Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

GUSTAVO FIUZA MARTINS

EXPLORANDO OS GAMES: ESTUDO SOB UMA PERSPECTIVA COMPUTACIONAL E MATEMÁTICA

Assis/SP
2023

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

GUSTAVO FIUZA MARTINS

EXPLORANDO OS GAMES: ESTUDO SOB UMA PERSPECTIVA COMPUTACIONAL E MATEMÁTICA

Trabalho de Conclusão de Curso apresentado ao curso de Bacharelado em Ciências da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito parcial à obtenção do Certificado de Conclusão.

Orientando(a): Gustavo Fiuza Martins
Orientador(a): Dr. Luiz Ricardo Begosso

Assis/SP
2023

Martins, Gustavo Fiuza

M386e Explorando os games: estudo sob uma perspectiva computacional e matemática / Gustavo Fiuza Martins. -- Assis, 2023.

63p. : il.

Trabalho de Conclusão de Curso (Ciência da Computação) -- Fundação Educacional do Município de Assis (FEMA), Instituto Municipal de Ensino Superior de Assis (IMESA), 2023.
Orientador: Prof. Dr. Luiz Carlos Begosso.

1. Aprendizagem computacional. 2. Desenvolvimento de tecnologia. 3. Game engine. I Begosso, Luiz Carlos. II Título.

CDD 004.2

Elaborada por Anna Carolina Antunes de Moraes – Bibliotecária – CRB-8/10982

“Se eu vi mais longe, foi por estar sobre o ombro de gigantes.”

– Isaac Newton

AGRADECIMENTOS

Agradeço a Deus por ter chegado até aqui, vivo, saudável e forte, por nada de grave ter acontecido comigo ou com minha família e por me dar a graça e o privilégio de poder viver para contribuir positivamente no mundo, ser ferramenta para melhorar a vida e não degradá-la.

Também deixo minha gratidão a minha família principalmente a minha avó Dirce Castelo Fiuza, que me criou desde pequeno depois do falecimento de minha mãe, 19 dias após o parto. Ela contribuiu com toda minha formação básica de caráter em casa e hoje levo a sério os estudos e os demais trabalhos que faço na minha vida por conta dessa criação. E acredito que de fato eu pude ir mais longe, devido a estrutura familiar forte que possuía em casa.

Não posso deixar de mencionar meus amigos Lucas de Assis Lima e Matheus Gustavo Santos Silva, que me incentivaram a entrar na faculdade em um período onde eu estava meio perdido no que fazer devido a alguns fracassos pessoais, e uma vez inserido no curso, me ajudaram a entrar no mercado e me desenvolver academicamente e profissionalmente.

Por fim mas não menos importante, eu quero deixar minha gratidão a todos os meus professores pela dedicação e capacitação, pois através das aulas pude abrir minha cabeça não somente para a área em que estou hoje introduzido mas para uma nova forma de aprender. Levarei isso para toda minha vida!

RESUMO

O presente trabalho possui o objetivo de desenvolver um estudo a fim de sintetizar o conhecimento sobre desenvolvimento de games, com foco no processo computacional de construção. Após a pesquisa, será criada uma *game engine* e um jogo baseado nela, a fim de demonstrar e pôr em prática um projeto com os conhecimentos adquiridos. Além de proporcionar uma visão mais refinada sobre o assunto, o estudo possibilita uma compreensão clara e objetiva, facilitando o aprendizado de terceiros.

Palavras-Chave: desenvolvimento; *game engine;* pesquisa; aprendizado; processo computacional.

ABSTRACT

The present work aims to develop a study in order to synthesize knowledge about game development, focusing on the computational process of construction. After the research, a game engine will be created along with a game based on it, in order to demonstrate and put into practice a project using the acquired knowledge. In addition to providing a more refined insight into the subject, the study enables a clear and objective understanding, facilitating the learning process for third parties.

Keywords: development; game engine; research; learning; computational process.

LISTA DE ILUSTRAÇÕES

SUMÁRIO

1. INTRODUÇÃO

O mercado de *games* há muito tempo conquistou o seu espaço e sempre demonstra uma crescente próspera em relação aos anos anteriores. Desde a popularização dele na década de 1980 na era 8-bits, foi um mercado que sempre teve potencial, mas sua fatia era limitada devido a disponibilização de dispositivos, tendo em vista que a evolução dos jogos eletrônicos, está diretamente ligada a evolução de outros itens tecnológicos como televisores, computadores e sistemas de transmissão de dados por exemplo, como diz Barboza (2014).

