de jogos.

Esta monografia está estruturada da seguinte maneira: o Capítulo 1 traz a introdução com a questão de pesquisa e os objetivos, além dos resultados esperados. O Capítulo 2 traz o referencial teórico, juntamente com os subcapítulos que expandem sobres os assuntos: Conceituação e características do *Software*; Conceituação e características de jogos e sua relevância; Aplicação da ES no desenvolvimento de jogos e por fim trabalhos relacionados. O Capítulo 3 expõe os materiais e métodos utilizados neste projeto. O Capítulo 4 mostra alguns dos módulos desenvolvidos para o jogo e seus processos de desenvolvimento. O Capítulo 5 mostra os resultados e discussões acerca do projeto. O Capítulo 6 traz a conclusão obtida a partir de toda a experiência de desenvolvimento do projeto.

2 REFERENCIAL TEÓRICO

2.1 Conceituação, Características do Software

A Engenharia de *Software* conforme esclarece Sommerville (2019) “é uma disciplina de engenharia que se preocupa com os aspectos da produção de *Software*” (Sommerville, 2019, p.20). A ES está inserida na área da Ciência da Computação e responde às demandas e exigências no desenvolvimento de um produto satisfatório, Sommerville (2019) aponta, “o mundo moderno não funciona sem o *Software*” (Sommerville, 2019, p.20). Pressman (2021, p.73) expõe que “ao longo dos últimos 60 anos, o *Software* evoluiu de uma ferramenta especializada em análise de informações e resolução de problemas para uma indústria propriamente dita”.

No processo de desenvolvimento de *Software* “existem muitos tipos diferentes de sistemas de *Software* e não há um método universal de Engenharia de *Software* que seja aplicável a todos eles[...] não existem processos de *Software* universalmente aplicáveis” (Sommerville, 2019, p.20). Pressman (2021) conceitua *Software* como: “[...] instruções (programas de computador) [...] estruturas de dados [...] informação descritiva, tanto na forma impressa quanto na virtual, descrevendo a operação e o uso dos programas” (Pressman, 2021, p.53).

Os processos de desenvolvimento de *Software* são flexíveis, e não

necessariamente seguem uma fórmula fixa, permitindo que as equipes de desenvolvimento escolham ações e tarefas adequadas para entregar *Softwares* de qualidade e dentro do prazo, assim atendendo às necessidades dos patrocinadores e usuários (Pressman, 2021).

A definição de um processo de *Software*, segundo Pressman (2021) consiste em:

> Um processo é um conjunto de atividades, ações e tarefas realizadas na criação de algum artefato. Uma atividade [...] é utilizada independentemente do domínio de aplicação, do tamanho do projeto, da complexidade dos esforços ou do grau de rigor com que a Engenharia de *Software* será aplicada. Uma ação [...] envolve um conjunto de tarefas que resultam em um artefato de *software* fundamental [...]. Uma tarefa se concentra em um objetivo pequeno, porém bem-definido [...], e produz um resultado tangível (Pressman, 2021, p. 61).

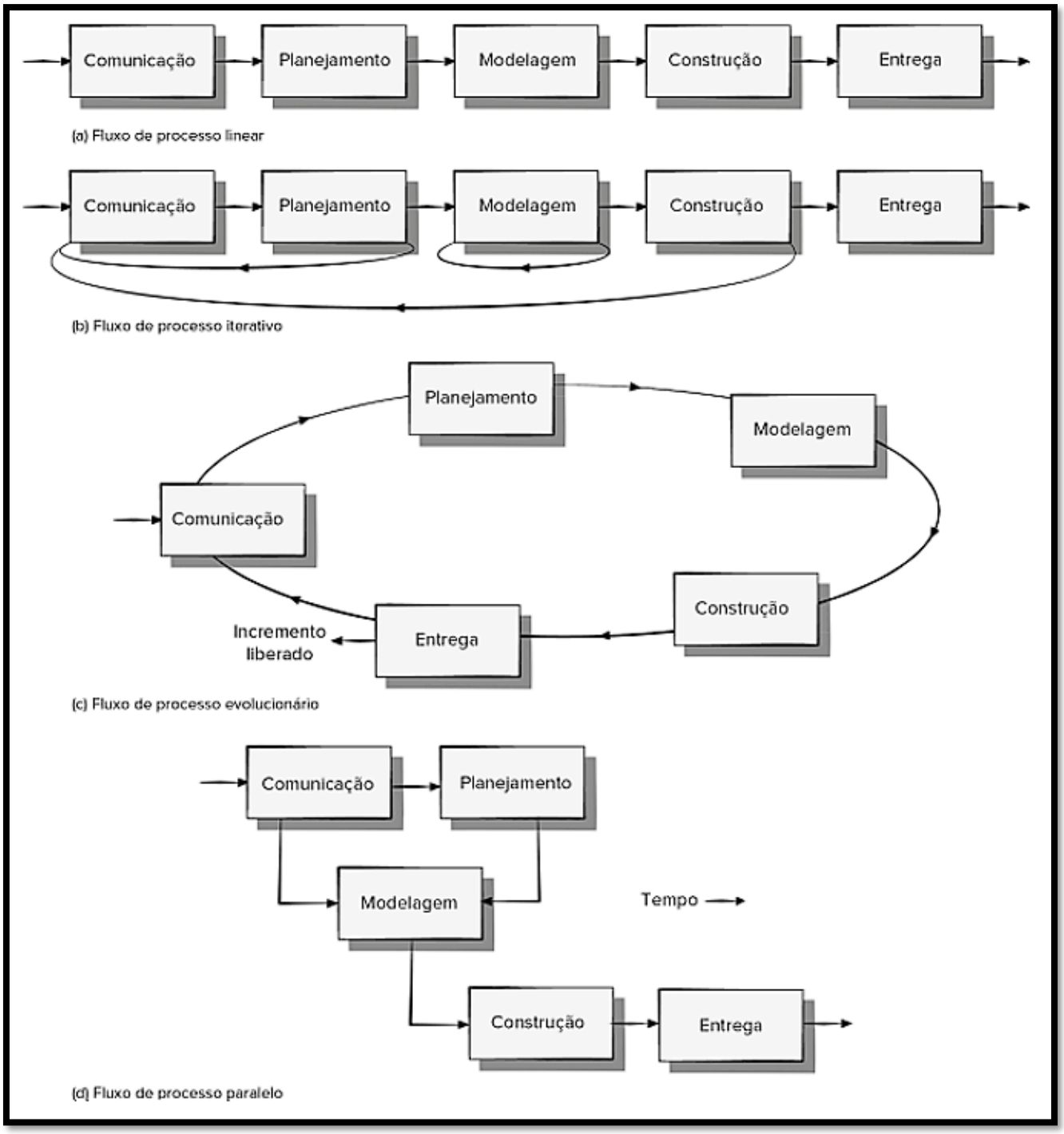
Sommerville (2019) pontua que “embora existam muitos processos de *Software* diferentes, todos eles devem incluir, de alguma forma, as quatro atividades fundamentais da Engenharia de *Software*” (Sommerville, 2019, p.29). E aponta estas atividades como sendo:

> 1.Especificação. A funcionalidade do *software* e as restrições sobre sua operação devem ser definidas. 2. Desenvolvimento. O *software* deve ser produzido para atender à especificação. 3. Validação. O *software* deve ser validado para garantir que atenda ao que o cliente deseja. 4. Evolução. O *software* deve evoluir para atender às mudanças nas necessidades dos clientes (Sommerville, 2019, p.30).

De acordo com Pressman (2021) processos de *Software* apresentam como característica o fluxo. O fluxo de processos de *Software* organiza atividades metodológicas e suas ações e tarefas em sequência e o tempo. Existem quatro tipos de fluxos: linear (sequencial), iterativo (repetitivo), evolucionário (circular para aprimoramento contínuo) e paralelo (atividades simultâneas) (Pressman, 2021).

A Figura 1 mostra a esquematização dos diferentes fluxos de processos de *Software*.

Figura 1: Fluxos de processos de *Software*



Fonte: Pressman (2021).

Segundo Pressman (2021) há 7 categorias de *Softwares* cujos desenvolvedores se debruçam em projetos e em muitas ocasiões trabalham com mais de uma categoria para responder às demandas e desafios, corrigindo, adaptando e aperfeiçoando os *Softwares*. Dentre as categorias de *Softwares* Pressman (2021) cita:

“*Software* de sistema, *Software* de aplicação, *Software* de Engenharia/científica, *Software* embarcado, *Software* para linha de produtos, *Software* de inteligência artificial, *Web*/aplicativos móveis” (Pressman, 2021, p.26).

Dentre várias aplicações dos conhecimentos da ES uma das aplicações consiste em “Sistemas de entretenimento. [...] a maioria desses sistemas consiste em jogos de gêneros variados, que podem ser executados em um *console* concebido especificamente para essa finalidade” (Sommerville, 2019, p. 26).

Jogos são considerados *Softwares*, logo em seu processo de produção e desenvolvimento e na escolha da metodologia demandam o entendimento e satisfação de requisitos específicos do jogo a serem implementados: requisitos funcionais que vão definir o que o jogo deve fazer, enquanto os requisitos não-funcionais abordam questões como desempenho, segurança e usabilidade (Sommerville, 2019).

Os *Softwares* de aplicação voltados para o desenvolvimento de produtos como jogos de videogame (*games*) possuem os mesmos desafios de produção de *Softwares* genéricos de uso comercial, contudo, jogos não podem prescindir de características artísticas, pois, jogos são arte, aponta Melissinos (2015): “A tecnologia expandiu a tela na qual os artistas podem pintar e contar suas histórias. [...] uma forma de arte que existe apenas no espaço digital, os jogos digitais são o fruto da colisão entre arte e ciência” (Melissinos, 2015, p.01). Segundo Smuts (2005) o desenvolvimento de *games* também é uma atividade multidisciplinar.

> Os *designers* modernos de videogames estão profundamente preocupados com considerações estéticas tradicionais familiares aos animadores, romancistas, designers de cenários para produções teatrais e diretores de arte para filmes. O desenvolvimento de ambientes de jogo é um processo intensivo que envolve a criação de mapas de níveis, fontes de iluminação, detalhes de configuração e complexidade de textura visual. Assim como o autor de um romance realista ou o designer de cenários de um filme pode colocar adereços em uma sala, os designers de níveis visam a incorporação consistente de detalhes para dar vida ao mundo do jogo (Smuts, 2005, p.08).

2.2 Conceituação, características de jogos

Os *games* ou jogos conquistaram seu espaço na sociedade, representando não somente um fenômeno da cultura, mas sobretudo, da estética, da arte e da linguagem. Para Caillois (2017, p. 31) “o jogo é ocasião de gasto puro: de tempo, de energia, de engenhosidade, de destreza e, muitas vezes, de dinheiro – para a compra dos acessórios do jogo ou para pagar eventualmente o aluguel do local”.

Os jogos são, “[...] incontáveis e variados: jogos de sociedade, de destreza, de

azar, jogos ao ar livre, de paciência, de construção etc. [...]" (Caillois, 2017, p.14) e exercem influência sobre todos os aspectos culturais fazendo parte da cultura de massa. No jogo virtual, os jogadores são levados a busca pela emoção imersiva através do fictício, do lúdico, do desafiador no qual reinam termos como prêmio, aposta, conquista, competição, vencedor, perdedor (Ramos, 2022). Huizinga (2007) caracteriza jogos em geral como:

> Resumindo as características formais do jogo, poderíamos considerá-lo uma atividade livre, conscientemente tomada como "não séria" e exterior à vida habitual, mas ao mesmo tempo capaz de absorver o jogador de maneira intensa e total. É uma atividade desligada de todo e qualquer interesse material, com a qual não se pode obter qualquer lucro, praticada dentro de limites espaciais e temporais próprios, segundo uma certa ordem e certas regras (Huizinga, 2007, p. 33).

A indústria dos jogos de videogame na sociedade contemporânea, seja para PC's, *tablets* ou *consoles* é expressiva no aspecto da importância e da rentabilidade, superando a soma em arrecadação financeira produzida pelos mercados da música e dos filmes Wakka (2021).

> Um novo estudo da TechNET *Immersive* apontou que a indústria de jogos está avaliada em US$ 163,1 bilhões. Com isso, o setor é responsável por mais da metade do valor da indústria de entretenimento, confirmando a já conhecida estatística de que é maior que o mercado de cinema e música juntos. [...] O mercado viu alto crescimento em 2020, por conta da pandemia. A estimativa do mesmo estudo é de que o setor estivesse avaliado em US$ 152 bilhões em 2019 (Wakka, 2021).

O mercado para jogos está em franca expansão devido ao interesse que desperta em novos usuários dispostos a investir tempo e recursos financeiros em jogos que propiciem entretenimento, aprendizagem e imersão, notada na 9ª edição da Pesquisa Games Brasil (2022), que faz o levantamento anual sobre o consumo de jogos no país apontando "um crescimento geral no público de games dentro do país: 74,5% da população jogou games em 2022, um crescimento de 2,5% em relação ao último ano" (Redação do GE, 2022).

> A indústria global de games movimentou US$ 175,8 bilhões em 2021, de acordo com os últimos dados consolidados e preliminares da consultoria *Newzoo*. Esse montante apresentou uma ligeira queda de -1,1% em relação a 2020, mas nada que afete o desempenho de alta dos próximos anos que deve levar um dos principais segmentos do entretenimento a movimentar mais de US$ 200 bilhões em 2023 (Pacete, 2022).

Jogo é um fenômeno presente em contextos culturais e sociais que redundam em ativos comercial/industrial significativos, mas quando se consegue aliar jogo/educação/lúdico o produto se torna muito mais atraente, o jogo potencializa a exploração e a construção do conhecimento, pois conta com a motivação interna,

típica do lúdico (Kisimoto, 2017).

Conjugar o aprendizado da ES ao desenvolvimento de jogos de videogame torna a teoria mais acessível e estimulante. Uma pesquisa da *National Research Council* (2000) registrou que “Estudantes de todas as idades ficam mais motivados quando conseguem ver a utilidade do que estão aprendendo e quando podem usar esta informação para fazer algo que tenha impacto para outros” *(National Research Concil,* 2000, p.61).

2.3 Aplicação da ES no desenvolvimento de jogos

A aplicação prática dos conhecimentos da ES no desenvolvimento de jogos envolve enorme complexidade pois engloba “os 4 P’s: pessoas, produto, processo e projeto” (Pressman, 2021, p.886), e sujeito de perfil multifacetado, sendo, simultaneamente, gestor, gerente de projetos, *design* gráfico, desenvolvedor e programador. Além do conhecimento técnico básico em desenvolvimento de *Software*. “Para gerenciar um projeto de *Software* bem-sucedido, é preciso saber o que pode sair errado, de modo que os problemas possam ser evitados” (Pressman, 2021, p.905).

Em sua obra de 2022 Bond destaca a proliferação de *frameworks*[1] no desenvolvimento de jogos, salientando um em particular: o *framework* MDA (*Mechanics*, *Dynamics*, *Aesthetics*) capaz de se moldar a qualquer estilo de jogo, tanto analógico quanto digital. (Bond, 2022)

O *framework* MDA, conforme descrito por (Hunicke; Leblanc; Zubek, 2004), oferece uma lente poderosa para analisar e compreender jogos. Ele divide os jogos em três componentes: Mecânica; Dinâmica; Estética. Este *framework* destaca a interação entre esses componentes e a maneira como eles trabalham juntos para criar uma experiência de jogo envolvente.

De acordo com Hunicke, Leblanc e Zubek (2004) a componente “mecânica” do *framework* é a base de qualquer jogo. Ela se refere às regras e sistemas que governam o jogo. Isso inclui cada ação básica que o jogador pode realizar no jogo, os

[1] “Um *framework* é uma estrutura genérica estendida para criar um subsistema ou aplicação mais específicos. Schmidt, Gokhale e Natarajan (2004) definem um *framework* como um conjunto integrado de artefatos de *Software* (como classes, objetos e componentes) que colaboram para proporcionar uma arquitetura reusável para uma família de aplicações relacionadas” (Sommerville, 2019, p.414-415 apud Schmidt D. C; Gokhale. A; Natarajan, B, 2004, p. 66-75).

.

algoritmos e as estruturas de dados o motor gráfico do jogo (Hunicke; Leblanc; Zubek, 2004). A mecânica é a fundação sobre a qual o jogo é construído e é a primeira coisa que um *designer* de jogos define ao criar um novo jogo (Bond, 2022). A mecânica define as possibilidades e limitações dentro do jogo e, portanto, tem um impacto direto na experiência do jogador (Hunicke; Leblanc; Zubek, 2004).

O componente nomeado “dinâmica” do *framework* se refere ao comportamento em tempo real da mecânica agindo sobre a entrada do jogador e ‘cooperando’ com outras mecânicas. A dinâmica é o que acontece quando o jogador começa a interagir com as regras e sistemas do jogo definidos pela mecânica (Hunicke; Leblanc; Zubek, 2004). Isso pode incluir estratégias emergentes, ações dos jogadores e a interação entre diferentes elementos mecânicos (Hunicke; Leblanc; Zubek, 2004). A dinâmica é, em essência, o fluxo do jogo que emerge da interação do jogador com a mecânica.

A estética se refere às respostas emocionais desejáveis evocadas no jogador quando ele interage com o sistema do jogo. A estética é a experiência emocional que o jogador tem ao interagir com o jogo. Isso pode incluir os visuais, efeitos sonoros, narrativa, porém de forma mais abstrata, a reação emocional que você deseja que os jogadores tenham (Hunicke; Leblanc; Zubek, 2004).

No componente “Mecânica” do *framework* MDA é citado o uso de *game engines*.

> ***Game Engines*** **(Motores de Jogos)**, também chamadas simplesmente de ***engines***, são *Softwares* que reúnem diversas ferramentas necessárias ao desenvolvimento de **jogos digitais** como física, luzes, áudio, mecânicas, animações, entre outras (Marques, 2020).

Conforme esclarece Marques (2020) *Unity*, *Unreal*, Godot, *Game Maker Studio* são exemplos de *engines* populares disponíveis no mercado (Marques, 2020).

A integração do *framework* MDA na concepção do (GDD) *Game Design Document* ou Documento de *Design* de Jogo é um processo vital, enriquecendo cada aspecto do jogo - desde sua funcionalidade e gênero até os personagens e o enredo.

O GDD emerge como um farol orientador, delineando a essência do jogo, seus protagonistas, ambientes, mecânicas, técnicas especializadas, paisagens e efeitos sonoros (Ricchiuti, 2023). Ricchiuti (2023) esclarece a ideia do GDD:

> Imagine um GDD como um conjunto de módulos, onde você pode adicionar novos e remover aqueles que não funcionam toda vez que você escreve um novo [...]. O cerne deste tipo de documento é permitir que você adicione ou remova qualquer coisa dependendo de todos os possíveis fatores envolvidos, desde o estilo da equipe até o elemento específico sobre o qual você está

> falando: você pode adicionar fluxogramas, imagens, gráficos e realmente qualquer coisa[...] (Ricchiuti, 2023, p.250).

Documentos de *design* de Jogo são extremamente adaptáveis e não seguem um modelo fixo, pois variam significativamente dependendo do seu propósito específico, seja ele detalhar uma mecânica, sistema ou personagem. Cada documento é único e pode exigir formatos distintos, mesmo entre documentos do mesmo tipo (Ricchiuti, 2023). “Este tipo de documento pode ser tão personalizado que assume o nome do departamento em que é utilizado: documento de *design* de jogo, documento de *design* de nível, documento de *design* de personagem, e assim por diante [...]” (Ricchiuti, 2023, p. 227).

Portanto, o GDD é um documento essencial que descreve de forma clara e abrangente tanto as características específicas do jogo quanto os requisitos funcionais e não funcionais do *Software*, facilitando o entendimento dos objetivos do projeto por todos os participantes do desenvolvimento. A Figura 2 apresenta o processo de desenvolvimento de jogos e suas atividades prevendo a criação do GDD em uma de suas etapas.

Figura 2: Processo de desenvolvimento de jogos

Fonte: Autoria Própria.

O Documento de *Design* de Jogo (GDD) é fundamentado pelos princípios da ES, incorporando ao menos duas de suas atividades essenciais que são cruciais para todos os projetos de *Software*. Conforme Pressman (2021) as atividades incluem comunicação, planejamento, modelagem, construção e entrega, formando a espinha dorsal do processo de desenvolvimento (Pressman, 2021).

A Engenharia de *Software* desempenha um papel crucial na fase de pré-produção, contribuindo com sua expertise para o planejamento meticuloso, a elaboração detalhada do *design* e o levantamento preciso dos requisitos. Este processo culmina na criação de um protótipo funcional do jogo, que serve como uma ferramenta valiosa para coletar *feedback*[2] sobre a experiência do usuário, permitindo que os jogadores testem e explorem a jogabilidade[3]. O *feedback* é essencial para identificar e refinar as áreas que necessitam de ajustes e aprimoramentos, garantindo assim a evolução contínua do produto.

A utilização do processo de desenvolvimento incremental-iterativo mostrou ser uma abordagem coerente por combinar planejamento, desenvolvimento e avaliação em um ciclo contínuo e iterativo, porém de escopo reduzido focando-se em pequenas funcionalidades a cada iteração.

2.4 Trabalhos relacionados

Existem alguns trabalhos na área de desenvolvimento de jogos auxiliados pela ES que merecem destaque e que colaboraram para o escopo deste trabalho.

2.4.1 Engenharia de Software e jogos digitais: uma experiência de ensino e extensão

O artigo de Leal (2023) apresenta sobre a “Engenharia de *Software* e jogos digitais: uma experiência de ensino e extensão”. A pesquisa adotou uma abordagem metodológica que integra atividades interdisciplinares e multidisciplinares, alinhadas com políticas educacionais e diretrizes, e favoreceu a orientação por projetos sempre que viável.

Foram empregados métodos e *frameworks* de gerenciamento de projetos de *Software*, incluindo aqueles baseados em planejamento e métodos ágeis. O foco do estudo foi proporcionar aos estudantes e outros participantes a oportunidade de aprender Engenharia de *Software* através do desenvolvimento de jogos digitais.

O autor concluiu que os resultados obtidos foram encorajadores e geraram interesse entre outros estudantes e discentes.

[2] *feedback*: informação que o emissor obtém da reação do receptor à sua mensagem, e que serve para avaliar os resultados da transmissão (Wikipedia, 2024).
[3] Leal (2023) traça o conceito do elemento Jogabilidade como o nível qualitativo de interação do jogo com as ações do jogador, é o resultado do conjunto das características de um jogo e precede a Interação (Leal, 2023).