A indústria dos *games* hoje já vale mais do que a indústria do cinema, com grande parte desse valor vindo do mercado *mobile* como destacado por Pittol (2019), e se revela extremamente interdisciplinar, gerando empregos para pessoas de diversas áreas com o intuito de gerar um único produto, contando para seu desenvolvimento desde artistas (como desenhistas, músicos e atores) até programadores e analistas sendo essa só uma fatia da produção.

Outra área favorecida pelo crescimento desse mercado é a pesquisa, que mesmo por dentro do setor de desenvolvimento de *games* sempre acaba atingindo e gerando novidades ou complementando outras áreas de inovações tecnológicas, como por exemplo o VR (*Virtual Reality* – Realidade Virtual), que hoje vem sendo explorado na área corporativa por meio do metaverso, apesar da tecnologia já existir e ser explorada há muito tempo pelos jogos eletrônicos.

Focando na parte de desenvolvimento de um jogo eletrônico, existem diversas ferramentas disponíveis para esta tarefa, e talvez a mais fundamental seja a *game engine*, que é um software criado para ajudar no desenvolvimento de um *game*. Entre elas pode-se citar as mais famosas do mercado como *Unity* (2022) e *Unreal Engine* (2022) por exemplo.

No entanto algumas empresas buscam fazer a sua própria *engine*, o seu próprio motor gráfico, justamente para atender melhor ao que elas desejam fazer, como observado na famosa *RE Engine* feita pela CAPCOM (2022), que vem sendo utilizada para criar os últimos jogos da série *Resident Evil*.

Ainda pensando nas *engines*, encontra-se o principal fator para o presente trabalho: entender como funciona um game de fato. Serão exploradas as principais práticas já

usadas na indústria, como era esse processo antigamente e ao final será demonstrado o que foi visto neste estudo, via implementação de uma *engine* própria, de forma simplificada.

1.1 OBJETIVOS

O objetivo principal deste trabalho é estudar o funcionamento de uma game engine e as práticas já usadas e abordadas dentro da indústria, para no fim implementar uma própria para fins de demonstração do aprendizado coletado durante a pesquisa, sintetizando o funcionamento de um game, tanto em termos de lógica de programação quanto de computação gráfica, assim servindo como base de estudo e referência para quem tem interesse em aprender mais sobre o tema.

1.2 JUSTIFICATIVA

Este trabalho é apoiado pela iniciativa de criar-se um material que descreva com clareza e de forma demonstrativa os passos para o desenvolvimento de um game e seu funcionamento computacional, a fim de elucidar para aqueles que desejam entender o funcionamento dos jogos por trás das ferramentas trazidas pelas empresas que desenvolvem as games *engines* atualmente.

1.3 MOTIVAÇÃO

Sendo um mercado em alta, e com muito interesse principalmente pelos mais jovens, o presente trabalho se sustenta na idealização de ser uma pesquisa que possa concentrar o conhecimento de várias fontes de forma clara e objetiva, para que outros estudantes com interesse sobre a área possam aprender com uma única fonte a principal base teórica de construção de games, além de ser um norte para quem está começando a desbravar essa área de estudo.

1.4 PERSPECTIVAS DE CONTRIBUIÇÃO

As pessoas que utilizarem este trabalho como base, terão a possibilidade de estendê-lo com um projeto próprio a fim de colocar em prática o que foi estudado aqui, além de adicionar sua própria experiência e até mesmo sua própria metodologia, possibilitando

outra visão e maximizando a área de desenvolvimento, visto que há vários estilos de jogos e as formas de desenvolver também se diferem em cada um.

2. ESTUDO SOBRE O DESENVOLVIMENTO DE GAMES

Falando em desenvolvimento de games, é interessante observar que os jogos eletrônicos são na verdade, uma espécie de evolução de consumo de obra de arte: eles podem possuir todo um roteiro construído, com vários personagens e reviravoltas como observado em um livro, existe a interação audiovisual como em um filme e além de tudo, você que controla as ações e conduz a história. Claro nem todos os jogos seguem essa receita, mas é válido observar o quão abrangente um game pode ser.