2.4.2 The role of Sprint planning and feedback in game development projects: Implications for game quality

O artigo produzido por Jing-Wei Liu et al. (2019) explora como o planejamento de *Sprints* e o *feedback* específico influenciam na qualidade dos jogos em projetos de desenvolvimento de jogos (GDPs).

O estudo revela que o planejamento de *Sprints* teve um impacto positivo na qualidade do jogo, enquanto o *feedback* específico dos testadores de jogos ajuda os desenvolvedores a transformarem o conceito abstrato do jogo em uma versão jogável concreta.

O estudo aprofunda o entendimento do papel dos testadores de jogos, demonstrando como eles podem contribuir para o sucesso de um GDP.

Os autores concluíram que o *feedback* iterativo dos testadores é visto como a melhor maneira de reforçar a flexibilidade e adaptabilidade do planejamento de *Sprints*, permitindo que os desenvolvedores aprendam imediatamente se a experiência desejada pelo usuário foi realizada e encontrem mais oportunidades para melhorar os recursos do jogo na próxima iteração.

2.4.3 XS-GAMES: Engenharia de jogos voltada para desenvolvedores individuais

O trabalho realizado pelo então estudante de Engenharia de Computação de Informação Confessor (2019), foca em desenvolver um protótipo de jogo digital, para o qual o autor adaptou modelos existentes e propôs uma nova abordagem voltada para desenvolvedores autônomos.

Durante o processo, foram utilizados métodos ágeis de desenvolvimento, como *Scrum* e *Extreme Programming*, além de ferramentas como *Unity* 3D e a linguagem de programação C#. O *Kanban*[4] também foi empregado como um *framework* para gerenciamento do desenvolvimento do produto.

O autor concluiu que a documentação de artefatos como um desafio significativo, devido à natureza solitária do projeto, que resultou em períodos de alta carga de trabalho conhecidos como "*crunch time"*. O autor sugeriu para trabalhos futuros para que a documentação interna fosse simplificada para facilitar a

[4] *Kanban* é uma estrutura popular usada para implementar o desenvolvimento de software *Agile* e *DevOps*. Requer comunicação de capacidade em tempo real e total transparência do trabalho. Os itens de trabalho são representados visualmente em um quadro *Kanban*, permitindo que os membros da equipe vejam o estado de cada trabalho a qualquer momento (Atlassian, 2024).

manutenção, reduzindo a formalidade de certos artefatos, mas preservando a qualidade do GDD, defendendo a implementação do GDD como uma prática essencial no processo.

2.4.4 Engenharia de Software: jogos eletrônicos

O trabalho de conclusão de curso de Freitas (2017) apresenta a Engenharia de *Software* com foco na criação de jogos eletrônicos, utilizando como estudo de caso um jogo RPG, desenvolvido para a plataforma *web*.

O documento detalha as metodologias empregadas, incluindo linguagens de programação orientada a objetos (Java), UML e RUP, apresentando o crescimento do desenvolvimento de jogos no Brasil e no mundo.

O autor conclui que a Engenharia de *Software* se baseia em modelos abstratos para a criação e manutenção de sistemas de qualidade. Além disso, no desenvolvimento de jogos, são cruciais a interdisciplinaridade e a criatividade para atingir o sucesso, pois diante da evolução constante da tecnologia a adaptação responde à complexidade dos desafios no processo de desenvolvimento.

3 MATERIAIS E MÉTODOS

Esta pesquisa, segundo sua natureza é um resumo de assunto, buscando a compreensão acerca do tema e a partir da investigação e observação das informações contiguas compreender suas causas e explicações (Wazlawick, 2014).

Segundo seus objetivos é de caráter exploratório, segundo Gil (2021): “As pesquisas exploratórias têm como propósito proporcionar maior familiaridade com o problema, com vistas a torná-lo mais explícito ou a construir hipóteses” (Gil, 2021, p. 27).

Segundo seus procedimentos técnicos, esta pesquisa é bibliográfica e experimental. De acordo com Gil (2021) a pesquisa bibliográfica é elaborada com base em material já publicado, inclui materiais impressos, como livros, revistas, jornais, teses, dissertações e anais de eventos científicos, bem como mídias digitais.

Toda pesquisa científica e considerada bibliográfica em sua concepção devido a necessidade de levantamento teórico para fundamentação do trabalho científico. (GIL, 2021)

Conforme Gil (2021) pesquisas bibliográficas seguem as seguintes etapas:

a) Escolha do tema: Desenvolvimento de games.

b) Levantamento bibliográfico preliminar: levantamento bibliográfico de acordo com área de interesse do pesquisador com intuito de familiarização com o tema e facilitação na definição do problema de pesquisa.

c) Formulação do problema: Seria possível um estudante desenvolver um jogo amador aplicando os conhecimentos da Engenharia de *Software*?

d) Busca das fontes: Fontes em artigos científicos disponíveis na plataforma Capes e Google *academy*, Livros sobre o tema obtidos em acervo pessoal. Tais fontes propiciaram a formulação do problema de pesquisa e a elucidação dos processos envolvidos acerca do tema.

e) Leitura do material: Identificação de pontos chaves correlacionadas ao tema de pesquisa.

f) Fichamento: Fichamento de citação produzido a partir do material bibliográfico selecionado afim guardar citações importantes acerca do tema.

g) Organização lógica: Foram organizadas as ideias com o propósito de realizar os objetivos da pesquisa.

h) Redação do texto: Escrita do TCC1.

Pesquisas experimentais consistem em determinar o objeto de estudo, selecionar variáveis capazes de influenciá-lo e definir as formas de controle e de observação dos efeitos que a variável produz no objeto de estudo. (Gil, 2021). Logo, de acordo com Gil (2021) esta pesquisa experimental apresenta as seguintes etapas:

a) Formulação do problema: Seria possível um estudante desenvolver um jogo amador aplicando os conhecimentos da Engenharia de *Software*?

b) Construção das hipóteses: -Aplicar a teoria da ES viabiliza o desenvolvimento do jogo por um estudante amador.

c) Operacionalização da variável: A variável de pesquisa foi o “modelo de processo de *Software*”. Especificamente, foi analisado o impacto do modelo incremental-iterativo no desenvolvimento do projeto em comparação com um projeto anterior sem um processo definido.

d) Determinação do ambiente: O ambiente de pesquisa foi todo virtual, utilizando duas máquinas (PC’s): ***Desktop*****:** CPU: AMD Ryzen 5 3600, RAM: Kingston 32gb a 3200mhz, *MotherBoard*: ASUS B550M e Placa Gráfica: ASUS GTX3060TI, Sistema Operacional: *Windows* 11 *Home version* 22H2, trabalhando unicamente com modelagem e texturização 3d, utilizando do *Software* de modelagem *Blender* 4.0 e *Adobe Substance Painter*. ***Laptop*****:** CPU: AMD Ryzen 7 3700u, RAM: 8gb a 2400mhz,

MotherBoard: *unknown* e Placa Gráfica: Radeon RX Vega 10, Sistema Operacional: WIN 11 *Home version* 22H2, trabalhando com o motor gráfico utilizando a *game engine Unity* 2022.1.22 e ambiente de desenvolvimento integrado (IDE) *Visual Studio* 19, além do *Github Desktop* para controle de versão do projeto.

e) Coleta de dados: A coleta de dados se deu, por meio da medição do tempo decorrido desde o início de uma etapa de desenvolvimento até sua conclusão.

f) Análise e interpretação dos dados: O processo de análise e interpretação se deu a partir da comparação do tempo de desenvolvimento em comparação com diferentes de modelos de processo, o modelo de processo que apresentar menor tempo para geração do artefato (GDD) foi considerado ideal.

g) Redação do relatório: escrita do TCC.

4 DESENVOLVIMENTO DO JOGO

4.1 Experiência passada de lições aprendidas

No início do primeiro período acadêmico, o desenvolvimento do primeiro projeto de jogo tomou forma. A disciplina de Engenharia de *Software* (ES) mostrou-se fundamental, embora, na prática, não tenha conseguido integrar plenamente seus princípios à primeira versão do jogo. Essa versão inicial acabou priorizando os aspectos visuais, negligenciando os pilares essenciais do desenvolvimento de software como, especificação, desenvolvimento, validação e evolução.

Esta experiência mostrou que, embora os aspectos visuais sejam cruciais para capturar o interesse do usuário, eles não podem eclipsar a importância de uma base sólida em Engenharia de *Software*. Foi enfrentado lacunas e desafios significativos que nos levaram a revisitar a teoria e replanejar nossas ações. Isso resultou na suspensão do primeiro protótipo, mas a experiência adquirida provou ser extremamente valiosa, servindo de alicerce para o desenvolvimento do segundo projeto.

Durante essa jornada foram identificados pontos críticos para o sucesso do desenvolvimento de *Software, a seguir*:

- Especificação Clara dos Requisitos: Uma compreensão precisa do que o *Software* deve realizar é vital para evitar retrabalho e assegurar que todos os aspectos funcionais sejam contemplados.
- Abordagem Modular: Segmentar o projeto em módulos menores, que podem

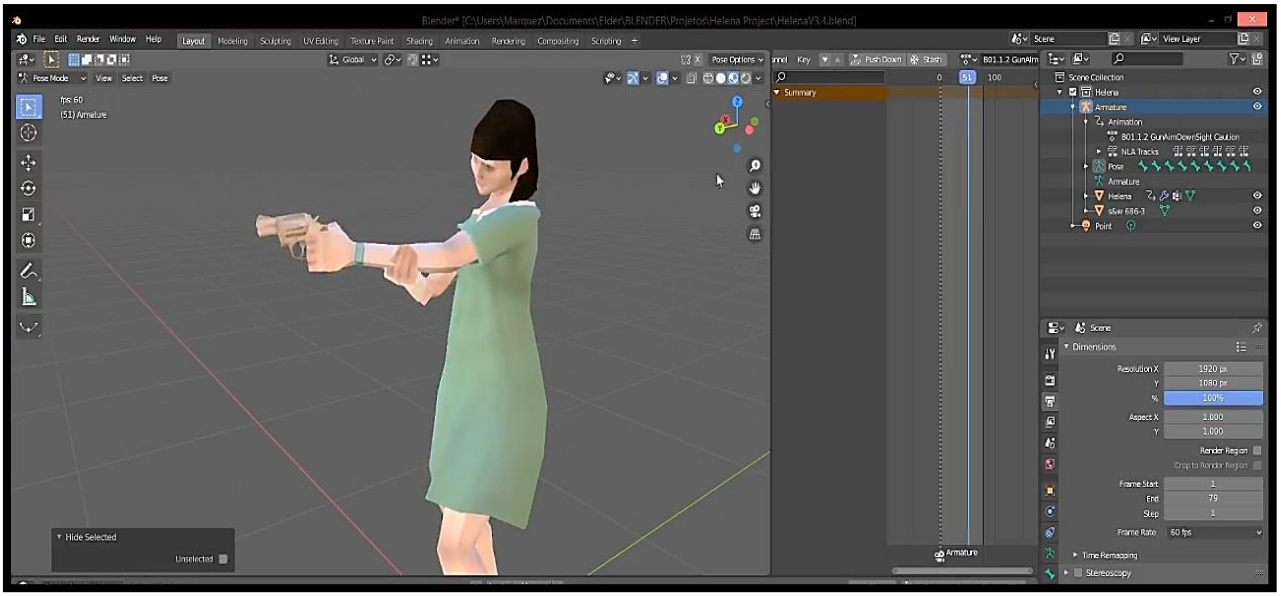
ser desenvolvidos, testados e validados de forma independente, isto simplifica a gestão do projeto e facilita a incorporação de novas funcionalidades ou alterações.

- Padrões de *Design*: A adoção de padrões de *design* comprovados ajuda a resolver problemas comuns de desenvolvimento, melhorando a manutenibilidade e escalabilidade do *Software*.
- Realização de Testes Contínuos: Testes automatizados e manuais contínuos são essenciais para assegurar que o *Software* funcione como esperado e ajuda a identificar problemas precocemente, seja no *design* ou na implementação.
- Planejamento da Evolução: O *Software* é dinâmico por natureza. É importante considerar como ele pode evoluir com o tempo, levando em conta novas tecnologias, o *feedback* dos usuários e as mudanças de mercado.

Integrando as lições adquiridas durante o breve desenvolvimento do primeiro projeto, conseguiu-se uma abordagem de desenvolvimento mais equilibrada para o segundo projeto, valorizando tanto a funcionalidade quanto a estética.

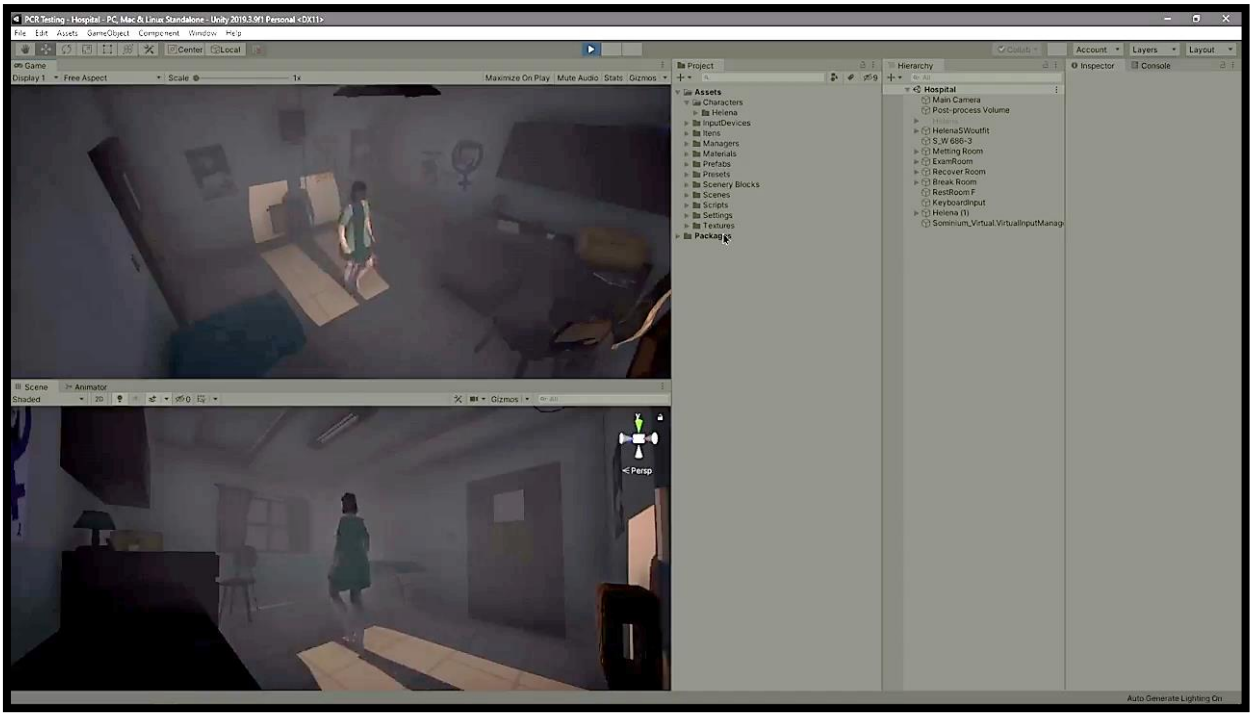
As Figuras 3 e 4 mostram diferentes estágios de desenvolvimentos do primeiro projeto.

Figura 3: Protagonistas do 1º projeto em processo de animação



Fonte: Autoria Própria.

Figura 4: Protagonista em uma sala médica no ambiente do jogo



Fonte: Autoria Própria.

4.2 Documentação

No 2º projeto, a teoria da ES foi valorizada, especialmente no que tange ao planejamento, organização e documentação de processos. Este projeto, de escopo mais limitado, não buscou cobrir a totalidade do conteúdo planejado para o jogo e sim, contemplá-lo parcialmente (ATO 1), devido a restrições temporais e a necessidade de uma visualização clara de todos os componentes do segmento de projeto. A estrutura de três atos é uma técnica clássica de narrativa usada em roteiros, peças de teatro e outras formas de contar histórias. Buzz McLaughlin (2017) descreve a estrutura da seguinte forma:

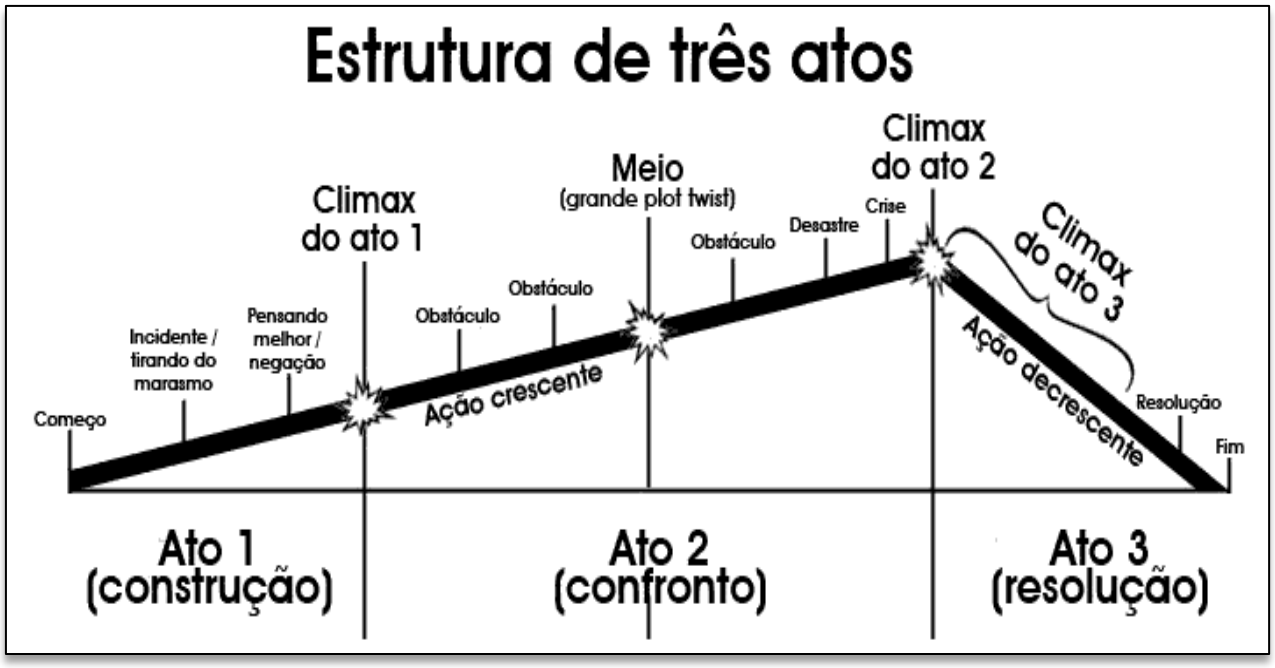
- **Primeiro Ato (Introdução)**: Este é o início da história, onde os personagens principais e o cenário são apresentados. O conflito principal é introduzido, e o protagonista é colocado em uma situação que requer ação. Este ato geralmente termina com um evento incitante que leva ao próximo ato.
- **Segundo Ato (Desenvolvimento)**: Este é o corpo da história, onde o conflito se desenvolve e as tensões aumentam. O protagonista enfrenta vários obstáculos e desafios. Este ato é frequentemente dividido em duas partes: a primeira metade, onde o protagonista tenta resolver o conflito, e a segunda

metade, onde as coisas pioram, levando a um ponto de virada importante.

- **Terceiro Ato (Conclusão)**: Este é o clímax e a resolução da história. O conflito atinge seu ponto mais alto e é finalmente resolvido. O protagonista enfrenta o desafio final e as consequências de suas ações são reveladas. Este ato conclui a história, amarrando todas as pontas soltas e proporcionando uma sensação de fechamento.

A Figura 5 mostra as disposições dos atos que compõem uma boa história contendo os pontos chaves de cada ato.

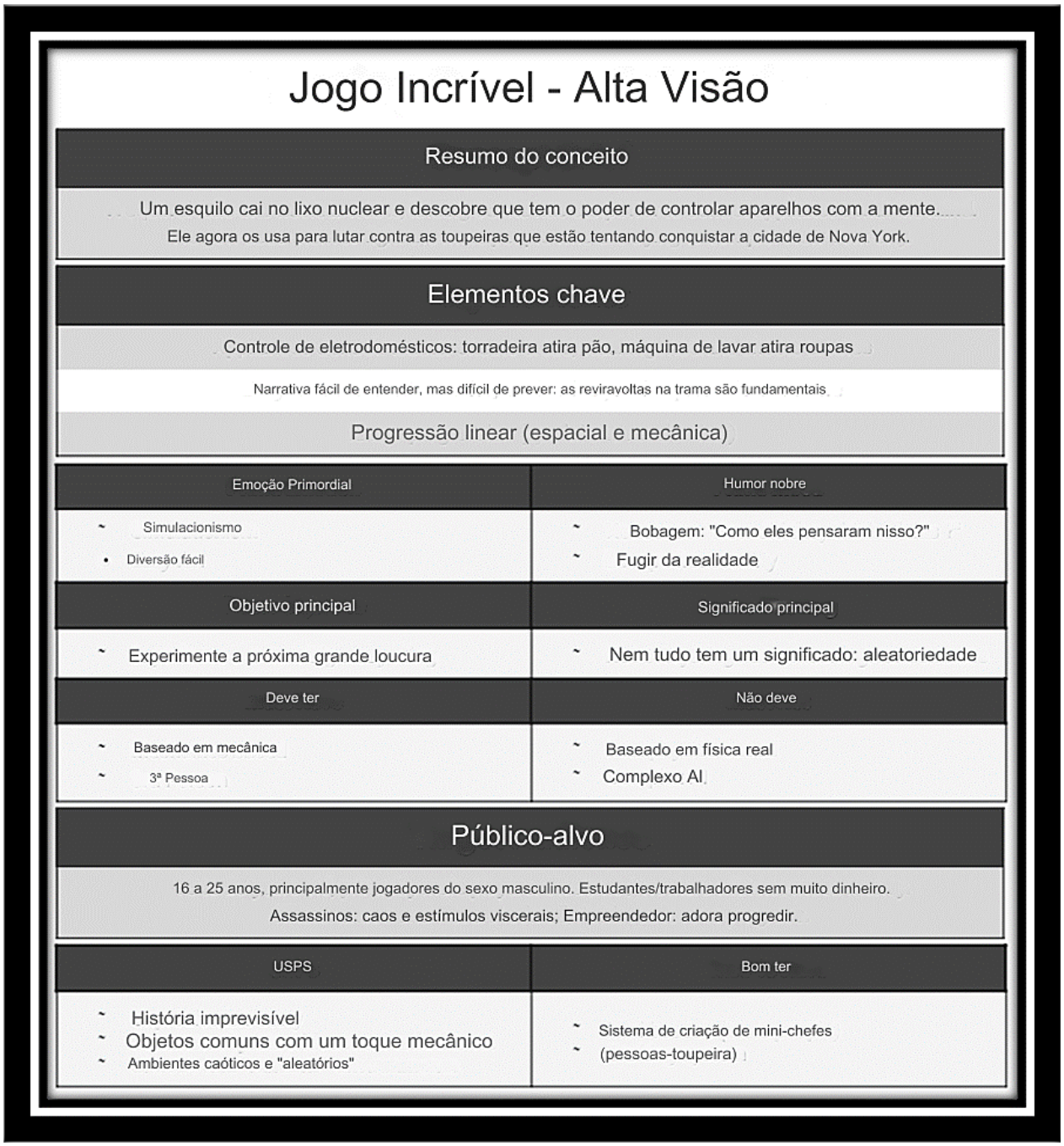
Figura 5: Estrutura de Atos que compõem uma história



Fonte: Buzz McLaughlin (2017).