Levando isso em consideração, reforça-se aqui, que o presente trabalho traz seu foco em apenas uma parte da construção de um jogo eletrônico, que é a implementação do software. Mais em específico, em como o computador entende um jogo, ou seja, o entendimento computacional e matemático destes softwares. Hoje possuímos diversas *game engines* que fazem esse trabalho para acelerar o processo de desenvolvimento, no entanto é muito válido explorar essa área, tanto para fins de aprimoramento quanto para fins de criação de formas novas de implementações

Neste capítulo temos uma revisão literária sobre o assunto, abordando a evolução da criação de jogos, formas da implementação do software, formas de organizar e estruturar o desenvolvimento de um game e quais áreas matemáticas são utilizadas nessa atividade.

2.1 A CORRELAÇÃO COMPUTACIONAL ENTRE GAMES E A MATEMÁTICA

Os games por natureza, fazem parte da computação, que por sua vez surgiu a partir da matemática. É impossível não correlacionar essas áreas, pois toda a base da computação e por consequência dos games também é matemática. Logicamente outras áreas foram se acrescentando aos games, dando a forma que hoje conhecemos, mas toda sua parte de implementação, de software, de processamento, possui como base fundamental a matemática, e por isso, é importante destacar alguns pontos comuns que encontramos na computação gráfica e que é importante se ter em mente ao estudar o funcionamento dos jogos.

Segundo Tonéis (2016), a matemática aplicada aos jogos digitais, traz uma nova abordagem para a aplicação matemática clássica (que por sua vez envolve Geometria Analítica, Álgebra Linear, Estatística, Cálculo Numérico), e também para o pensamento

lógico matemática, uma vez que para a produção de jogos digitais, é necessário um novo olhar sobre os processos computacionais.

2.1.1 Matemática e Computação

Pensando nisso, é interessante voltar os olhos diretamente a base desse assunto: a teoria da computação. Sendo uma área que se origina diretamente da matemática, a computação evoluiu através de teoremas matemáticos e construção de alguns conceitos que se tornaram a base dessa ciência. Como afirma Diverio (2011), um programa é um conjunto estruturado de instruções que viabilizam uma máquina a realização de operações matemáticas básicas e de testes lógicos sobre os dados iniciais fornecidos, com objetivo de transformar estes dados em uma forma mais desejável. E ainda finaliza, afirmando que um programa deve ter uma estrutura de controle de operações e testes, fazendo a máquina por meio de sua leitura, manipular os dados com a finalidade de alcançar os resultados desejados.

Como se pode notar, existe uma forte relação entre o programa e a máquina nos modelos matemáticos, e isso acontece porque o programa entra dentro da fórmula de uma máquina. Falando em máquina, o objetivo desta, é suprir todas as informações necessárias para que a computação de um computador seja descrita segundo Diverio (2011). Uma máquina por definição é uma 7-upla ordenada, recebendo a seguinte expressão matemática:

$$M = (V, X, Y, \pi X, \pi Y, \Pi F, \Pi T)$$

Onde:

$V \rightarrow$ Conjunto de valores de memória

$X \rightarrow$ Conjunto de valores de entrada

$Y \rightarrow$ Conjunto de valores de saída

$\pi X \rightarrow$ Função de entrada tal que:

$$\pi X: X \rightarrow V$$

$\pi Y \rightarrow$ Função de Saída tal que:

$$\pi X: V \rightarrow Y$$

ΠF → Um conjunto de interpretações de operações onde, para cada identificador de operação F interpretando M, existe uma única função:

$$\Pi F: V \rightarrow V \text{ em } \Pi F$$

ΠT → Um conjunto de interpretações de testes tal que, para cada identificador de teste T interpretado por M, existe uma única função:

$$\Pi T: V \rightarrow \{ \text{verdadeiro, falso} \} \text{ em } \Pi T$$

Essa 7-upla representada acima, demonstra o funcionamento de uma máquina, e de fato isso ocorre dentro do processador, que teve sua arquitetura melhorada justamente para poder realizar testes e operações de forma mais rápida e eficiente, além de baratear o custo de produção. Então hoje temos máquinas com a mesma base teórica, porém podendo colocar isso em prática numa escala extremamente grande de processamento de dados, mas logo mais falaremos a respeito disso.