A estratégia adotada enfatizou a particularização de ações e decisões inerentes ao projeto e ao processo criativo, por meio da elaboração do *Game Design Document* (GDD). Foram redigidos documentos auxiliares, indispensáveis para este projeto, como o Documento de Visão, que articula a concepção do jogo pelo líder do projeto de maneira breve e sucinta (Ricchiuti, 2023). Este documento normalmente é o primeiro documento a ser criado, sendo justamente a “ideia escrita no papel”. A Figura 6 exemplifica o documento de visão e seus campos.

Figura 6: Modelo referência para documento de visão



Fonte: Ricchiuti (2023) Traduzido por *Google Image Translator*.

Em seguida o documento de *Character and Enviroment,* este, descreve as amplas técnicas de criação de personagens e terrenos. O documento de *Character and Enviroment* serve de suporte para especificação de módulos relacionados ao *design* contidos no GDD, especificando passos para criação de cenário e personagens. A Figura 7 mostra o conteúdo da primeira página do documento de *Character and Enviroment* desenvolvido para este projeto.

Figura 7: 1ª página do documento de *Character and Enviroment*

SOMINIUM VIRTUAL

Técnicas e implementações de criação visual para jogos

PERSONAGEM

Passo 1: Conceito de modelo

Neste passo o artista encarregado devera criar uma serie de ilustrações em que represente sua ideia visual de uma descrição textual base fornecida previamente. O intuito do conceito é alinhar de forma coerente todos os setores de criação do jogo, programação e design devem entrar em consenso tendo em vista futuras interações do personagem com seu meio digital, deixando em evidenciar possíveis limitações e possibilidades a serem implementadas para criar um balanço entre aparência e funcionalidade.

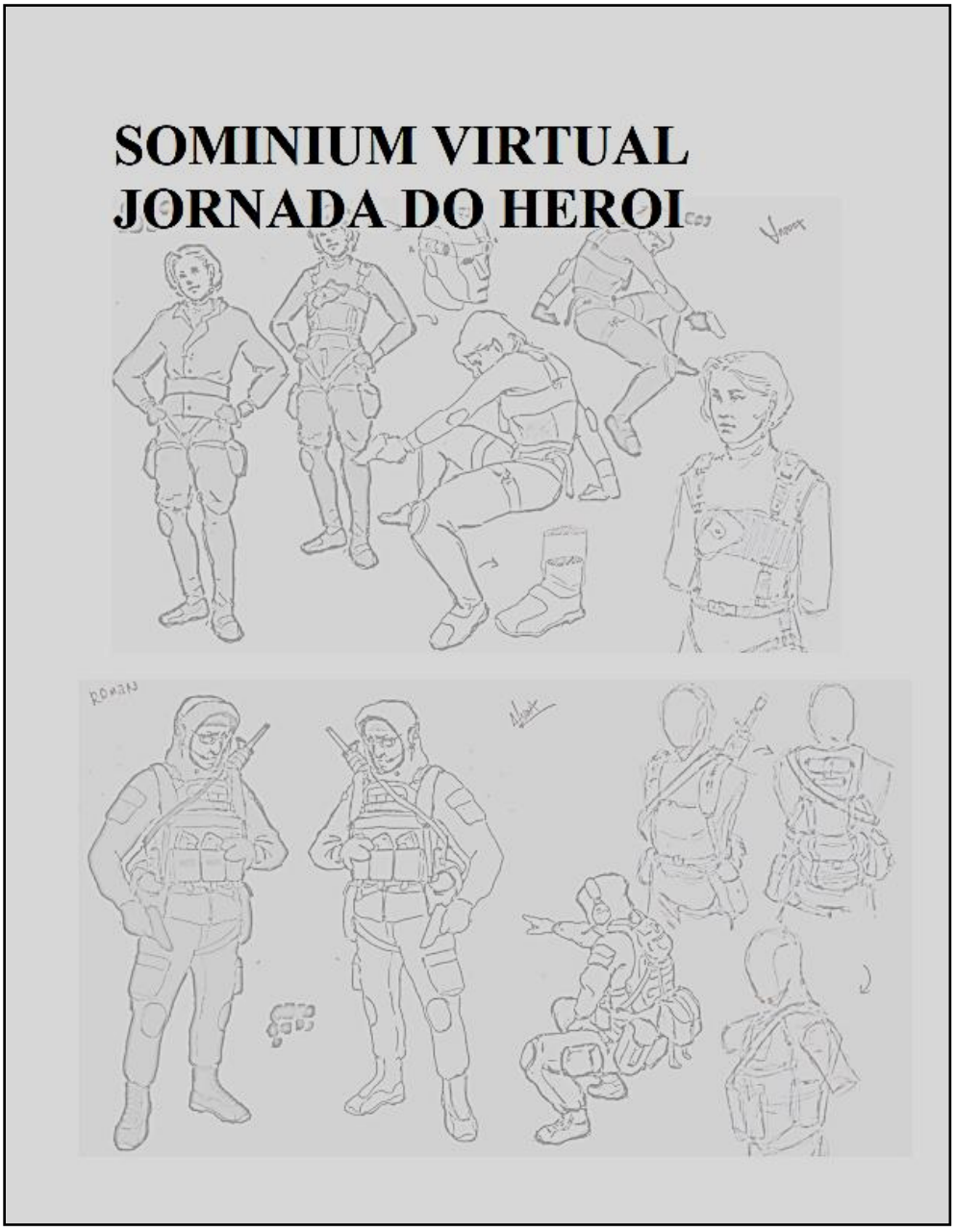
Fonte: Autoria Própria.

Por último, e não menos importante, outro documento auxiliar ao GDD é o documento de Jornada do herói[5] que auxilia na criação da narrativa de jogo.

[5] A jornada do herói é uma estratégia para narrar histórias, que conta com uma estrutura bem definida, além de recursos que ajudam a prender a atenção do espectador e gerar sentimentos diversos, como identificação e empatia (Martinson, 2021).

Adicionalmente, um documento simplificado de análise detalhada foi produzido, especificando os requisitos funcionais e não funcionais do Ato 1 delineado a partir do GDD. A Figura 8 mostra a capa de um dos artefatos gerados, o documento de Jornada de Herói.

Figura 8: Capa do documento de Jornada de Herói



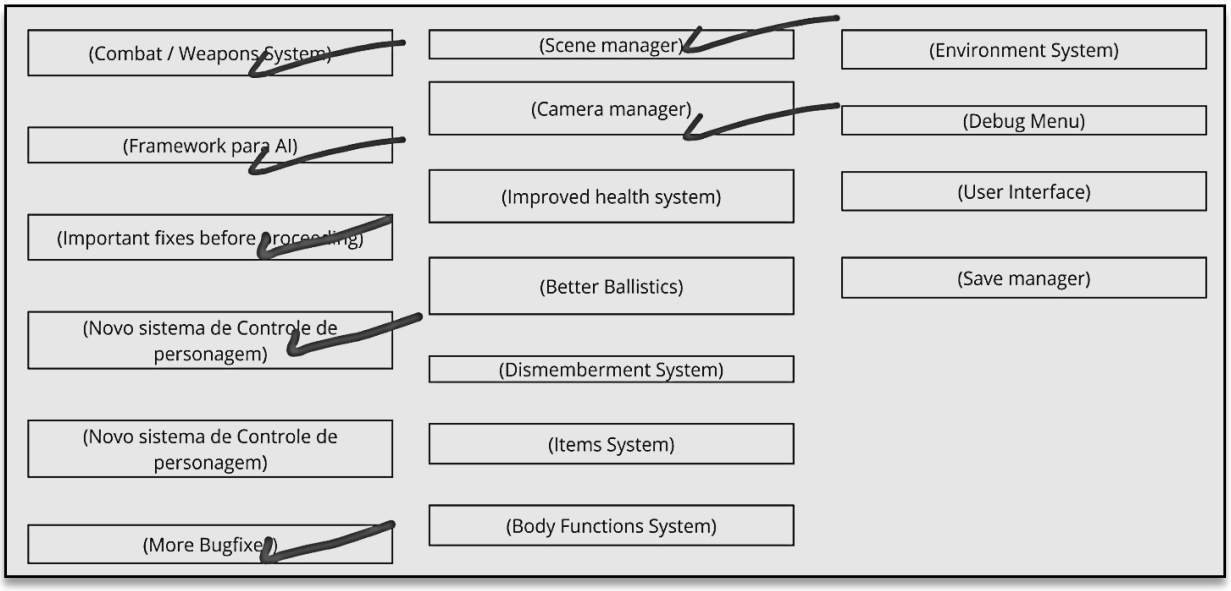
Fonte: Autoria Própria.

Com o GDD e os demais artefatos gerados foi possível apontar com precisão as tarefas à frente, separando-as em módulos de desenvolvimento, e suas correlações: *Character Controller, Weapon's Systems, Health System,* AI*, User Interface*, Menus, *Cutscenes*. Logo, a visão geral do escopo do projeto contida no

primeiro ato é possível assim como todos os seus componentes. Isto facilita o processo de planejamento e busca por ferramentas e soluções computacionais.

A Figura 9 apresenta a *Kaban Board* para o projeto, criada a partir dos artefatos gerados, especificando todos os componentes planejados.

Figura 9: *Kanban Board* para o *Project Cold Rain*.



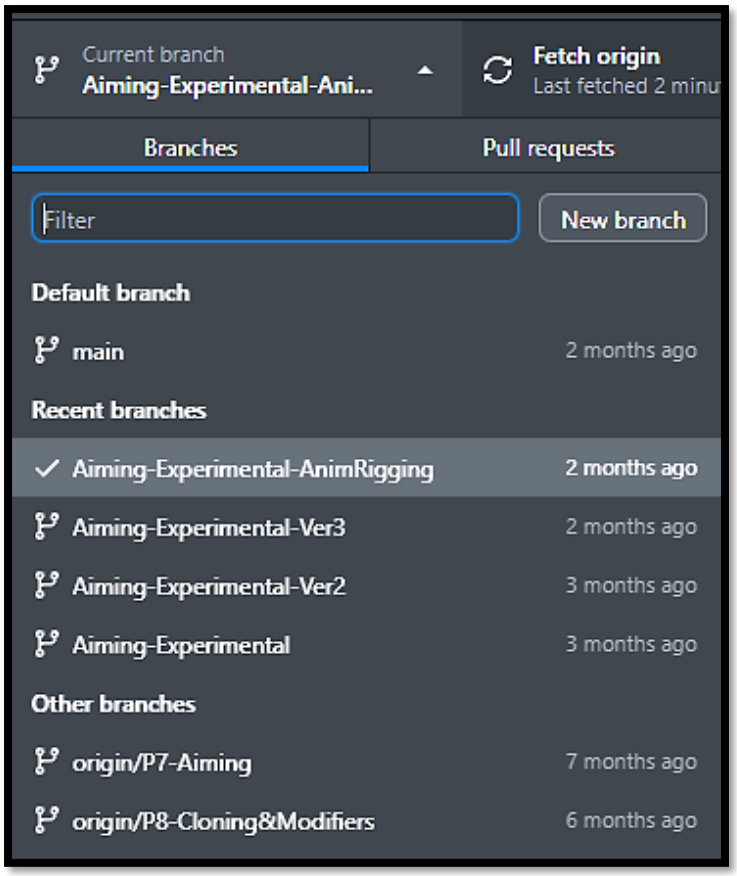
Fonte: Autoria Própria.

4.3 Ferramentas

A utilização do *GitHub*[6] como plataforma de controle de versões revelou-se uma escolha acertada para o gerenciamento do ciclo de vida do *Software*. Através do *GitHub*, foi possível não apenas hospedar e compartilhar o código-fonte do projeto, mas também adotar práticas de Integração Contínua (CI) e Entrega Contínua (CD), que são essenciais para a manutenção da qualidade e agilidade no desenvolvimento. A Figura 10 apresenta a atual configuração do projeto e sua divisão em várias *branches* para melhor gerenciamento dos ambientes de desenvolvimento.

[6] *GitHub* é uma plataforma de hospedagem de código-fonte e arquivos com controle de versão usando o *Git*. *GitHub* é amplamente utilizado por programadores para divulgação de seus trabalhos ou para que outros programadores contribuam com o projeto, além de promover fácil comunicação através de recursos que relatam problemas ou misturam repositórios remotos (*issues*, *pull request*) (Wikipedia, 2024).

Figura 10: *Branches* utilizadas em diferentes iterações



Fonte: Autoria Própria.

A estruturação do projeto em *branches* permitiu um gerenciamento eficaz das funcionalidades em desenvolvimento. *Commits* minuciosos acompanharam cada etapa, criando um registro detalhado e facilmente rastreável das alterações. Isto otimizou o processo de revisão de código. Por exemplo, sistemas críticos como o *Aiming System* foram desenvolvidos em *branches* dedicadas, isolando-as da versão principal do *software*. Essa abordagem assegurou que o ambiente de produção permanecesse estável e funcional, livre dos riscos associados a modificações significativas em outros subsistemas.

A escolha da *Unity*[7] como *game engine* para suporte ao desenvolvimento do projeto foi baseada em diversas características e vantagens oferecidas. A *Unity* é uma *engine* de fácil acesso e amplamente documentada, isso facilita o aprendizado e a

[7] ***Unity*** é *Game Engine* multiplataforma desenvolvido pela *Unity Technologies*, anunciado e lançado pela primeira vez em junho de 2005 na *Apple Worldwide Developers Conference* como um motor de jogo para Mac OS X. Desde então, foi gradualmente estendida para oferecer suporte a uma variedade de plataformas desde *desktop*, dispositivos móveis, consoles a realidade aumentada e realidade virtual. É particularmente popular para o desenvolvimento de jogos para dispositivos móveis iOS e *Android*, sendo considerada fácil de usar por desenvolvedores iniciantes e considerada popular para o desenvolvimento de jogos independentes. (Wikipedia, 2024)

resolução de problemas. Possui uma grande comunidade de desenvolvedores proporcionando um vasto suporte e troca de conhecimentos. A *Unity* também oferece planos acessíveis para desenvolvedores *indie*[8], tornando-a uma opção viável economicamente.

4.4 Módulos

Neste subcapítulo, serão explorados alguns dos sistemas basilares do projeto desenvolvidos considerando os requisitos técnicos e de *design* extraídos dos documentos de GDD e Alta Visão.

4.4.1 Ambiente de Desenvolvimento

A necessidade de um ambiente de desenvolvimento para um jogo de videogame se mostra de extrema importância, a partir dele é possível a realização de testes para garantia de qualidade dos módulos durante desenvolvimento. A Figura 11, mostra o cenário de testes construído para testes de implantação e integração com a *engine* para este projeto.

Figura 11: Cenário, para testes de áudio, colisão e efeitos especiais

Fonte: Autoria Própria.

[8] *Indie* é a abreviação de ***independent***, e *indie games* se refere àqueles jogos que são desenvolvidos de forma independente das grandes empresas de publicação e dos principais estúdios de desenvolvimento de jogos. Não necessariamente um jogo *indie* é desenvolvido por apenas uma pessoa. (AlfaMidia, 2019)

Este ambiente, visualizado na Figura 11, permite teste e ajuste de variadas mecânicas de jogo, como por exemplo, os sons de passos, os impactos de tiros em diferentes tipos de superfícies e outros sistemas como, o sistema de locomoção (*Character Controller)*. Os sons de passos podem variar dependendo do tipo de superfície em que o personagem está caminhando, seja grama, concreto ou madeira. Da mesma forma, o impacto de tiros pode ter efeitos sonoros e visuais diferentes para superfícies metálicas, de vidro ou de concreto. O teste dos sistemas de locomoção, colisão e interação com o cenário são vitais para garantia da jogabilidade esperada, pois estes afetam diretamente como os jogadores se movem e interagem com o mundo do jogo. Logo, se mostra imprescindível um cenário de testes que abranja esta ampla gama de casos de usos para os diferentes componentes.

4.4.2 Character Controller

Character Controller em jogos de videogames são componentes que permitem controlar os movimentos e as interações de um personagem em um ambiente virtual de jogo. Eles gerenciam o andar, pular e outros meios de locomoção e colisão com o cenário virtual e objetos em cena.

Os documentos de Alta Visão e GDD descrevem os requisitos essenciais para o desenvolvimento do (*Character Controller)* utilizado pelo projeto. Este deve ser desenvolvido sob uma perspectiva de terceira pessoa, com ações do personagem projetadas para simular movimentos realistas. Isso inclui a incorporação de elementos como inércia, exigindo, portanto, de um sistema de física avançado que opere em harmonia com o *Controller*.

O documento GDD enfatizou a criação de uma experiência de combate dinâmica e fluída, onde o personagem é capaz de executar ações simultâneas de ataque e evasão. Isso visa capturar a atenção do jogador e aumentar sua imersão. O objetivo é que, ao enfrentar hordas de inimigos, oponentes poderosos ou ambientes hostis, o jogador sinta o combate como algo crível e envolvente, contribuindo para uma narrativa interativa mais autêntica e imersiva. Isso requisita um *design* cuidadoso do sistema de animação, a fim de garantir que as transições entre movimento e ação sejam suaves e condizentes.

A animação do personagem também poderá ser controlada por *Inverse*

Kinematics (IK)[9], permitindo movimentos mais naturais e adaptativos em resposta ao ambiente e interações. A tecnologia IK, altera e ajusta a postura e a orientação do personagem em tempo real dinamicamente para refletir aspectos relacionados ao contato com terrenos, ações ou mesmo objetos, e favorece a implementação do sistema de mira em inimigos *Aiming System*, que é responsável pela mudança de alvo e ajuste de mira em diferentes partes do corpo.

A escolha do *Unity Starter Assets - Third Person Character Controller* foi motivada pela necessidade de prototipagem rápida. Embora não seja uma decisão definitiva, optou-se por essa solução da *Unity* devido à sua adequação às exigências iniciais do projeto. A criação de um *Character Controller* customizado do zero não é justificada no momento, uma vez que outras mecânicas demandam atenção e desenvolvimento. A implementação da *Unity* já oferece uma solução eficiente e satisfatória, permitindo focar recursos nas áreas que ainda precisam ser desenvolvidas.

4.4.3 Sistema de combate

Neste subcapítulo, serão expostos alguns aspectos que compõem o sistema de combate desenvolvido considerando os requisitos técnicos e de design extraídos dos documentos de GDD e Alta Visão.

Variedade de armas - o jogador, imerso nesse universo, contará com uma ampla gama de armas para enfrentar inimigos. A variedade de armamentos favorece a jogabilidade e melhora o aspecto de rejogabilidade do projeto.

Aprimoramento de armas - os jogadores poderão melhorar suas armas à medida que avançam. A evolução de armamentos tem como intuito o aprofundamento da mecânica de combate e aumento da rejogabilidade, ambos resultando em maior atratividade aos jogadores.

Horror e Sobrevivência - jogadores devem sentir tensão e medo ao enfrentarem os inimigos. A fidelidade ao tema de horror e sobrevivência é crucial. Cada elemento do jogo, desde a arte até a jogabilidade, deve reforçar essa atmosfera.

Controles acessíveis, desafios estratégicos - os controles devem ser intuitivos, permitindo que os jogadores se concentrem na estratégia. A dificuldade não

[9] Em animação computacional e robótica, *Inverse Kinematics* é o processo matemático de calcular os parâmetros articulares variáveis necessários para colocar o final de uma cadeia cinemática, como um manipulador de robô ou o esqueleto de um personagem de animação, em uma determinada posição e orientação em relação ao início da cadeia. (Wikipedia, 2024)

reside na complexidade dos comandos, mas sim nos encontros com inimigos e na gestão cuidadosa de recursos escassos, como munição e vida. A simplicidade dos controles contrasta com a complexidade das decisões táticas.

Variedade de inimigos e simplicidade - o projeto deverá contar com ampla variedade de inimigos. Estes em sua maioria não operarão com um sistema de Inteligência artificial complexo, e sim, simples ações unitárias ou pequenas cadeias de ações. Cada encontro se dará como único a partir da combinação estratégica de inimigos e sua quantidade em cena, isto ditará dificuldade de cada batalha. Inimigos especiais denominados Chefes contarão com ações mais complexas.

Sistema de mira - o sistema de mira será automático e não manual. Esta decisão visa garantir a incerteza quanto à precisão da efetividade das armas durante combate, tornando os inimigos menos previsíveis ou evitando que o programador apresente inimigos difíceis de eliminar, a fim de não perder o aspecto desafiador do jogo. Outras questões centram-se em propiciar a acessibilidade, pois facilita a experiência para jogadores amadores, cujas habilidades com jogos de tiros possam ser limitadas, promovendo a imersão do jogador que escolherá se concentrar mais nas estratégias e narrativas, livrando o jogador da frustração causada por sistema de mira manual.

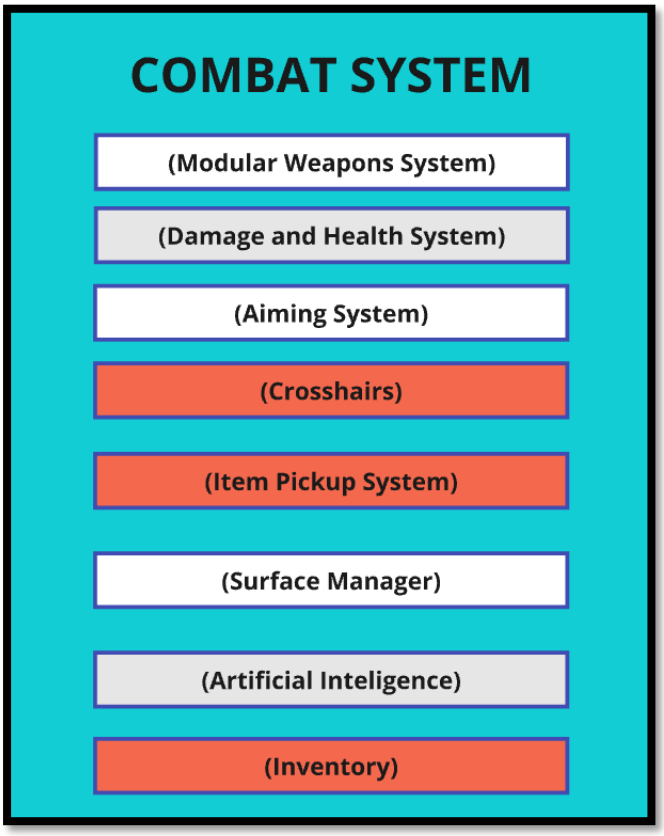
O sistema de combate emprega também a possibilidade de andar e atirar simultaneamente adicionando dinamismo e realismo durante o combate, permitindo ao jogador reagir às ameaças de maneira dinâmica e eficaz. Portanto, a não restrição de ações como, a fuga simultaneamente ao recarregamento de armamentos e uso de itens evitam que os inimigos possam ter vantagem no jogo por serem desproporcionalmente mais fortes que o jogador.

Limitação de armamento (*Swap System*) - com recursos limitados, os jogadores precisam tomar decisões estratégicas. Armas poderão ser encontradas ao longo do jogo obtidas através de outros inimigos ou encontradas em diferentes cenários. O jogador é limitado a carregar somente dois tipos de arma de fogo consigo, armas longas e curtas. Armas longas apresentarão clara superioridade de poder de fogo, porém com munição escassa, já armas curtas contarão com grande quantidade de munição e poder de fogo moderado. Por isso, a existência de um sistema de troca rápida (*Swap System*) se prova necessária, permitindo o jogador a adaptar-se aos diferentes cenários de combate.

Com os requisitos documentados a elaboração de um plano de implementação

e esboço da solução tecnológica se torna possível. A Figura 12 apresenta esboço dos componentes que compõem o sistema de combate e em vermelho, componentes indiretamente relacionados a este sistema.

Figura 12: Modelagem para o Sistema de Combate



Fonte: Autoria Própria.

4.4.4 Modular Weapons System (Sistema de armamento modular)

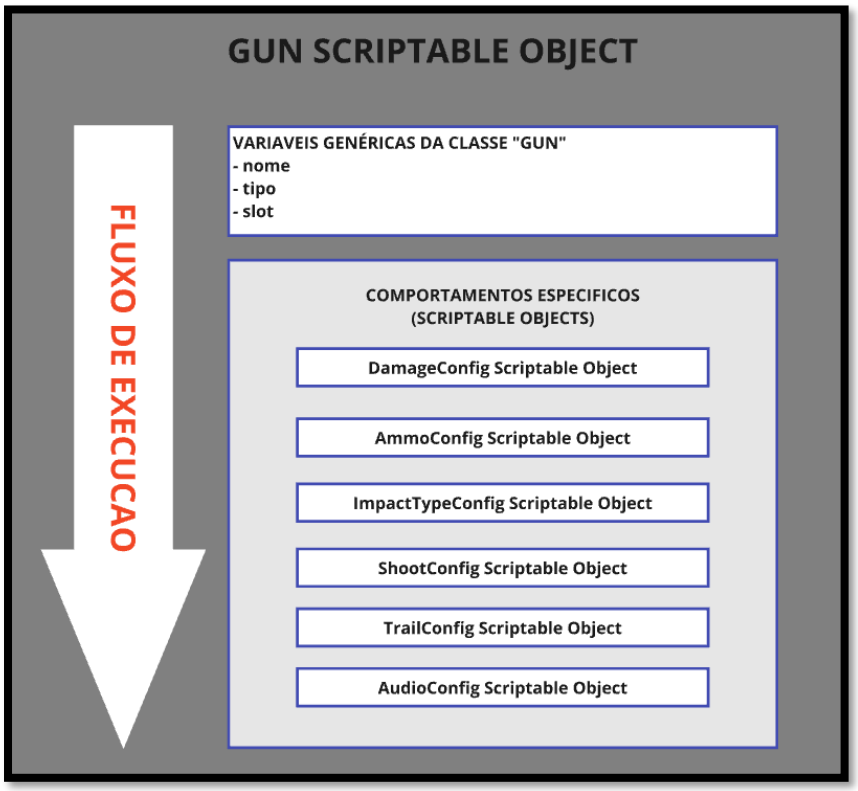
A busca pela solução computacional inicia-se pelo componente (*Modular Weapons System*), com a adoção de uma base modular para criação de todo armamento do jogo, privilegiando a facilidade na escalabilidade e personalização de características de cada armamento.

O uso de *Scriptable Objects*[10] como técnica programacional para o alcance de um sistema modular para os armamentos (*Modular Weapons System*), é justificado uma vez que são candidatos perfeitos ao permitirem a segmentação e modularização do código em partes menores e especializadas, hierarquizando o sistema de um script "pai" para scripts "filhos" que executarão tarefas delegadas pelo script "pai". A Figura

[10] Um *ScriptableObject* é um contêiner de dados que você pode usar para salvar grandes quantidades de dados, independentemente de instâncias de classe. Um dos principais casos de uso de *ScriptableObjects* é reduzir o uso de memória do seu projeto, evitando cópias de valores. (Unity, 2022)

13 mostra o esboço organizacional da classe *GunScriptableObject*.

Figura 13: *GunScriptableObject* e seus componentes genéricos e especializados



Fonte: Autoria Própria.

Foi determinado que um armamento é composto pelos seguintes atributos:

- **Componentes gerais - *Name*** (nome do armamento);
- ***Type*** (tipo/modelo da arma);
- ***SlotCategory*** (categoria do slot que ocupa no inventário primário para armas longas ou secundário para armas curtas);
- ***ModelPrefab*** (modelo 3d que representa a arma, assim como outros componentes necessários para outros subsistemas, sendo eles, objetos que delimitam a posição da mão do personagem ao empunhar a arma e sistema de partículas que deve ser tocado ao disparar a arma);
- ***MagazineModel*** (modelo do cartucho 3d utilizado no momento de recarga);
- ***Animator***[11] (utilizado para sobrescrever animações genéricas como por exemplo animações de recarga do armamento);
- ***SpawnPoint*** e ***SpawnRotation*** (utilizados para controle de empunhadura,

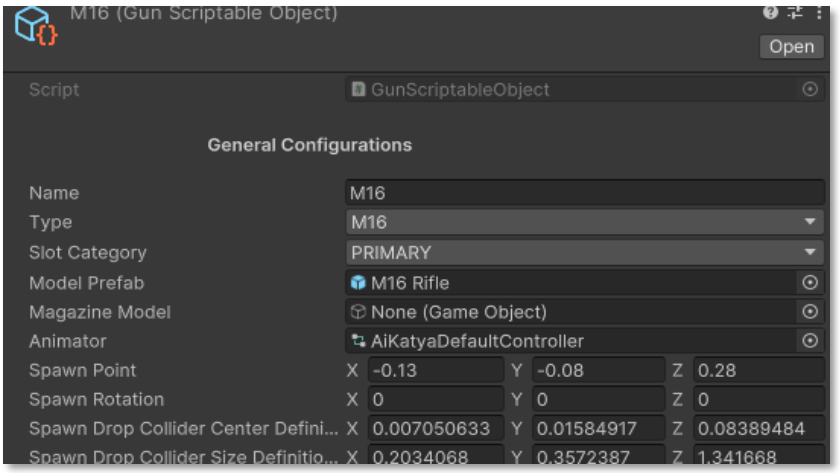
[11] O componente *Animator* é usado para atribuir animação a um *GameObject* em sua cena (Unity, 2022)

determina a posição e rotação de instanciação da arma relativo ao seu parente, jogador);

- ***SpawnDropColliderCenterDefinitions*** e ***SpawnDropColliderSizeDefinitions*** (são utilizados no momento de morte do personagem, no qual são instanciados colisores ao modelo 3d da arma, permitindo o sistema de física interno da *Unity* exercer forças como gravidade sobre o objeto);

A Figura 14 apresenta o campo de configurações gerais para um *GunScriptableObject* já preenchidos para uma das armas já implementadas, M16.

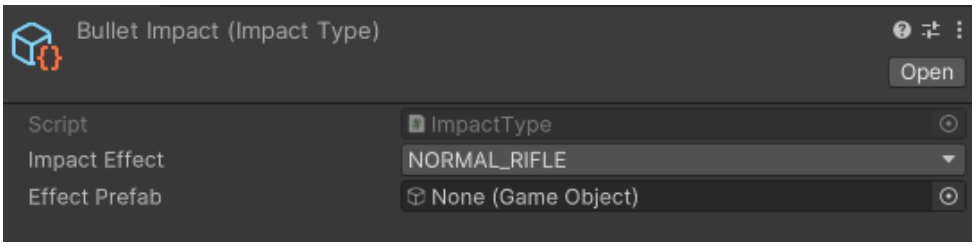
Figura 14: Configurações gerais para *GunScriptableObject* M16.



Fonte: Autoria Própria.

- **ScriptableObjects: *ImpactType*** (aponta o tipo de impacto a ser processado pelo subsistema *SurfaceManager,* gerenciador de superfícies); A Figura 15 mostra o *ScriptableObject ImpactType* para o *GunScriptableObject* M16.

Figura 15: Configurações de *ImpactType* para *GunScriptableObject* M16.

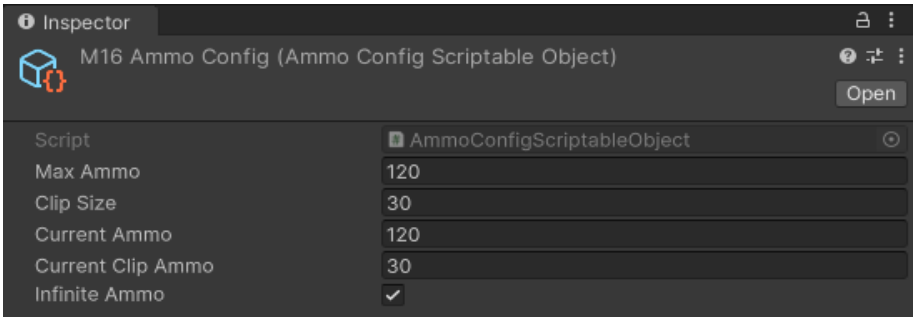


Fonte: Autoria Própria.

- ***AmmoConfig*** (segura valores determinando a quantidade de munição que a arma suporta, utilizado para armas que não necessitam de cartuchos para a recarga, como por exemplo, escopetas que não necessitam de cartucho para

recarga de munição, ou mesmo a Inteligência Artificial, que utiliza deste método de recarga por não necessitar de cartuchos); A Figura 16 mostra o *ScriptableObject AmmoConfig* para o *GunScriptableObject* M16.

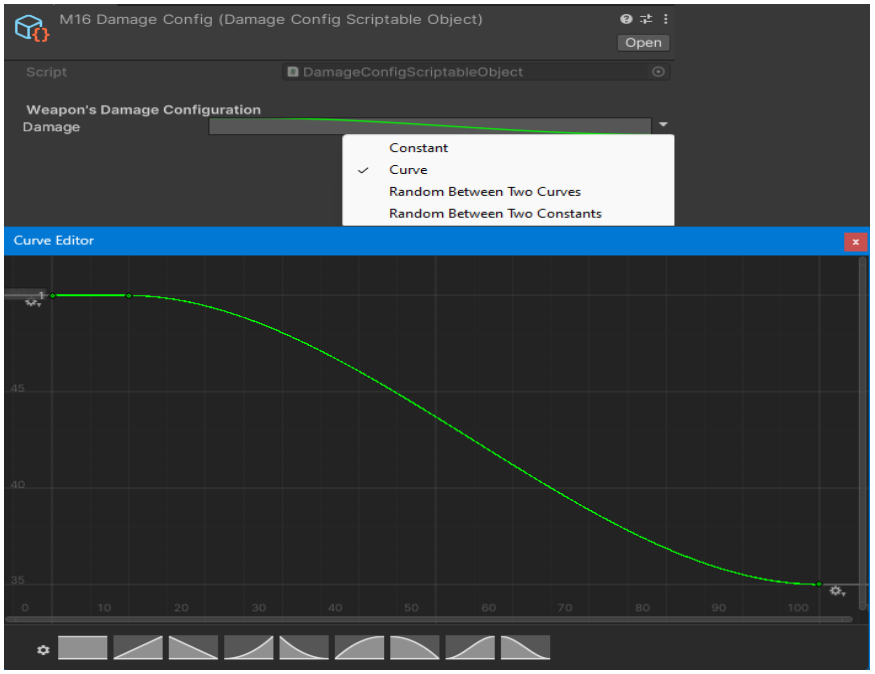
Figura 16: Configurações de *AmmoConfig* para *GunScriptableObject* M16.



Fonte: Autoria Própria.

- ***DamageConfig*** (*Scriptable Object* que armazena toda a característica de dano infligido pela arma, altamente configurável podendo ser constante, linear, exponencial, aleatório entre duas curvas, ou aleatório entre duas constantes); A Figura 17 mostra o *ScriptableObject DamageConfig* para o *GunScriptableObject* M16.

Figura 17: Configurações de *DamageConfig* para *GunScriptableObject* M16.

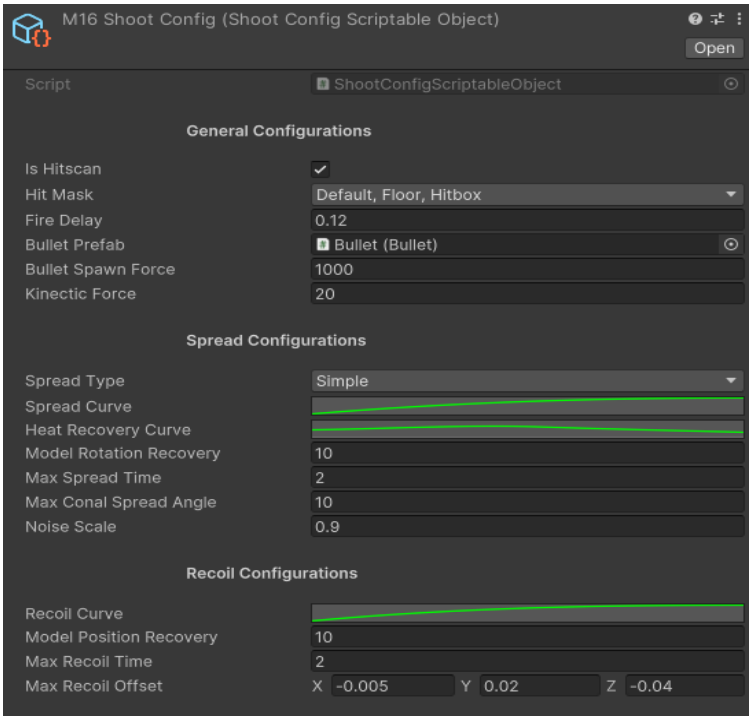


Fonte: Autoria Própria.

- ***ShootConfig*** (armazena configurações de dispersão de disparo, coice, e variáveis que ditam o comportamento do projétil *hitscan* ou físico, cadência de tiro, força cinética do projétil ao atingir o alvo e a camada (*LayerMask*) com que

o projétil colidirá); A Figura 18 mostra o *ScriptableObject ShootConfig* para o *GunScriptableObject* M16.

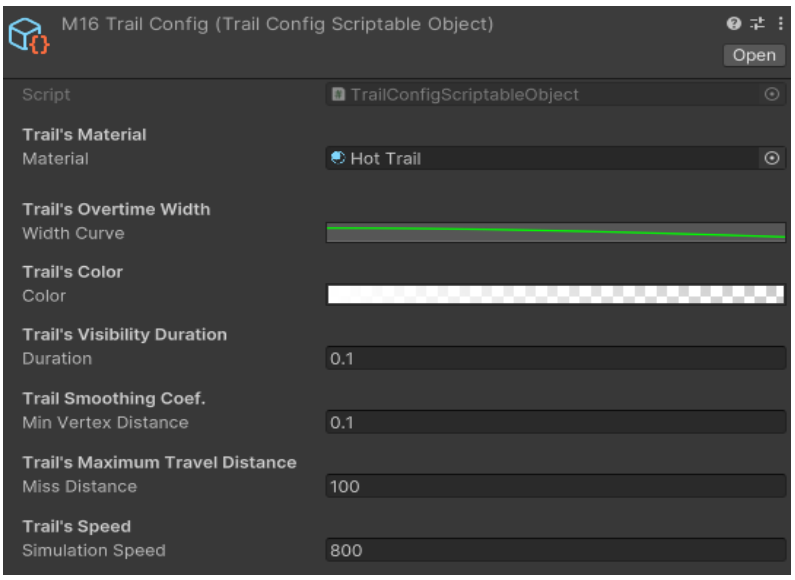
Figura 18: Configurações de *ShootConfig* para *GunScriptableObject* M16.



Fonte: Autoria Própria.

- ***TrailConfig*** (este *ScriptableObject* armazena as configurações de traçado da arma como por exemplo sua cor, duração, tipo de efeito, tamanho e velocidade); A Figura 19 mostra o *ScriptableObject TrailConfig* para o *GunScriptableObject* M16.

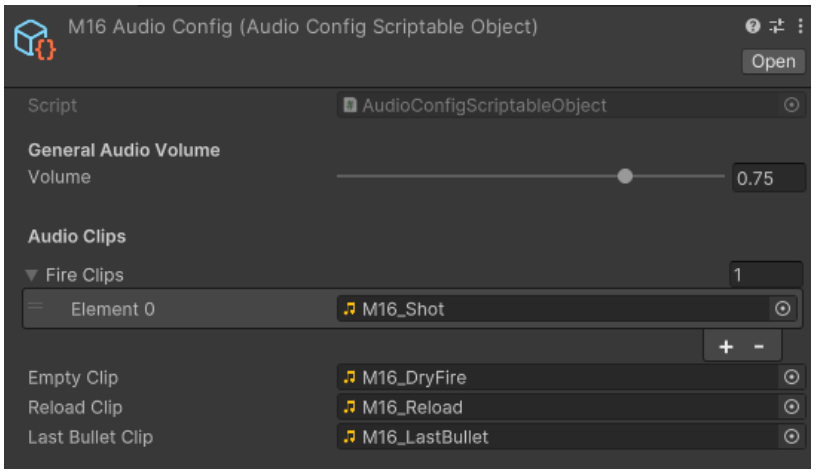
Figura 19: Configurações de *TrailConfig* para *GunScriptableObject* M16.



Fonte: Autoria Própria.

- ***AudioConfig*** (este objeto armazena todos os sons de funcionamento do armamento, sons de disparo, sons de recarregamento etc.); A Figura 20 mostra o *ScriptableObject AudioConfig* para o *GunScriptableObject* M16.

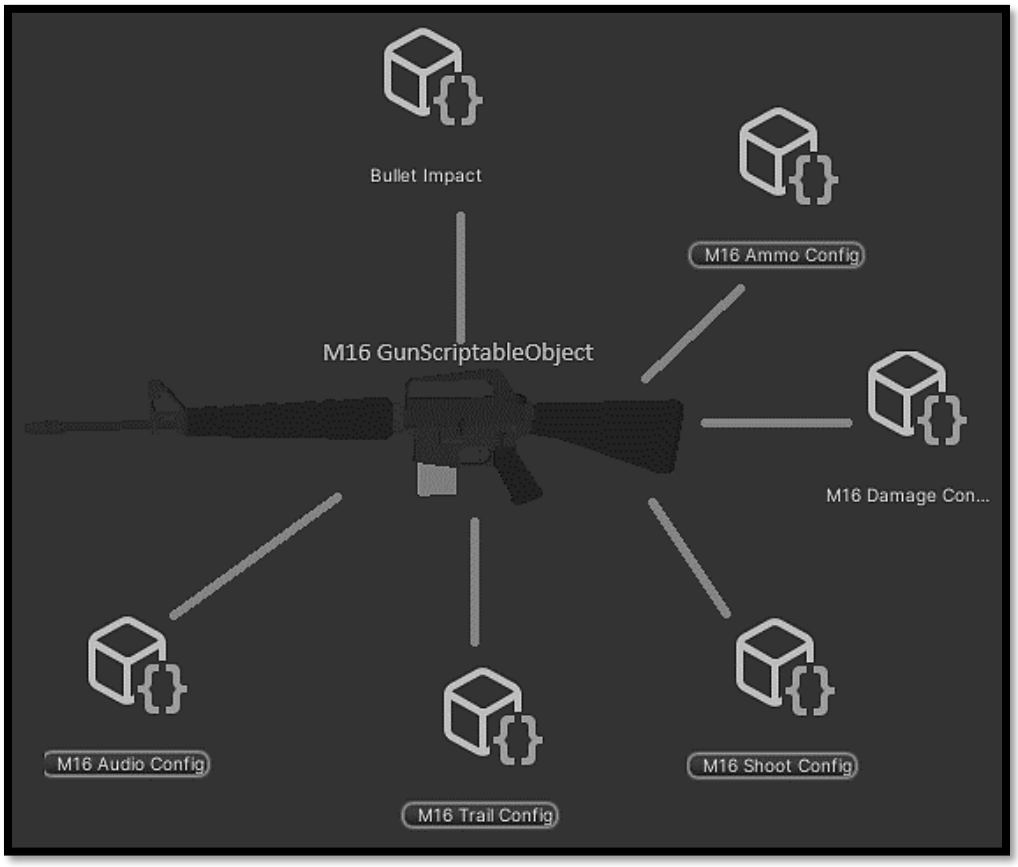
Figura 20: Configurações de *AudioConfig* para *GunScriptableObject* M16.



Fonte: Autoria Própria.

A Figura 21 apresenta de maneira visual a composição de uma arma de fogo a partir de todos os *ScriptableObjects*.

Figura 21: Todos os componentes que formam uma arma de fogo.