E finalizando essa relação matemática entre máquinas, programas, matemática, Diverio (2011), ainda diz que a computação é resumidamente, um histórico do funcionamento de uma máquina para um determinado programa, levando em conta seu valor inicial. Então tudo que instruímos dentro de um programa, vira um histórico para a máquina, dessa forma, em um jogo digital, também se obtém este histórico da computação, onde a máquina irá por meio de operações simples e testes lógicos, realizar todo o processamento que nos leva ao resultado que vemos em tela. Com essa visão mais aprofundada, juntando o conhecimento que iremos ver logo mais, fica mais fácil de entender o porque o processamento de um jogo exige tanto de um computador.

2.1.2 Matemática diretamente aplicada aos jogos digitais

Como mencionado anteriormente, algumas áreas da matemática clássica são amplamente utilizadas dentro dos jogos digitais. Talvez uma das mais perceptíveis seja justamente a Geometria Analítica, pelo fato de estarmos utilizando em boa parte do tempo o plano cartesiano.

Tonéis (2016), cita que a utilização mais simples do espaço cartesiano é a de representar de forma gráfica a localização de um ponto em um determinado plano, e acrescenta ainda que este uso é amplamente dado em trabalhos relacionados à cartografia. Também

relembra que é possível representar um segmento de reta ou qualquer outra localização necessária. E isso se aplicando ao contexto de jogos, poderia ser a localização de um objeto, de um jogador, de um inimigo ou até mesmo de uma armadilha.

Além disso, esse sistema também é utilizado em jogos estratégicos, como Batalha Naval e até mesmo xadrez, mesmo que não aproveitado em toda sua totalidade (utilizando os quatro quadrantes por exemplo), como demonstrado na figura 1.

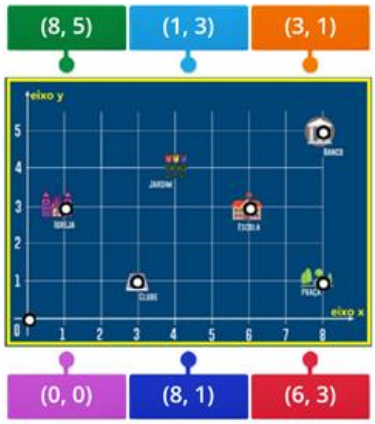

Figura 1: Exemplo de uso do plano cartesiano em um jogo de Batalha Naval

Dentro da computação gráfica, essa aplicação matemática é fundamental na construção de dois cenários que são amplamente trabalhados dentro dos jogos digitais: bidimensionalidade e tridimensionalidade, mais conhecidos também como 2D e 3D. Isso acontece, porque é por meio da representação de René Descartes que se torna possível representar fórmulas matemáticas e expressões de maneira gráfica.

Mais adiante foi se criado baseado nesse conceito um novo plano tridimensional, adicionando o eixo Z, possibilitando o trabalho com a Geometria Espacial, como observado na figura 2.

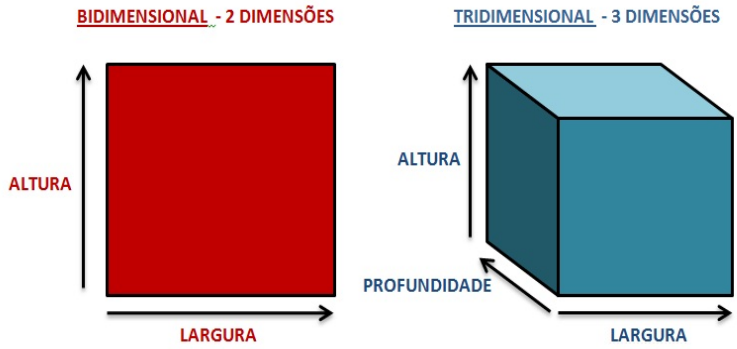

Figura 2: Exemplo de quadrados em um plano bidimensional e tridimensional

Foi dessa forma também foi possível representar imagens em monitores e televisores. Utilizando o Sistema Cartesiano e o sistema de cores RGB (Red, Green and Blue – Vermelho, Verde e Azul), foi criado uma forma de não somente desenhar imagens na tela, como conseguir colorir elas, utilizando como par ordenado os pixeis, que nos televisores vão utilizar as 3 cores e dependendo do que precisa exibir mudará seu comportamento.