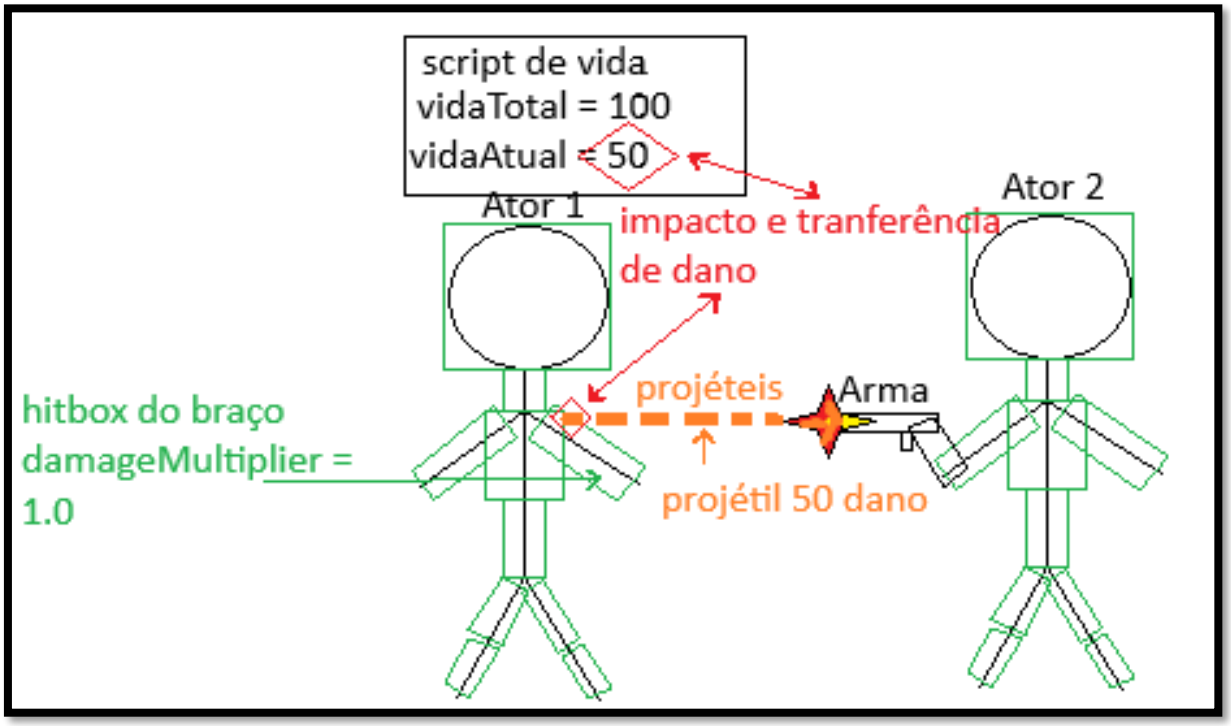


Fonte: Autoria Própria.

4.4.5 Damage and Health System (Sistema de saúde)

O subcomponente *Damage and Health System* provou-se um dos componentes razoavelmente simples e rápidos de modelagem e implementação. Este componente utiliza de *hitboxes*[12] em cada parte do corpo do personagem, tanto para o jogador quanto para os agentes de inteligência artificial (IA) ou mesmo objetos inanimados. A abordagem do problema partindo do uso de *hitboxes* permite a fácil expansibilidade para outros subsistemas como: a configuração da quantidade de pontos alvos que um determinado personagem oferece, configuração de como o dano a cada ponto alvo deverá refletir em sua vida global, ou mesmo atributos especiais como invulnerabilidades ou resistências em um a determinada parte do corpo. A Figura 22 demonstra a modelagem inicial para o sistema de cálculo de dano.

Figura 22: Representação do sistema de dano e *hitboxes*.



Fonte: Autoria Própria.

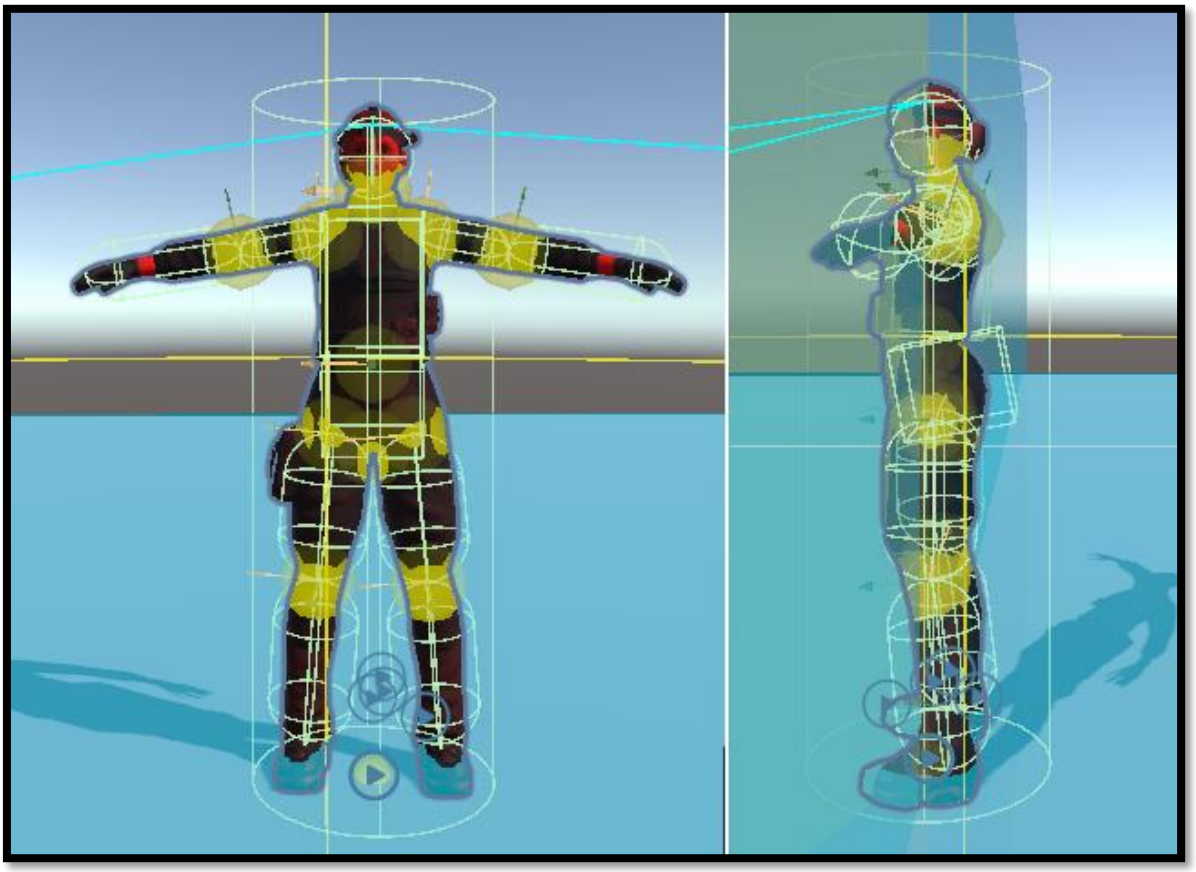
As *hitboxes* verificam colisões com projéteis, que são detectados via *raycast*[13] ou colisões físicas partidas de projéteis físicos transferindo o dano para um *script* de saúde global vinculado ao personagem. O script base, chamado *Health*, gerencia a

[12] *Hitboxes* são as geometrias invisíveis que informam a um jogo quando e como as coisas colidem. (Wiltshire, 2020)
[13] Um *raycast* é conceitualmente como um feixe de laser que é disparado de um ponto no espaço ao longo de uma direção específica. Qualquer objeto que entre em contato com o feixe pode ser detectado e relatado. (Unity, 2024)

saúde máxima e a saúde atual de um personagem ou objeto e pode ser estendida para versões específicas, como ***PlayerHealth***, ***AiHealth*** ou mesmo ***ObjectHealth***. Cada *hitbox* também aplica um multiplicador de dano configurável, a distribuição dos valores para o multiplicador de dano fica a critério do desenvolvedor. Por padrão o multiplicador aplicado a todas *hitboxes* é de 1.0, ou seja, não interferem no dano causado pela arma transferindo o dano original para o sistema de saúde central. Além disso, as *hitboxes* são essenciais para o funcionamento do sistema de *ragdoll* (boneca de pano). Este sistema é ativado quando ocorrer a morte de um personagem humanoide, sendo ele o jogador ou inteligência artificial. Este sistema faz com que as partes do corpo interajam fisicamente com o ambiente após a morte de agente, simulando um corpo inerte.

A Figura 23 mostra o arranjo atual do sistema de saúde vinculado a um agente de Inteligência Artificial com propósito de testes.

Figura 23: *Hitboxes* para um agente de teste.



Fonte: Autoria Própria.

Embora simples, este sistema agrega bastante ao realismo do jogo, devido a maneira com que os impactos e interações entre projéteis e superfície são

processados.

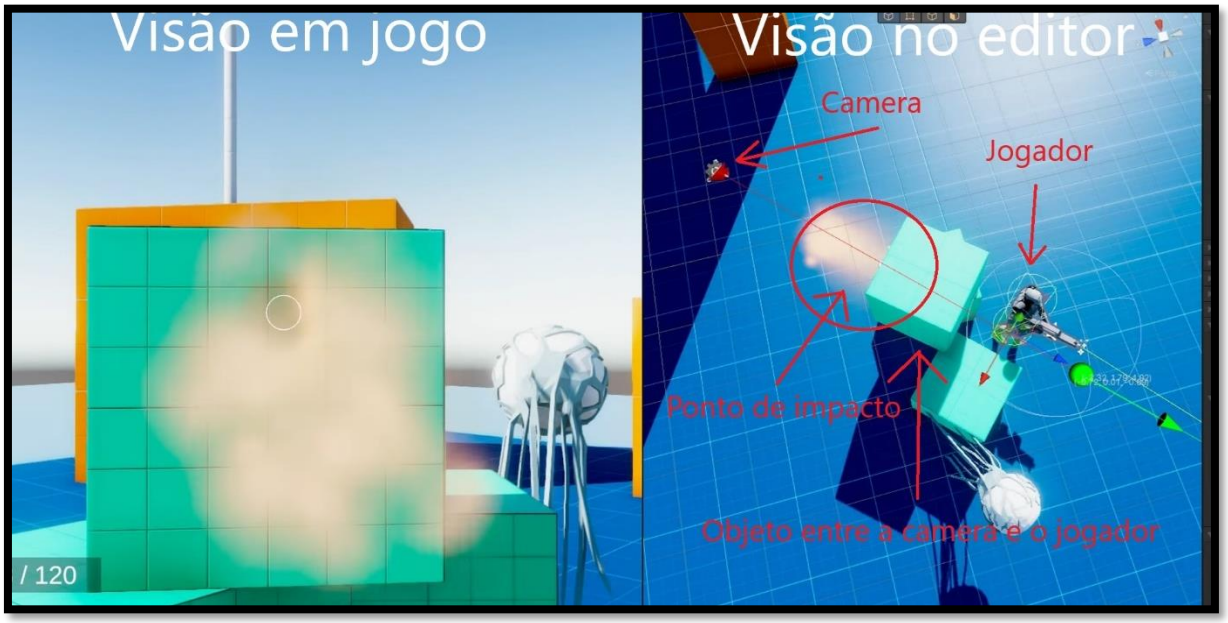
4.4.6 Crosshair System (Sistema de ponto de Mira)

O desenvolvimento de um sistema de ponto de mira ou retículo (*Crosshair System*) para *games* apresenta diversos desafios técnicos que devem ser cuidadosamente considerados.

Criar uma reticulo eficaz para jogos de tiro em terceira pessoa envolve superar vários desafios singulares. Isso inclui lidar com o deslocamento e a perspectiva da câmera, garantia da trajetória precisa do projetil, a detecção de acerto, gerência do campo de visão e das mudanças visuais causadas por fatores externos tais como o recuo e o movimento do armamento em situações de jogo.

Deslocamento e perspectiva da câmera: em jogos de tiro em terceira pessoa, a câmera normalmente é posicionada atrás e acima do personagem. Esse deslocamento pode dificultar o alinhamento da mira com o ponto real de impacto, especialmente ao mirar em alvos a distâncias diferentes. A seguir se demonstra como o deslocamento da câmera pode resultar no erro de lógica em sistemas onde o cálculo de trajetória depende da câmera. Este erro causa o deslocamento do ponto de impacto da arma com relação ao ponto de mira fazendo com que o projetil acerte um objeto posicionado atrás do jogador, pois neste sistema o a instanciação do projetil parte da câmera e nenhuma verificação adicional é feita. A Figura 24 apresenta o erro introduzido pela dependência da câmera para cálculo de trajetória do projetil.

Figura 24: lógica de trajetória.



Fonte: Chris Kurhan LlamAcademy

Primeiramente, a precisão do projetil em relação ao posicionamento do retículo (*crosshair)* é indispensável, uma vez que deve refletir com exatidão o ponto de impacto do projetil da arma correlacionado com a intenção do jogador e escolha de alvo. Outro desafio técnico significativo é a escolha do comportamento do retículo com o ambiente do jogo e suas regras por meio de retículos estáticos ou dinâmicos.

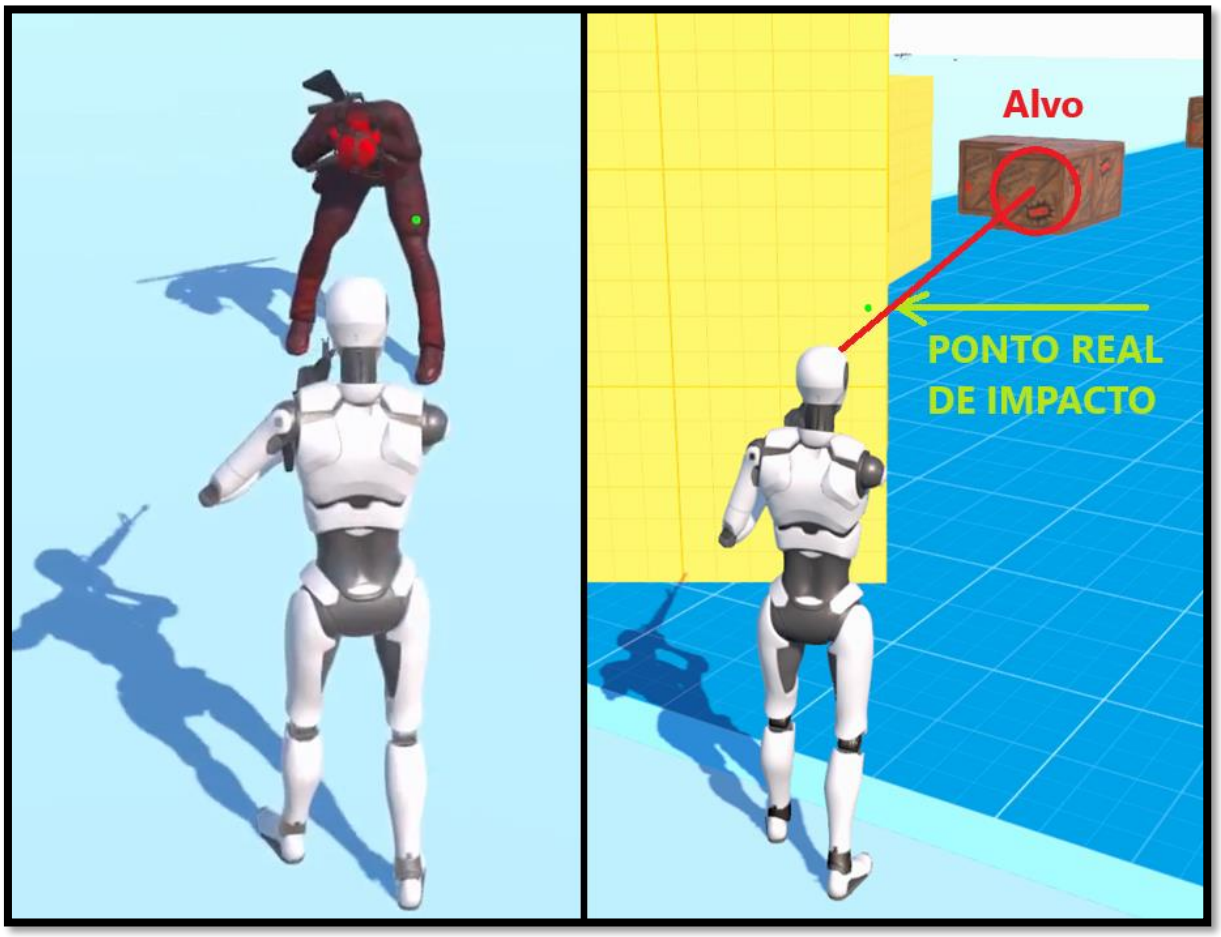
Um retículo dinâmico muda sua aparência com base nas ações do jogador e no ambiente do jogo. Por exemplo: o retículo pode se expandir quando o jogador está se movendo para indicar a perda de precisão ou dispersão ao efetuar disparos nesta condição. Outra característica é a mudança de forma ou cor durante certas ações demonstrando características momentâneas como o recuo e, ou, a dispersão de disparo do armamento.

Como vantagem para retículos dinâmicos tem-se: *feedback* em tempo real sobre precisão e comportamento da arma fornecendo aos jogadores informações importantes que auxiliam na tomada de decisões durante combate. Desvantagens: retículos dinâmicos podem ser uma distração se as mudanças forem muito frequentes ou drásticas. Devido ao aspecto mutável deste estilo a sua implementação pode exigir mais poder de processamento, afetando potencialmente o desempenho do jogo.

Um retículo estático permanece constante independentemente das ações do jogador ou do ambiente do jogo. Ele não muda de tamanho, forma ou cor ao se mover ou atirar. Como vantagens para este estilo de retículos tem-se: consistência e previsibilidade, o que torna mais fácil a adaptação por parte dos jogadores. Retículos estáticos são menos distrativos, logo permitem que os jogadores se concentrem outras características durante o jogo. A principal desvantagem deste estilo de retículo e o não *feedback* sobre o comportamento da arma ou condições que influenciam e atuam sobre as mecânicas de combate.

No que tange às considerações de *design*, a visibilidade do *crosshair* é um aspecto fundamental. Ela deve ser claramente visível em diversos cenários e condições de iluminação, sem se tornar intrusiva para o jogador. A Figura 25 mostra o sistema em funcionamento.

Figura 25: Demonstração em tempo real do sistema de mira.



Fonte: Autoria própria

Durante o processo de modelagem dos sistemas de combate ficou evidente a dependência por parte do sistema de ponto de mira *(Crosshair System)* de uma solução que culminasse em uma representação fiel das condições de jogo. Devido a estas exigências optou-se pelo desenvolvimento de um sistema dinâmico minimalista, no qual a *crosshair* seja de característica visual não intrusiva e distrativa, porém perpassando informações importantes ao jogador.

4.4.7 Artificial intelligence System (Sistema de Inteligência Artificial)

A Inteligência Artificial (AI) desempenha um papel crucial no desenvolvimento de sistemas de combate em jogos, permitindo que os inimigos ou aliados controlados pelo computador (NPCs[14]) interajam com o jogador e cenário de maneira autônoma.

[14] A sigla vem da expressão: *Non-Playable-Character*. Ou seja: Personagem não jogável. Os NPCs são aqueles personagens colocados nos jogos para ambientação, dos quais não conseguimos controlar. É isso o que é NPC. São os "figurantes" que às vezes até interagem, mas que suas ações são controladas e colocadas pelo mecanismo do *game*. (Montovani, 2020)

Em um sistema de combate moderno, a AI vai além de movimentos ou comportamentos pré-programados, aplicando algoritmos avançados que permitem que os NPCs se adaptem ao ambiente, ao comportamento do jogador e às estratégias emergentes. Esses sistemas podem variar em complexidade, desde simples decisões baseadas em condições, como atacar ou se defender, até arquiteturas mais elaboradas que envolvem aprendizado de máquina e análise situacional em tempo real por meio de sensores.

A AI utilizada no projeto foi implementada por meio de um sistema simples de máquina de estados finitos (FSM), com foco na simplicidade e eficiência. A filosofia de *design* adotada prioriza a criação de inimigos com comportamentos simples, ou seja, inimigos comuns denominados “*mobs*” contarão com ações unitárias ou curtas cadeias de ações. Adicionalmente, o sistema permite a criação de inimigos especiais, como "Chefes", que apresentam estados e comportamentos mais complexos com tomada de decisões utilizando sensores. Como definido pelos artefatos de documentação o desafio do jogo emerge da combinação de diferentes tipos de inimigos no cenário.

O controle da AI é realizado por um script central pertencente a um agente de AI que gerência os diferentes estados atribuídos a ele. Cada estado é representado por um *script* individual que segue a lógica básica de uma máquina de estados. A seguir são listados os componentes e etapas que compõem o sistema desenvolvido:

- **Máquina de Estados (*AiStateMachine*)**

O *script AiStateMachine* gerencia o que o agente de AI faz em um determinado momento, mudando entre diferentes "estados" (como "perseguir o jogador" ou "ficar ocioso"). A máquina de estados mantém um registro dos possíveis estados em um *array* de estados (*states*[]) e muda entre eles com base em eventos ou condições.

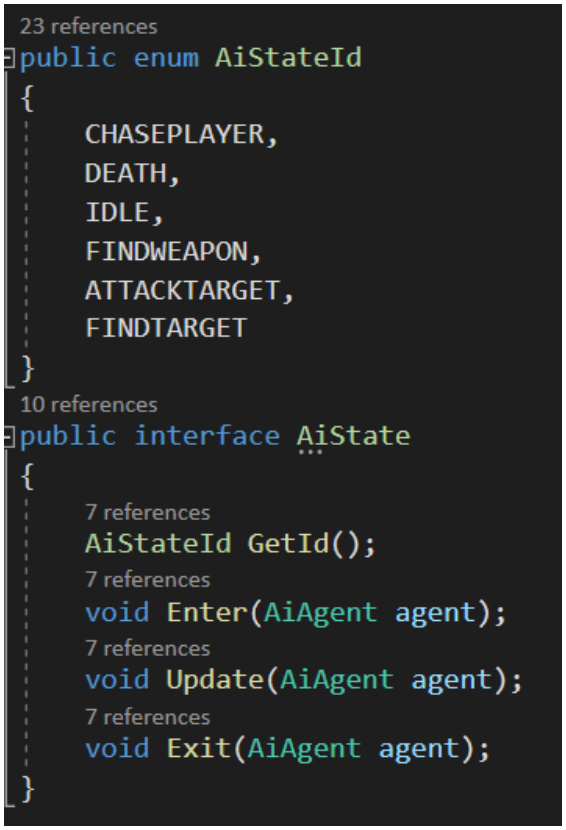
- **Estados (*AiState* e *AiStateId*)**

Os estados são definidos como uma interface (*AiState*) com três métodos principais: As funções definidas pela interface são implementadas em classes concretas que definem o comportamento do agente no momento de entrada do estado, durante sua permanência neste estado, ou na sua saída.

  - *Enter*(): chamado quando o agente entra em um novo estado;
  - *Update*(): chamado repetidamente a cada quadro, enquanto o agente está em um estado específico;
  - *Exit*(): chamado quando o agente sai do estado.

Os diferentes estados são identificados por um enum (*AiStateId*), que contém valores como *CHASEPLAYER*, *DEATH*, *IDLE* etc. Cada valor corresponde a um estado específico. A Figura 26 mostra a estrutura base dos estados que alimentam a FSM.

Figura 26: Interface matriz para criação de estados.



Fonte: Autoria própria

- **Agente de AI (*AiAgent*)**

O *AiAgent* é o componente principal que controla o personagem ou a entidade de AI. Ele contém a máquina de estados (*AiStateMachine*) e vários outros componentes, como o sistema de navegação (*NavMeshAgent*), armas, sensores etc. No método *Start*(), ele inicializa esses componentes e registra os estados que a AI poderá assumir. Por exemplo, o estado de "morte" (*AiDeathState*) e "ficar ocioso" (*AiIdleState*) são registrados na máquina de estados.

- **Mudança de Estado**

A máquina de estados permite que o agente mude de estado. Isso é feito no método *ChangeState*(), onde:

  - O estado atual executa seu método *Exit*() (para limpar qualquer coisa necessária ao sair do estado);
  - O estado novo executa seu método *Enter*() (para configurar o novo

comportamento).

- **Atualização dos Estados**

A cada frame (no método *Update*()), a máquina de estados chama o método *Update*() do estado atual, permitindo que o agente execute a lógica daquele estado. Por exemplo, se o estado atual for *CHASEPLAYER*, o agente executará a lógica para perseguir o jogador. A Figura 27 mostra as o registro das ações para a máquina de estados de um agente de teste.

Figura 27: Registro de estados para um agente de teste.

```
Unity Message | 0 references
void Start()
{
    ragdoll = GetComponent<Ragdoll>();
    navMeshAgent = GetComponent<NavMeshAgent>();
    weapons = GetComponent<AiWeapons>();
    sensor = GetComponent<Sensor>();
    awarenessSensor = GetComponent<AwarenessSensor>();
    targetingSystem = GetComponent<TargetingSystem>();

    stateMachine = new AiStateMachine(this);

    stateMachine.RegisterState(new AiDeathState());
    stateMachine.RegisterState(new AiIdleState());
    stateMachine.RegisterState(new AiFindWeaponState());
    stateMachine.RegisterState(new AiAttackTargetState());
    stateMachine.RegisterState(new AiFindTargetState());
    stateMachine.ChangeState(initialState);
}

Unity Message | 0 references
void Update()
{
    stateMachine.Update();
}
```

Fonte: Autoria própria

Assim, a máquina de estados permite gerenciar de forma organizada o comportamento do agente de AI, separando a lógica de cada comportamento específico em diferentes estados.

4.4.8 Aiming System (Sistema de mira em inimigos e alvos)

Os sistemas de mira em jogos variam para atender diferentes estilos de jogabilidade e plataformas. Sistemas de *free aim* ou mira livre/manual oferecem ao

jogador controle total sobre a mira, exigindo precisão e reflexos. Este tipo de sistema é comumente usado em jogos de tiro (em primeira pessoa FPS's ou terceira pessoa TPS's), onde a habilidade com o *mouse* e teclado é essencial.

Sistemas com *aim assist* ou assistência de mira, são utilizados em grande parte por *consoles*. Estes estilos focam em suavizar o movimento da mira quando o alvo está próximo ao retículo controlado pelo jogador. Em seu núcleo, estes tipos de sistema também continuam sendo de mira livre, porém com esta mínima assistência facilitando a jogabilidade.

Já sistemas de *lock-on* ou travamento, permitem ao jogador a escolha e travamento da mira em um inimigo específico em cena, facilitando o combate em jogos de ação ou RPG's, no qual o foco está mais na estratégia do que na precisão absoluta.

Além desses, alguns jogos utilizam sistemas *auto-aim*, que ajustam a mira automaticamente para alvos dentro de uma certa área delineada (normalmente a área de sensores). Este é considerado o sistema mais tolerante de todos, permitindo que o jogador selecione apenas algumas decisões de combate como: mudar a parte do corpo que deseja mirar ou mudar de alvo.

Cada sistema visa balancear o desafio e a imersão, levando em conta as necessidades específicas do estilo de jogo e a plataforma em que está sendo jogado. Cada estilo de sistema de mira afeta diretamente o balanceamento de inimigos e as mecânicas de jogo.

Em **sistemas de mira livre com ou sem aim assist**, a precisão depende das habilidades do jogador, isso pode tornar o combate desafiador em alguns aspectos ou facilmente explorável em outros. Em jogos de terror, este tipo de sistema pode comprometer o equilíbrio do jogo, pois os jogadores podem facilmente abusar das áreas vulneráveis dos inimigos, como a cabeça, tornando o combate fácil e menos assustador. Para compensar isso, desenvolvedores muitas vezes se veem obrigados a aumentar desproporcionalmente a saúde ou mobilidade dos inimigos, prejudicando outros aspectos como a tensão do jogo. Como resultado, obtém-se combates prolongados que quebram a imersão ou situações que deixam o jogador com um sentimento barato.

Por outro lado, sistemas como ***aim-assist*** facilitam o combate, permitindo que os inimigos sejam mais rápidos ou imprevisíveis, já que a mira ajuda a suavizar o desafio. Isso cria um equilíbrio entre manter a dificuldade sem sobrecarregar o jogador com a necessidade de uma alta habilidade mecânica. No entanto, em jogos

competitivos, a assistência de mira pode ser vista como uma vantagem injusta, especialmente entre jogadores de diferentes plataformas (como console vs. PC).

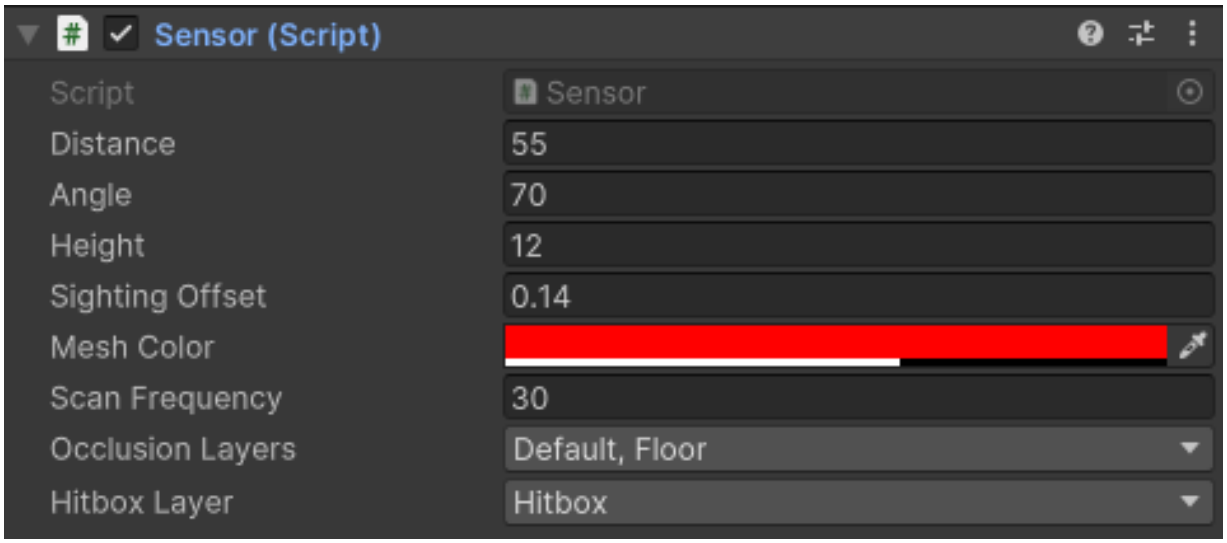
Em sistemas mais automatizados, como o ***auto-aim*** e ***lock-on***, o equilíbrio se desloca ainda mais para o *design* do inimigo, o combate se torna menos dependente da habilidade mecânica do jogador e mais em sua habilidade estratégica. Isso permite que os desenvolvedores criem inimigos mais complexos ou em maior número, ou utilizem métodos mais criativos de balanceamento para alcançar um equilíbrio entre a dificuldade e a jogabilidade. No entanto, isso pode reduzir a satisfação dos jogadores que buscam desafios mais técnicos, pois metade do desafio passa a ser de controle do sistema.

Para este projeto o sistema de mira *Aiming System* adotou um estilo de *auto-aim* sendo composto por quatro componentes principais utilizados tanto pelo jogador quando pelo agente de AI:

- **Sistema de Sensores (Sensor)**:

A função principal do sensor é verificar objetos que estão dentro do campo de visão e acessíveis (sem obstruções) para o agente. Isso permite ao sistema identificar possíveis alvos. A Figura 28 mostra a configuração dos parâmetros do sensor para o jogador.

Figura 28: Parâmetros para o sensor do jogador.



Fonte: Autoria própria

Os principais componentes do sensor incluem variáveis como: *distance*, que define o alcance máximo da detecção do sensor, e *angle*, que define o ângulo de visão em torno do personagem. A *height* determina a altura do sensor em relação ao chão, enquanto o *sightingOffset* ajusta a altura do ponto de origem do *raycast* para

verificação de linha de visão (LoS) de objetos detectados. A frequência das varreduras do sensor não acontece a cada quadro visando otimização logo passa ser controlada por *scanFrequency* que dita a frequência da atualização do sensor. As camadas de objetos a serem detectadas e as que podem obstruir a visão do sensor são definidas por *hitboxLayer e occlusionLayers* respectivamente.

A definição de uma lista chamada *objects* visa o armazenamento de todos os objetos atualmente detectados pelo sensor, estes *objects* então podem ser acessados por outras funções como as utilizadas pelo sistema de *targeting.*

O sensor também conta as seguintes funções:

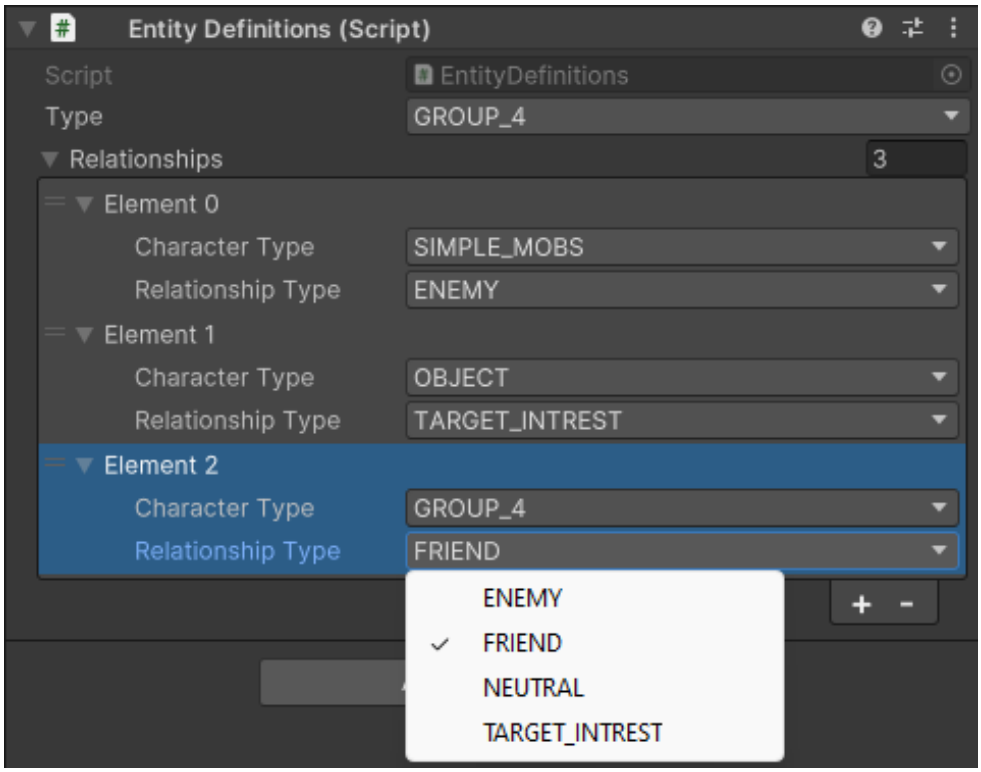
- A função *Start*() inicializa o intervalo entre as varreduras do sensor baseado na frequência definida.
- A função *Update*() verifica se o tempo desde a última varredura ultrapassou o intervalo e, se sim, chama o método *Scan*() para detectar objetos.
- *Scan*() executa a varredura na cena para encontrar objetos dentro do alcance do sensor, utilizando o método *Physics*.*OverlapSphereNonAlloc*() para detectar colisores ao redor do personagem em uma área esférica, baseando-se na camada *hitboxLayer*. Um *HashSet* é utilizado para garantir que o sensor adicione apenas objetos únicos à lista de detecção, se faz a necessidade pois sem esta garantia objetos com múltiplas *hitboxes* são adicionados várias vezes a lista de objetos.
- *FindRelevantObject*(*Collider collider*) é chamada para identificar o objeto de interesse, buscando o componente *EntityDefinitions* no objeto identificado.
- *IsInSight*() determina se o objeto está dentro do campo de visão, considerando tanto a posição quanto possíveis obstruções. Isso é feito verificando o ângulo entre a direção do sensor e o objeto e realizando uma linha de teste (*Linecast*) entre o sensor e o objeto para verificar se algum obstáculo impede a trajetória do raio.
- *Filter*() filtra a lista de objetos detectados para identificar alvos específicos com base no tipo de relacionamento entre o personagem e o objeto detectado (inimigo, aliado etc.), determinado por meio do componente *EntityDefinitions* previamente obtido no momento de detecção do objeto por parte do sensor.

Em resumo, o sensor periodicamente escaneia ao redor do personagem para detectar objetos ou entidades dentro de um raio específico. Ele utiliza um campo de visão, definido por um ângulo, distância, e uma altura, para determinar se os objetos estão visíveis e acessíveis ao personagem. O sistema filtra esses objetos com base em seu relacionamento com o personagem (inimigos, alvos, aliados), assim determinando os alvos candidatos.

- **Sistema de Relacionamentos (*EntityDefinitions* e *Relationships*)**:

O Sistema de Relacionamentos complementa essa dinâmica, categorizando as entidades do jogo e suas interações. Com ele, o sistema é capaz de diferenciar inimigos, aliados e objetos neutros, fornecendo uma base para a tomada de decisões tanto da AI quanto do jogador. O uso de *enums* e classes especializadas permite que essa lógica seja organizada de forma eficiente, oferecendo flexibilidade para ajustes futuros, como a inclusão de novos tipos de entidades ou mudanças no comportamento das relações entre elas. A Figura 29 mostra a definição dos relacionamentos por parte do jogador.

Figura 29: Definição de relacionamentos para o jogador.



Fonte: Autoria própria

Esta categorização é essencial, pois serve como base para decisões da AI e do funcionamento do sensor, como determinar se um personagem é um inimigo ou um aliado por parte do jogador ou AI.

O sistema inclui *enums* e classes como principais componentes. O enum *CharacterType* define os tipos de personagens ou objetos no jogo, sendo seus valores provisórios para testes internos: *NOT_DEFINED* (tipo padrão), *GROUP_4* (grupo dos quatro protagonistas), *SIMPLE_MOBS* (inimigos controlados por AI simples) e *OBJECT* (objetos destrutíveis, como caixas). O *enum RelationshipType* especifica os tipos de relação entre entidades: *ENEMY* (inimigo), *FRIEND* (aliado), *NEUTRAL* (neutro) e *TARGET_INTEREST* (entidades de interesse, como objetos).

A classe *Relationship* representa uma relação entre um *CharacterType* e um *RelationshipType*. A classe *EntityDefinitions* contém um campo *type*, que define o tipo da entidade com base em *CharacterType* e uma lista *relationships* que armazena as relações dessa entidade com outras.

A também uma função denominada *GetRelationshipWith*(*CharacterType otherType*) que realiza a consulta do relacionamento da entidade detentora do script com outro tipo de entidade. Ao detectar outra entidade, a função avalia como ela deve ser tratada e, se for um inimigo ou alvo de interesse, a informação é registrada, caso contrário, é ignorada. Ela percorre a lista de relacionamentos e retorna o tipo correspondente, como *ENEMY* ou *FRIEND*. Se não houver uma relação específica, retorna *NEUTRAL*. Por exemplo, a entidade *GROUP*_4 pode ver *SIMPLE*_MOBS como inimigos.

- **Memória Sensorial (*SensoryMemory*)**:

Outro componente chave é a Memória Sensorial, que oferece ao agente a capacidade de "lembrar" de alvos previamente detectados, mesmo que saiam de seu campo de visão. Esse sistema de memória torna as interações da AI mais realistas, permitindo que ela continue a perseguir ou reagir a entidades com base em encontros anteriores. A memória gerencia informações como a última posição e o tempo em que o alvo foi visto, atualizando esses dados conforme novos eventos ocorrem.

A classe *Memory* armazena detalhes sobre um objeto ou entidade detectada pela AI. Cada instância de *Memory* corresponde a uma entidade percebida e inclui vários atributos:

- *Age*, que calcula o tempo desde que a entidade foi vista pela última vez;
- *gameObject*, que é o objeto percebido;
- *position* e *direction*, que são a posição e direção da entidade em relação ao agente;

- *distance* e *angle*, que são a distância e o ângulo da entidade em relação ao agente;
- *lastSeen*, que é o tempo da última vez que o agente viu a entidade;
- score, um valor que pode ser usado para priorizar memórias;
- *defaultAimPoint* e *aimPoints*, que são posições no objeto onde o agente pode focar sua mira.

A classe *SensoryMemory* gerencia as memórias de todas as entidades detectadas por um agente específico. Ela contém uma lista memories, que armazena informações sobre todas as entidades percebidas pelo agente, e um *array characters*, representando os personagens que podem ser percebidos. Esta classe e composta pelas seguintes funções:

A função *UpdateSenses*(*Sensor sensor*) atualiza as memórias sensoriais do agente com base nos alvos detectados pelo sensor. Ela utiliza o método *Filter*() do sensor para obter os alvos detectados e, para cada alvo, chama *RefreshMemory*() para atualizar ou criar uma nova memória associada ao alvo detectado. A função *RefreshMemory*(*GameObject agent, GameObject target*) atualiza as informações sobre um alvo no sistema de memória sensorial. Se o alvo já existe na memória, ela atualiza a posição, direção, distância e o tempo em que foi visto pela última vez, além de popular a lista de pontos de mira (*aimPoints*) e definir um ponto de mira padrão (*defaultAimPoint*).

A função *FetchMemory*(*GameObject gameObject*) recupera a memória existente de um determinado objeto. Se o objeto não estiver na memória, cria uma instância de *Memory*. Ela procura na lista de memórias uma memória associada ao objeto e, se não encontrar, cria uma memória e a adiciona à lista.

A função *ForgetMemories*(*float olderThan*) remove memórias de entidades que não foram vistas por muito tempo ou que não existem mais no jogo. Ela remove memórias mais antigas que o tempo especificado em *olderThan* e memórias associadas a objetos que já não existem.

Em resumo, este arranjo fornece uma maneira de armazenar e gerenciar memórias sensoriais, permitindo que a AI reaja de maneira mais sofisticada com base em interações passadas. Ele se integra com o sistema de sensores, garantindo um método mais avançado de tratamento para os objetos detectados pelo sensor.

- **Sistema de Mira (*TargetingSystem*)**:

Por fim, o Sistema de Mira integra todos esses elementos ao avaliar e priorizar alvos com base em uma série de critérios, como proximidade e ângulo. Através desse sistema, tanto o jogador quanto a AI pode focar em alvos de forma dinâmica, com a capacidade de alternar entre diferentes partes de um alvo ou mesmo entre múltiplos alvos. A avaliação de alvos ocorre em tempo real, considerando as memórias sensoriais e as condições do ambiente para definir o alvo mais adequado para cada situação.

Os principais componentes do sistema incluem atributos e propriedades. Entre os atributos principais, *memorySpan* define o tempo de duração da memória antes que seja esquecida, enquanto *distanceWeight* e *angleWeight* determinam a influência da distância e do ângulo ao calcular a prioridade de um alvo. A variável *currentAimPointIndex* é o índice do ponto de mira atual dentro do alvo, útil para selecionar diferentes partes do corpo ou outros pontos relevantes.

As propriedades do sistema incluem *HasTarget*, que verifica se há um alvo ativo, e *HasMemoryStored*, que verifica se há memórias armazenadas. A propriedade Target retorna o *GameObject* do alvo selecionado, enquanto *TargetInSight* verifica se o alvo foi visto recentemente (dentro de 0,5 segundos). *TargetDistance* retorna a distância do alvo atual, *MemoryQuantity* retorna a quantidade de memórias armazenadas, *TargetPosition* retorna a posição mais recente do alvo, e *GetDefaultAimPoint* retorna o ponto de mira padrão do alvo.

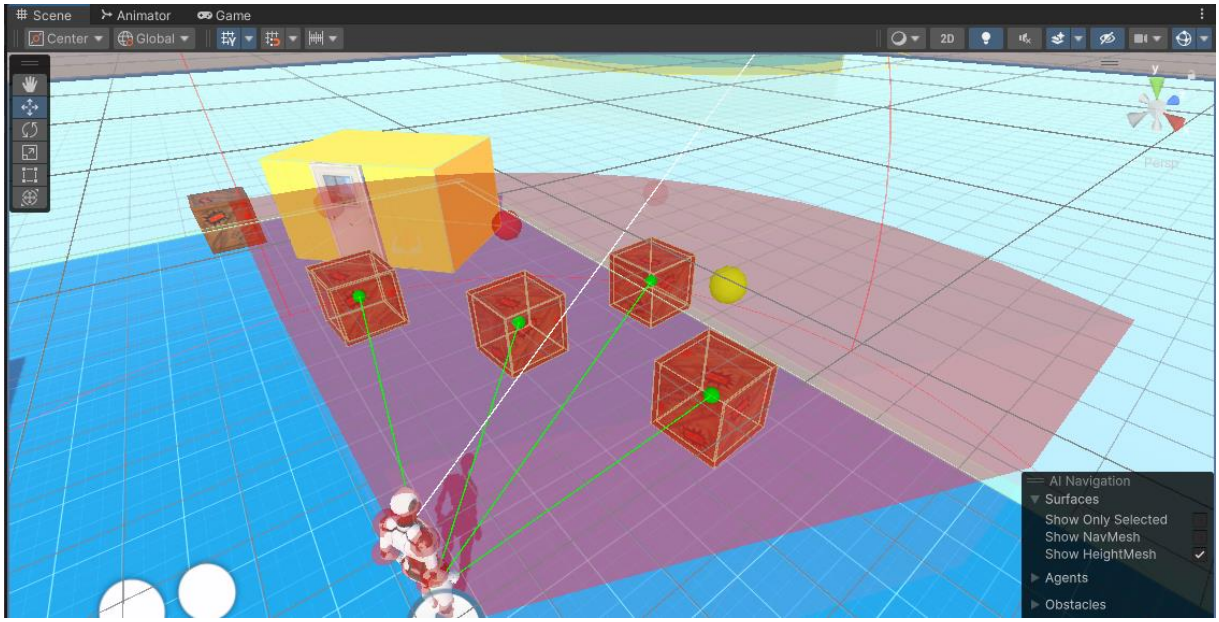
As funções principais do sistema incluem a atualização dos sentidos da AI e a seleção de pontos de mira. A função *Update*() atualiza os sentidos da IA verificando se há novos alvos e esquecendo aqueles cujas memórias estão fora do *memorySpan*. Ela também avalia os alvos armazenados e escolhe o melhor baseado em um sistema de pontuação. As funções *NextAimPoint*() e *PreviousAimPoint*() permitem que a AI mude entre diferentes pontos de mira, como cabeça e torso, utilizando o *currentAimPointIndex* para alternar entre esses pontos. A função *getTargetPoint*() retorna a posição de um ponto de mira específico ou, se não houver, a posição geral do alvo.

A pontuação e avaliação de alvos são realizadas pela função *EvaluateScores*(), que avalia as memórias armazenadas e calcula uma pontuação para cada uma com base na distância e no ângulo. Ela seleciona a memória com a maior pontuação como o melhor alvo, armazenado em *bestMemory*. A função *CalculateScore*() calcula a

pontuação de um alvo normalizando a distância e o ângulo com base nos valores máximos permitidos pelo sensor e aplicando os pesos definidos.

Para visualização, a função *OnDrawGizmos*() desenha esferas nas posições dos alvos no editor da Unity, com cores que indicam sua pontuação e importância, destacando o alvo atual em amarelo. O mecanismo de escolha de alvos do algoritmo baseia-se na combinação de distância e ângulo, dando prioridade àqueles que estão mais próximos e mais diretamente na linha de visão do agente. Ele avalia todas as memórias atuais, atribui uma pontuação a cada uma e escolhe o alvo com a maior pontuação como o “melhor alvo”. A Figura 30 mostra todo o sistema de Mira em funcionamento, detectando objetos considerados como *TARGET_INTREST*. É possível notar a área do Sensor, as linhas de verificação do campo de visão, e a classificação das memórias para cada objeto.

Figura 30: Sistema de mira em inimigos e suas funcionalidades



Fonte: Autoria Própria.

Em resumo, esse arranjo modular permite uma interação sofisticada e fluida entre os agentes AI o jogador e seu ambiente, combinando detecção sensorial, memória e mira para melhor alcance de eficiência do comportamento da AI e do jogador.

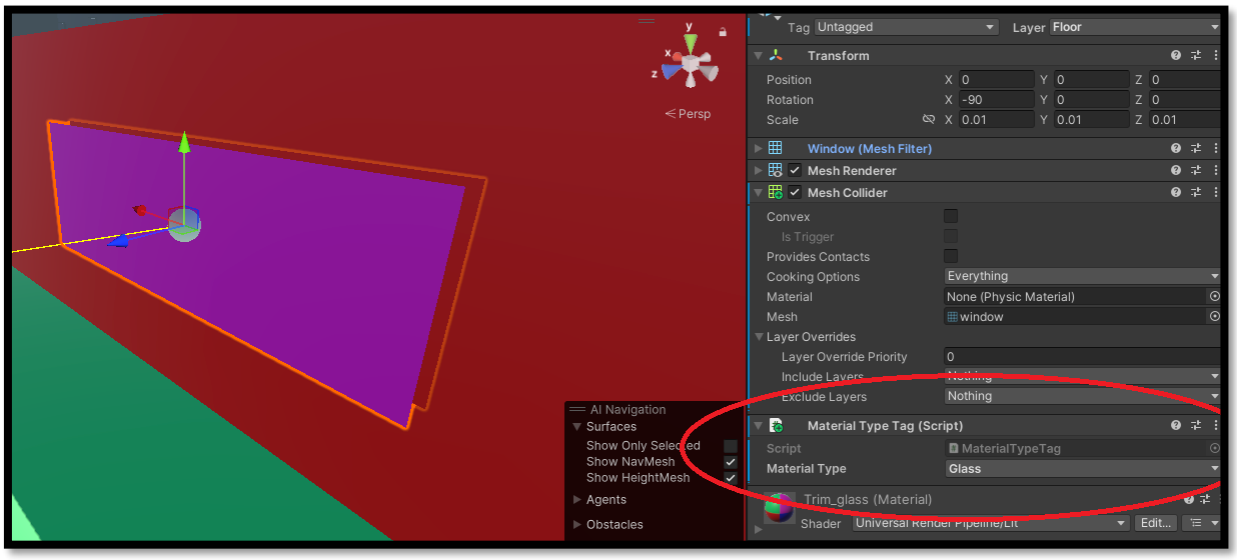
4.4.9 Surface Manager (Gerenciador de Superfícies)

O *SurfaceManager* gerencia os efeitos de impacto em diferentes superfícies dentro da cena do jogo. Ele utiliza o padrão *Singleton* para garantir que apenas uma instância do *SurfaceManager* exista na cena, prevenindo conflitos.

O principal objetivo do *SurfaceManager* é aplicar efeitos visuais e sonoros quando um objeto colide com uma superfície. Ele verifica o tipo de material da superfície atingida e, com base nisso, reproduz os efeitos apropriados. Se o material não for encontrado, ele usa uma superfície padrão. A instanciação e destruição de objetos sempre foi uma operação considerada cara em qualquer software, com isso em mente o *SurfaceManager* utiliza de *ObjectPools*[15] para gerenciar instâncias de objetos e fontes de áudio de forma eficiente, evitando a criação e destruição contínua de objetos.

Objetos em cena que utilizam de colisores são marcados com um script chamado *MaterialTypeTag* que segura um enum referenciando o tipo de material que este colisor será. A Figura 31 mostra a atribuição do script material *type tag* a uma janela.

Figura 31: Atribuição *MaterialTypeTag* "*Glass*" para janela.



Fonte: Autoria Própria.

Logo, é possível a identificação da superfície de contato do impacto, seja impacto de passos ou mesmo projeteis por meio da função *HandleImpact* declarada no algoritmo do gerenciador de superfície. A função *HandleImpact* verifica o tipo de material do objeto atingido e aplica os efeitos visuais e sonoros apropriados. Se o tipo de material não for encontrado ou o objeto não possuir um *MaterialTypeTag*, a função

[15] *Object Pooling* é uma maneira de otimizar seus projetos e reduzir a carga que é colocada na CPU ao criar e destruir rapidamente novos objetos. É uma boa prática e um padrão de *design* para ajudar a aliviar o poder de processamento da CPU para lidar com tarefas mais importantes e não ser inundado por chamadas repetitivas de criação e destruição [...] isso é particularmente útil ao lidar com balas [...] (Unity, 2019)

utiliza os efeitos da superfície padrão. A Figura 32 mostra a simples estrutura da função *HandleImpact*.

Figura 32: Função *HandleImpact*() em *SurfaceManager*.cs.

```
1 reference
public void HandleImpact(GameObject HitObject, Vector3 HitPoint, Vector3 HitNormal, ImpactType Impact)
{
    MaterialTypeTag tag = HitObject.GetComponent<MaterialTypeTag>();
    if (tag != null)
    {
        SurfaceType surfaceType = Surfaces.Find(surface => surface.MaterialType == tag.MaterialType);
        if (surfaceType != null)
        {
            foreach (Surface.SurfaceImpactTypeEffect typeEffect in surfaceType.Surface.ImpactTypeEffects)
            {
                if (typeEffect.ImpactType == Impact)
                {
                    PlayEffects(HitPoint, HitNormal, typeEffect.SurfaceEffect, 1);
                }
            }
        }
        else
        {
            // Handle case where no matching SurfaceType is found...
            foreach (Surface.SurfaceImpactTypeEffect typeEffect in DefaultSurface.ImpactTypeEffects)
            {
                if (typeEffect.ImpactType == Impact)
                {
                    PlayEffects(HitPoint, HitNormal, typeEffect.SurfaceEffect, 1);
                }
            }
        }
    }
    else
    {
        // Handle case where no MaterialTypeTag is attached...
        foreach (Surface.SurfaceImpactTypeEffect typeEffect in DefaultSurface.ImpactTypeEffects)
        {
            if (typeEffect.ImpactType == Impact)
            {
                PlayEffects(HitPoint, HitNormal, typeEffect.SurfaceEffect, 1);
            }
        }
    }
}
```

Fonte: Autoria Própria.

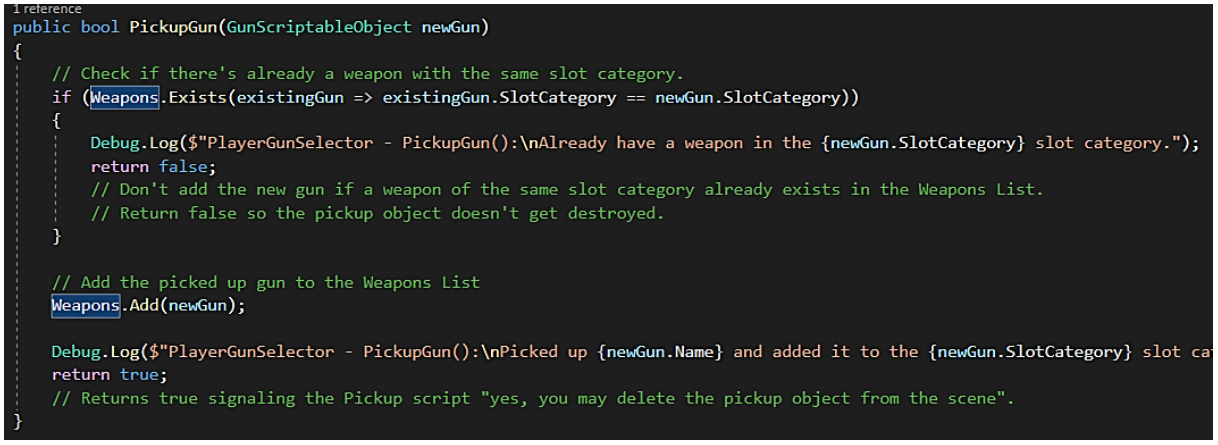
O preenchimento das superfícies e feito no Editor. São criados *ScriptableObjects* para superfícies, que seguram os efeitos para cada tipo de impacto desejado.

4.4.10 Inventory e Item Pickup System

O sistema de inventário e coleta de itens também foram sistemas simples de implementar. O inventario do jogador é gerido por um script nominado *PlayerGunSelector*, uma das variáveis presentes neste script é uma simples lista de *GunScriptableObject* denominada *Weapons*. Neste script há uma função chamada *PickupGun*(*GunScriptableObject newGun*), esta função e responsável por adicionar

um armamento a lista a *Weapons.* A Figura 33 mostra a função *PickupGun* em sua totalidade.

Figura 33: Função *PickupGun*() em *PlayerGunSelector*.cs.



```
1 reference
public bool PickupGun(GunScriptableObject newGun)
{
    // Check if there's already a weapon with the same slot category.
    if (Weapons.Exists(existingGun => existingGun.SlotCategory == newGun.SlotCategory))
    {
        Debug.Log($"PlayerGunSelector - PickupGun():\nAlready have a weapon in the {newGun.SlotCategory} slot category.");
        return false;
        // Don't add the new gun if a weapon of the same slot category already exists in the Weapons List.
        // Return false so the pickup object doesn't get destroyed.
    }

    // Add the picked up gun to the Weapons List
    Weapons.Add(newGun);

    Debug.Log($"PlayerGunSelector - PickupGun():\nPicked up {newGun.Name} and added it to the {newGun.SlotCategory} slot ca
    return true;
    // Returns true signaling the Pickup script "yes, you may delete the pickup object from the scene".
}
```

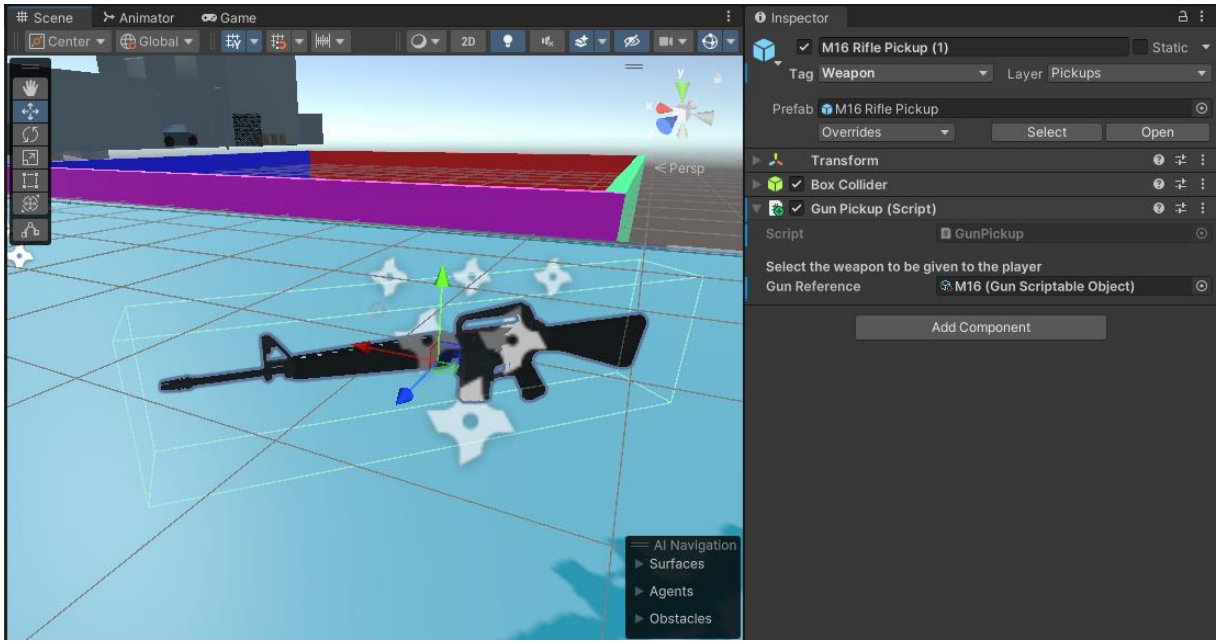
Fonte: Autoria Própria.

O algoritmo desenvolvido para gerenciar a mecânica de coleta de armas (*GunPickup.cs*) permite a coleta por personagens controlados pelo jogador ou por uma inteligência artificial (AI). A principal funcionalidade é a atribuição de uma arma ao personagem que interagir com um objeto de coleta no ambiente do jogo, preservando a integridade dos dados da arma original.

O sistema funciona da seguinte maneira: ao iniciar, o algoritmo cria uma cópia da arma pré-configurada (arma de referência *GunScriptableObject*). Essa clonagem é realizada para evitar alterações no estado original da arma ao longo do gameplay visto que *ScriptableObjects* mantem valores modificados em modo de jogo. Logo a clonagem permite que a versão clonada seja modificada sem impactar a arma base.

Quando o personagem entra em contato com o objeto de coleta, o algoritmo verifica se ele possui a capacidade de receber armas, consultando se o componente adequado está presente no personagem, ou seja, se ele possui um (*PlayerGunSelector*). Caso o personagem seja elegível, a arma clonada é atribuída ao seu inventário através da chamada de função *PickupGun*() do seu componente *PlayerGunSelector*. Após a coleta bem-sucedida, o objeto de coleta é removido da cena, evitando que a arma seja recolhida mais de uma vez. Além disso, o algoritmo permite que tanto jogadores quanto personagens controlados pela AI possam interagir com o objeto, utilizando uma lógica similar para ambos, porém para agentes de AI a o componente a ser buscado é *AiWeapons*. A Figura 34 mostra a configuração de uma das armas disponíveis para coleta no cenário.

Figura 34: M16 coletável



Fonte: Autoria Própria.

Essa abordagem garante flexibilidade, protegendo os dados originais e permitindo que diferentes tipos de personagens interajam com o sistema de forma controlada.

4.5 Testes

O teste de mecânicas permite identificar e corrigir bugs, equilibrar a dificuldade do jogo e assegurar que todas as funcionalidades funcionem conforme o esperado. Além disso, os testes ajudam a avaliar a resposta dos jogadores às diferentes mecânicas, fornecendo insights valiosos sobre o que pode ser melhorado ou ajustado. Sem uma fase de testes, o jogo corre o risco de apresentar problemas técnicos e de jogabilidade que podem frustrar os jogadores e impactar negativamente a recepção do jogo no mercado. Portanto, investir tempo e recursos em testes é essencial para o sucesso e a qualidade final do produto.

Desenvolvedores indie frequentemente enfrentam dificuldades em obter *feedback* sobre aspectos de seu projeto devido à falta de recursos e de uma base de jogadores estabelecida. Sem acesso a um grande público, pode ser desafiador identificar problemas de jogabilidade e entender as preferências dos jogadores. Além disso, a ausência de uma equipe dedicada de testes pode resultar em *bugs* e desequilíbrios que passam despercebidos.

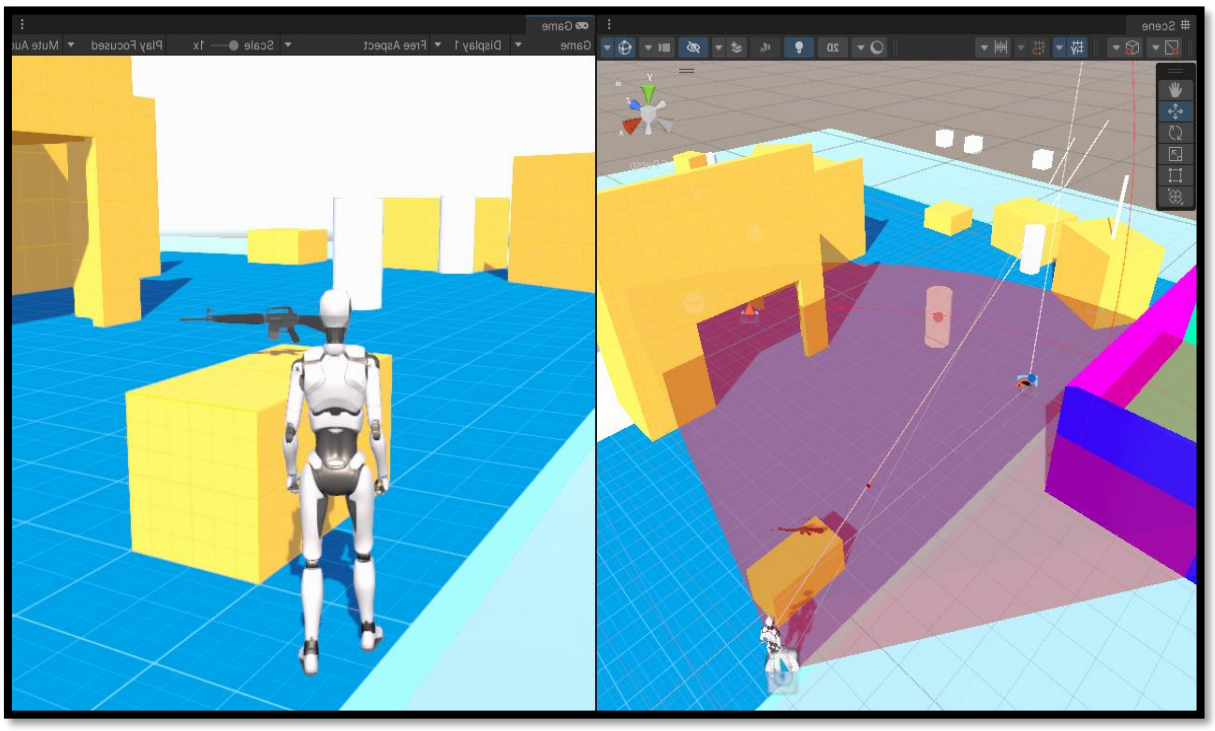
Para mitigar a falta de feedback direto dos jogadores e garantir que o jogo seja polido e bem-recebido no mercado, desenvolvedores indie podem adotar várias estratégias:

- **Parcerias e Colaborações**: Trabalhar com outros desenvolvedores ou artistas, pode fornecer perspectivas valiosas e ajudar a identificar problemas.
- **Comunidades Online**: Participar de fóruns e grupos de desenvolvedores de jogos pode ser uma excelente maneira de obter *feedback*.
- **Testes Internos**: Realizar sessões de jogo com amigos, familiares ou colegas pode ajudar a simular a experiência do jogador e identificar áreas que precisam de ajustes.
- **Ferramentas de Teste Automatizadas**: Utilizar *software* de teste automatizado pode ajudar a detectar bugs e problemas de desempenho que podem não ser óbvios durante o desenvolvimento.

No contexto deste projeto, foram realizados testes internos e coletado *feedback* qualitativo para aperfeiçoamento de características, adição de novas mecânicas e definição de novos requisitos funcionais.

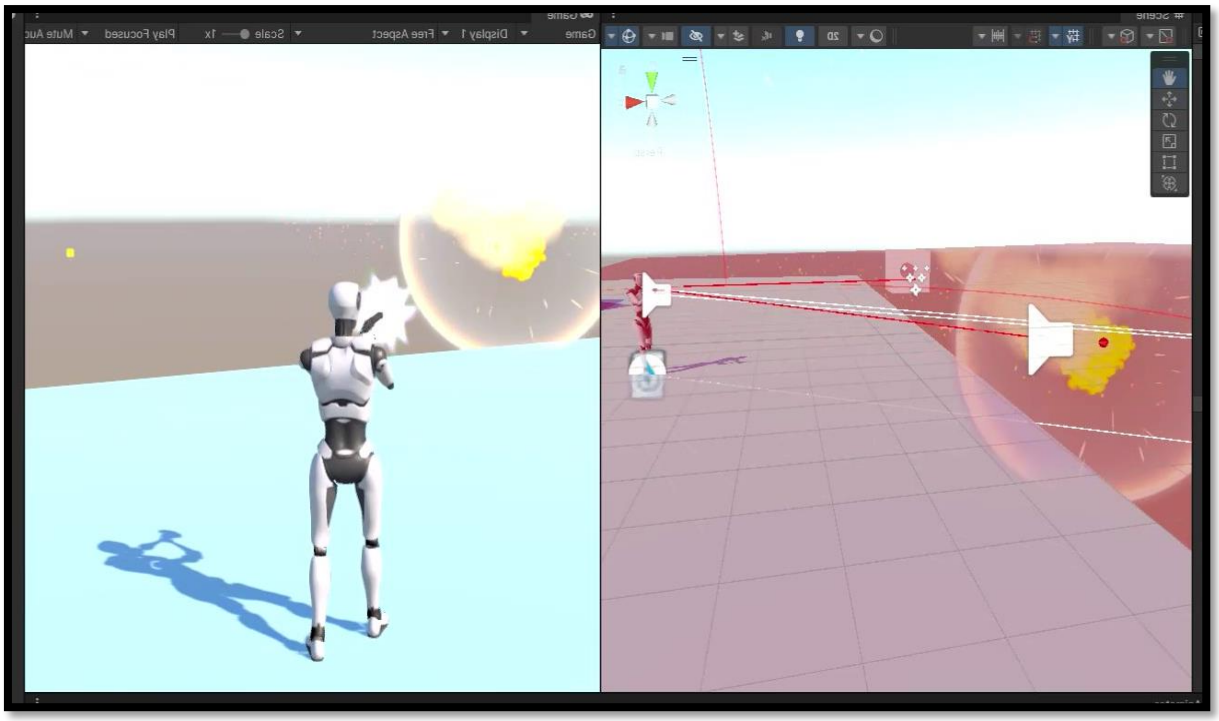
As Figuras 35, 36 e 37 apresentam alguns testes realizados de mecânicas atualmente presentes no jogo.

Figura 35: Personagem detectando inimigos e podendo coletar itens



Fonte: Autoria Própria.

Figura 36: Personagem fazendo mira e disparando em um inimigo estático



Fonte: Autoria Própria.

A Figura 36 ilustra um teste realizado no ambiente de desenvolvimento do jogo no qual o personagem principal está mirando e disparando em um inimigo estático, representado por um cubo flutuante.

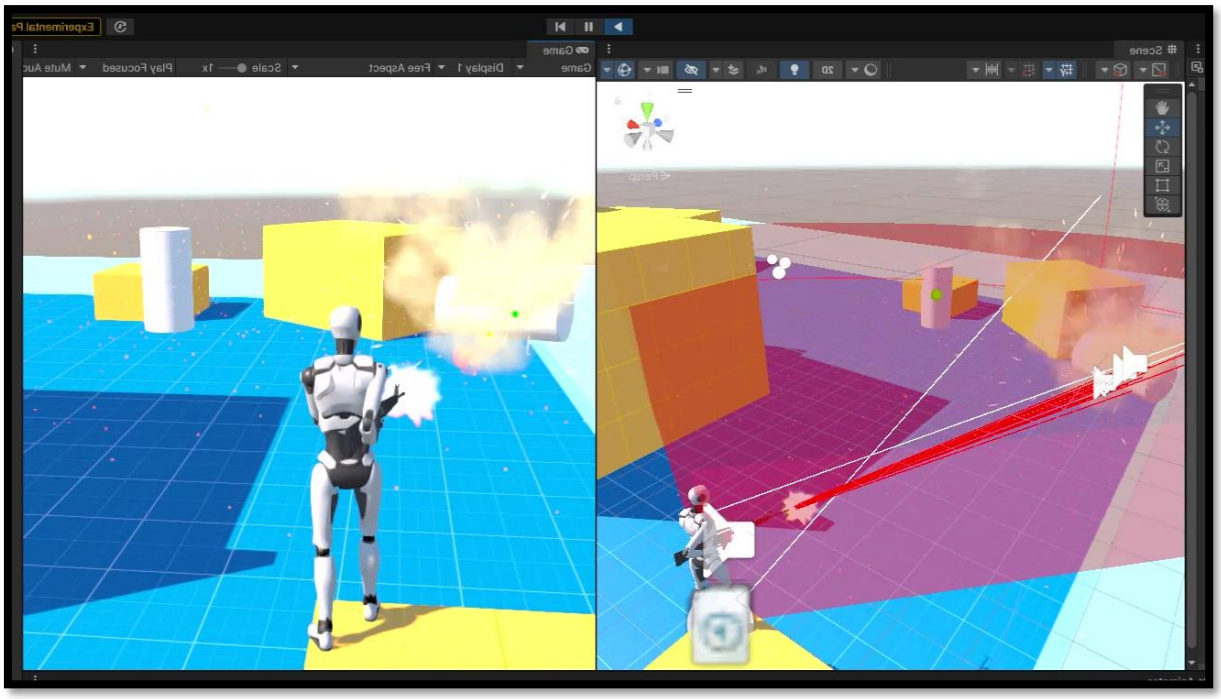
O objetivo deste teste inicial foi a garantia de que as mecânicas de tiro do jogo estivessem funcionando conforme o esperado. Neste simples teste a tela está dividida em duas partes para facilidade de visualização. Na primeira parte, lado esquerdo da tela, vemos o personagem em ação, disparando sua arma. Na segunda parte, é possível observar efeitos visuais adicionais de suporte ao desenvolvimento, compostos por *gizmos* e efeitos de *debug* projetados para possibilitar a visualização do traçado do projetil, área do sensor e identificação de alvos, representado respectivamente por um ícone de áudio em formato de auto falante, uma linha de *debug* vermelha, o volume preenchido em vermelho e uma esfera ao centro do alvo.

Pode-se através deste teste facilmente validar o funcionamento das mecânicas previamente implementadas, aspectos como: o comportamento da arma e suas características previamente definidas (dispersão, curva de coice/recuo, cadência de tiro, precisão etc.); sua interação com o cenário de jogo e integração com outros sistemas como por exemplo o *Surface Manager*, afim de garantir que o instanciação e trajetória de percurso do projetil seja coesa com os valores estipulados, o dano

casado ao alvo, e os efeitos especiais visuais e sonoros sejam condizentes com os estipulados.

A Figura 37 mostra um teste realizado com o personagem efetuando disparos em um inimigo móvel que se move de maneira aleatória no cenário de desenvolvimento.

Figura 37: Personagem para efetuando disparos em um inimigo móvel



Fonte: Autoria Própria.

O objetivo deste teste foi o de refinamento de características balísticas e validação do sistema de *tracking* ou “acompanhamento” de inimigos pelo sensor. O inimigo se move em direções aleatórias, adicionando um elemento de imprevisibilidade ao teste. Nesta cena, o personagem está equipado com uma arma longa, que possui características balísticas como dano e precisão elevados quando comparado a arma curta utilizada no teste anterior. No momento acerto do alvo foram adicionados efeitos visuais diferentes para este tipo de projetil, estes efeitos exagerados (explosões) propositalmente.

Por este teste foi possível comprovar a perfeita detecção e acompanhamento dos movimentos do inimigo por parte do sensor, assim como a correta atribuição e execução dos efeitos de uma nova superfície atribuídas a este alvo, sendo gerenciadas pelo *Surface Manager*. A coleta de itens e sua correta atribuição ao inventario do jogador também pode ser validada de maneira satisfatória. Após

algumas iterações a fim de correções de bugs relacionados aos colisores e suas interações com itens em cena foi possível chegar em um resultado satisfatório que culminasse em um sistema simples, porém eficaz.

Nas Figuras 35 e 37 pode-se notar na parte direita da tela o campo de visão do personagem, representado por um cone frontal em vermelho, este é o sensor utilizado pelo sistema de mira automática.

Objetos de interesse (filtrados) que entram neste sensor são adicionados a uma lista de objetos e usados pelo sistema de mira automática do personagem. Com isso é possível determinar se o sensor está funcionando corretamente. Novamente, o traçado da projetil é visualizado por meio de linhas de *debug* vermelhas, permitindo que para este teste sejam validadas e testadas as características balísticas de cada arma podendo-se ajustar, precisão e dispersão dos disparos.

Pela execução destes simples testes foram gerados feedbacks e coletados informações de extrema importância no que tange necessidades adicionais por parte do sistema, mudanças necessárias de *design* e possíveis problemas de balanceamento. Também foi possível validar a implementação das funcionalidades de sistemas como: *Character Controller, Surface Manager, Damage and Health Systems,* comprovando seu funcionamento na prática, possibilitando a identificação e correção de quaisquer problemas na lógica ou implementação, melhorando aspectos como desempenho e jogabilidade geral do jogo.

5 ANÁLISE DOS RESULTADOS OBTIDOS E DISCUSSAO

O projeto utilizou os conceitos da Engenharia de Software (ES) para desenvolver um protótipo de jogo, envolvendo várias etapas, desde uma revisão bibliográfica sobre a ES e desenvolvimento de jogos até a criação de um protótipo jogável. A abordagem utilizada incluiu o uso de *frameworks* conhecidos no desenvolvimento de jogos, como o MDA *(Mechanics, Dynamics, Aesthetics)* aplicadas ao Documento de *Design* de Jogo ou Game *Design Document* (GDD).

A aplicação mesmo que informal da ES em conjunção da aplicação de processos mais formais adaptados como descrição de requisitos, desenvolvimento modular e validação de funcionalidades trouxe melhorias significativas em todos os processos de execução do projeto. Foi possível notar um claro aumento na elucidação dos problemas, suas possíveis soluções e estratégias/planos de desenvolvimento.

Os artefatos gerados pelo projeto foram: GDD, Documento de Alta Visão,

Documento de *Design* de Personagens e Cenários, Documento auxiliar de criação de narrativas e um documento de análise simplificado.

As funcionalidades desenvolvidas incluem: um sistema de combate robusto, com mira automática, armamento modular e dinâmico, retículos dinâmicos, detecção de colisões e gerenciamento de superfícies, sistema de Inteligência artificial utilizando de FSM, sistema de gerenciamento de saúde, sistema de troca de cenas, sistema de coleta de itens simples e um sistema dinâmico de troca de câmeras.

A comparação com o trabalho "XS-GAME" de Brian Rocha Confessor (2019) destaca semelhanças importantes, como a aplicação de conceitos da Engenharia de Software, o uso de metodologias ágeis adaptadas, e a criação de protótipos para validar mecânicas de jogo. Ambos os projetos utilizaram Unity e buscaram otimizar o processo de desenvolvimento através da prototipagem.

No entanto, enquanto o "XS-GAME" foca na criação de uma metodologia voltada para desenvolvedores individuais, com uso de Histórias e uma abordagem mais flexível de Scrum e XP, o presente trabalho segue um escopo mais amplo, com foco no desenvolvimento de um sistema completo de combate, inteligência artificial e outros subsistemas, estruturado através de documentos como GDD. Confessor sugere que a documentação informal dos artefatos da ES esta sugestão foi seguida para este projeto, utilizando conceitos teóricos adaptados ao contexto pessoal para produção dos artefatos.

Quando comparado ao trabalho de Freitas (2017) intitulado “ENGENHARIA DE SOFTWARE: JOGOS ELETRÔNICOS” e citada a multidisciplinariedade atrelada ao desenvolvimento de jogos evidenciada pelo atual projeto. Freitas utiliza de uma abordagem formal de desenvolvimento gerando artefatos da ES como: diagramas de caso de uso, diagrama de classes, diagramas de sequência, modelagem do banco de dados e desenhos de interfaces. O enfoque do presente trabalho não foi na aplicação formal da ES como realizado por Freitas e sim na sua tradução para modelos de documentação mais práticos contendo somente aspectos considerados essenciais para auxílio do desenvolvimento.

Realizando este projeto foi possível observar que, mesmo sendo um desenvolvedor amador, é possível aplicar práticas de ES para criar um produto funcional e otimizado. O uso de boas práticas de ES ajudou a organizar e estruturar o processo de criação do jogo, mesmo com restrições severas de tempo e recursos. A documentação informal e a abordagem incremental iterativa garantiram o rápido foco

na implementação prática dos sistemas e clara visualização dos requisitos e etapas de desenvolvimento.

6 CONCLUSÃO

Este trabalho teve o intuito de responder a seguinte questão de pesquisa: Seria possível desenvolver jogos aplicando conhecimentos da Engenharia de Software e a ferramenta *Unity3d*?

O objetivo geral foi de desenvolver um jogo de videogame (game), aplicando conhecimentos da ES e utilizando a ferramenta Unity3d.

O estudo realizado permitiu concluir que desenvolver jogos como amador implica em desafios que muitas vezes geram frustração e desânimo, pois, neste caso, a tarefa empreendida de maneira solitária, dificulta entrelaçar teoria à prática de maneira formal e protocolar, nisso, o perfil do desenvolvedor do jogo passa a ter de ser multifacetado, exigindo que assuma a capacidade e habilidade de ser gestor, gerente de projetos, *design* gráfico, desenvolvedor e programador simultaneamente.

No decorrer do projeto, com o desenrolar das atividades inerentes ao jogo, surgiram, lacunas e dificuldades exigindo retomada da teoria que por vezes redirecionaram as ações. Muitos conceitos e atividades previstas pela ES tiveram de sofrer adaptações devido ao escopo do projeto e a habilidade do estudante. Além das obrigações impostas na vida secular, como conciliar trabalho e estudo, outros fatores também se interpuseram no alcance dos objetivos: o arcabouço teórico insuficiente, falta de equipe, exíguo conhecimento das ferramentas disponíveis para a finalidade de desenvolver jogos e de poucos recursos financeiros.

Foi possível concluir também que, embora tendo de driblar os percalços, notou-se que a ES permite um panorama geral acerca de soluções, tarefas e maneiras de implementar o jogo. A aplicação, mesmo que informal da ES no processo de desenvolvimento de jogo propiciou um norte para que todas as atividades vinculadas ao *design*, arte, implementação e busca de soluções técnicas redundassem em resultado satisfatório e encorajadores. Quando comparado ao projeto anterior, no qual a falta de especificação e formalização de artefatos predominou foi possível notar um grande aumento em produtividade, clareza, e motivação ao realizar as tarefas previstas permitindo realmente visualizar o progresso em cada etapa do desenvolvimento.

Para continuidade deste trabalho sugere-se as seguintes sugestões de

trabalhos futuros:

- Desenvolvimento de um novo *Character Controller* (mais avançado)
- Sistema para persistência de progresso (*Save system*)
- Polimento geral (animações, áudio, pós-processamento)

7 REFERÊNCIAS BIBLIOGRÁFICAS

ALFAMÍDIA. **Desenvolvimento profissional de jogos *indie****: o desenvolvedor individual*. Disponível em: <http://www.alfamidia.com.br/14-02-2019-desenvolvimento-profissional-de-jogos-indie-o-desenvolvedor-individual/>. Acesso em: 25 ago. 2024.

ATLASSIAN**. *Agile Kanban***. Disponível em: <https://www.atlassian.com/agile/kanban>. Acesso em: 05, mar. 2024.

BOND, Jeremy.: ***Introduction to Game Design, Prototyping, and Development****: From Concept to Playable Game with Unity and C#.* Addison-Wesley Professional, 2022.

CAILLOIS, Roger.: **Os jogos e os homens**: A máscara e a vertigem. Tradução de Maria Ferreira. Petrópolis: Vozes, 2017.

CONFESSOR, Brian.: **XS-*Games***: Engenharia de jogos voltada para desenvolvedores individuais. Rio de Janeiro, 2019.

FREITAS, Eric.: **Engenharia de Software**: Jogos Eletrônicos. Curitiba, 2017.

GIL, Antônio Carlos.: **Como Elaborar Projetos de Pesquisa**. 7. ed. São Paulo: Editora Atlas Ltda., 2021.

GULARTE, Daniel.: **Jogos Eletrônicos**: 50 anos de interação e diversão. Teresópolis: Novas Ideias, 2010.

HUIZINGA, Johan.: ***Homo ludens****:* o jogo como elemento da cultura. 5ª ed. São Paulo: Perspectiva, 2007.

HUNICKE, Robin; LEBLANC, Marc; ZUBEK, Robert. **MDA**: *A Formal Approach to Game Design and Game Research*. Disponível em: https://users.cs.northwestern.edu/~hunicke/MDA.pdf. Acesso em: 20 de mar. de 2024.

KISHIMOTO, T. M. **O jogo e a educação infantil**. In: KISHIMOTO, T. M. (org.). Jogo, brinquedo, brincadeira e a educação. Cortez: São Paulo, 2003.

KURHAN, Chris. ***Accurate Aiming Options Discussed & Implemented*** | *Gun Series 7* | *Unity Tutorial*. Disponível em: <https://youtu.be/x8ECpNWMmag?si=ejUY93ioylU2Dmfe>. Acesso em: 29 out. 2024.

LEAL, André Luiz de Castro.: **Engenharia de Software e Jogos Digitais**: Uma experiência de ensino e extensão. Revista de Extensão Tecnológica - IFC. Vol. 11, No. 19. Blumenau, SC. Jan. - Jul. 2023.

LIU, J., Ho, C.-Y., CHANG, J., & TSAI, J. C.: ***The role of Sprint planning and feedback in game development projects****: Implications for game quality. Journal of Systems and Software*. Elsevier, 2019.

MARQUES, Marcio.: ***Game Engines***: Conceitos e aplicações no desenvolvimento de jogos digitais. Warpzone, 2020. Disponível em: <https://warpzone.me/game-engines-conceitos-e-aplicacoes-no-desenvolvimento-de-jogos-digitais/> Acesso em: 10 de mar. de 2024.

McLAUGHLIN, Buzz. ***Developing organic story structure…*** Disponível em: <https://buzzmclaughlin.com/developing-organic-story-structure/>. Acesso em: 02 out. 2024.

MARTINSON, Júlia. **Jornada do Herói**: O que é, suas etapas e como utilizar. Resultados Digitais, 2021. Disponível em: < https://resultadosdigitais.com.br/agencias/jornada-do-heroi/> Acesso em: 10 de mar. de 2024.

MELISSINOS, Chris. ***Video Games Are One of the Most Important Art Forms in History***. Time, 22. set. 2015. Disponível em: <https://time.com/collection-post/4038820/chris-melissinos-are-video-games-art/>. Acesso em: 05 Mar. 2024.

MKT ESPORTS. **O que é NPC?** Disponível em: <https://mktesports.com.br/blog/dicionario-gamer/o-que-e-npc/>. Acesso em: 20 out. 2024.

NATIONAL RESEARCH COUNCIL: ***How people learn****: brain, mind, experience, and school. Expanded edition*. Washington, DC: The National Academies Press, 2000.

PACETE, Luiz. **2022 promissor**: mercado de games ultrapassará US$ 200 bi até 2023. Forbes Tech. 03 jan. 2022. Disponível em: < https://forbes.com.br/forbes-tech/2022/01/com-2022-decisivo-mercado-de-games-ultrapassara-us-200-bi-ate-2023/>. Acesso em 03 mar. 2024.

PRESSMAN, R. S.; Maxim, B.: **Engenharia de *Software***: uma abordagem profissional. 9ª ed. Porto Alegre: AMGH, 2021.

RAMOS, Elisa.: **Relações entre o tema da comunidade e os jogos**. Boletim De Pesquisa NELIC 20.32 (2022): Boletim De Pesquisa NELIC, 2022, Vol.20 (32). Web.

REDAÇÃO DO GE. **Pesquisa Games Brasil 2022**: público de games aumentou para 74,5%. São Paulo,18 abr. 2022. Disponível em: <https://ge.globo.com/esports/noticia/2022/04/18/pesquisa-games-brasil-2022-publico-de-games-aumentou-para-745percent.ghtml> Acesso em: 05 de mar. de 2024.

RICCHIUTI, Diego.: ***Game design tools****: cognitive, psychological, and practical approaches*. *First edition*, Boca Raton, FL, CRC Press, 2023.

SCHMIDT, D. C.; GOKHALE, A.; NATARAJAN, B. "***Leveraging application frameworks.***" ACM Queue, v.2, n. 5. jul. - ago. 2004.

SMUTS, Aaron.: ***Are video games art?*** *Contemporary Aesthetics*. *American Society for Aesthetics Newsletter. Analytics.* Wisconsin*,* Madison. Michigan *Publishing*. 2005. Acesso em: 05 mar. 2024. Disponível em: http://hdl.handle.net/2027/spo.7523862.0003.006

SOMMERVILLE, I.: **Engenharia de *Software***. 10ª ed. São Paulo: Pearson, 2019.

UNITY TECHNOLOGIES. ***Animator*.** Disponível em: <https://docs.unity3d.com/Manual/class-Animator.html>. Acesso em: 29 ago. 2024.

UNITY TECHNOLOGIES. ***Introduction to Object Pooling***. Disponível em: <https://learn.unity.com/tutorial/introduction-to-object-pooling>. Acesso em: 20 out. 2024.

UNITY TECHNOLOGIES. ***Unity - Scripting API****: Physics2D.Raycast*. Disponível em: <https://docs.unity3d.com/ScriptReference/Physics2D.Raycast.html>. Acesso em: 20 set. 2024.

UNITY TECHNOLOGIES. ***Scriptable Objects***. Disponível em: <https://docs.unity3d.com/Manual/class-ScriptableObject.html>. Acesso em: 14 ago. 2024.

WAKKA, Wagner.: **Mercado de games agora vale mais que indústrias de música e cinema juntas**. Canaltech, 2021. Acesso em: 02 de mar. de 2024. Disponível em: https://canaltech.com.br/games/mercado-de-games-agora-vale-mais-que-industrias-de-musica-e-cinema-juntas-179455/

WAZLAWICK, R. S.: **Metodologia da Pesquisa para Ciência da Computação**. 2ª. ed. Campus, 2014.

WILTSHIRE, Alex*.: **How hitboxes work.** PCGamer*. 10, ago. 2020. Disponível em: <https://www.pcgamer.com/how-hitboxes-work/> Acesso em: 05, set. 2024.

WIKIPEDIA ***Feedback.*** Disponível em: < https://en.wikipedia.org/wiki/Feedback>. Acesso em: 01, mar. 2024.

WIKIPEDIA. ***GitHub***. Disponível em: <https://en.wikipedia.org/wiki/GitHub>. Acesso em: 05, mar. 2024.

WIKIPEDIA. ***Inverse Kinematics*** Disponível em: < https://en.wikipedia.org/wiki/Inverse_kinematics> Acesso em: 05, mar. 2024.

WIKIPEDIA*. **Unity (game engine)**.* Disponível em: < https://en.wikipedia.org/wiki/Unity_(game_engine)>. Acesso em: 08, ago. 2024.