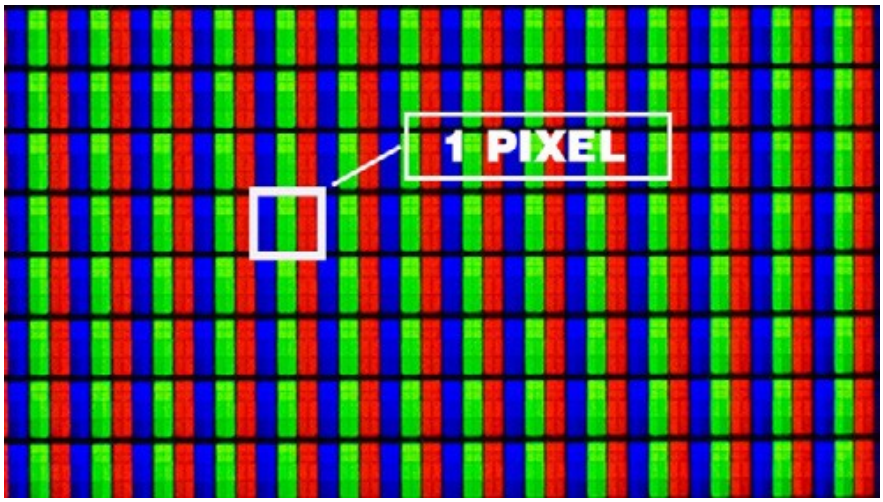

Figura 3: Pixeis RGB em um televisor

E dessa forma, as imagens que vemos hoje em nossos jogos são reproduzidas, trabalhando o plano cartesiano em conjunto com o estudo de matrizes, que serve para estabelecer e posteriormente localizar os pares ordenados dentro deste plano, com a imagem sendo o resultado dessa matriz criada.

2.2 A EVOLUÇÃO DOS GAMES EM SUA FORMA DE CRIAÇÃO

A produção de jogos eletrônicos era totalmente limitada anos atrás. Além das questões de hardware, não se havia uma compreensão do quão longe essa área poderia ir, no entanto, essa indústria sempre se destacou por estar contribuindo fortemente em inovações, estando sempre a frente do seu tempo. Um bom exemplo disso é o uso de tecnologias de realidade virtual, com o lançamento do VR-1 da SEGA lançado em 1994.

Hoje temos uma ampla gama de tecnologias prontas para trabalhar, com jogos incríveis que foram criados por apenas uma pessoa, mas isso não foi sempre assim. Lá atrás, fazer um jogo bem desenvolvido era uma tarefa extremamente pesada. Isso se deve principalmente pela escassez de pessoas com acesso à tecnologia, e também a hardwares potentes, que na época não era tão abundante como hoje.

Como observado por Barboza (2014), os games mais bem desenvolvidos foram criados com intuitos diferentes do lazer, sendo criados em laboratórios e instalações militares, para outros fins.

Apesar de não ser fácil acessar informações de como eram produzidos os jogos antigamente, existem alguns documentários mostrando os bastidores das empresas. Nestes documentários, é possível ver que o desenvolvimento de jogos era muito precário, com os gráficos sendo apresentados com puras figuras geográficas como blocos. Além disso a limitação de hardware dificultava produzir um game de qualidade, pois as artes gráficas deveriam ser pensadas para se adaptar a essas limitações como a falta de variedade de cores e também a questão do processamento gráfico que não poderia ser muito alto, fora também a questão da trilha sonora, onde os músicos precisavam criar trilhas pequenas para não ocupar tanto espaço em disco.

Por esses motivos, o desenvolvimento nas primeiras eras dos videogames eram tão limitados. Mesmo com um salto enorme do 16 bit para o 8 bit, ainda existiam limitações muito grandes, tanto que era impossível até então desenvolver um jogo com perspectiva tridimensional.

Após a façanha de conseguir reproduzir jogos tridimensionais na década de 80, os desenvolvedores foram buscando formas de simular o 3D explorando a Geometria Espacial e polígonos, contudo, a indústria ainda possuía um grande limitador era o hardware, pois um jogo precisa ser processado em tempo real, o que exige demais de um computador.

Mas com o avanço da tecnologia dos processadores e também das mídias físicas (cartucho, CD, DVD e Blu-Ray), a indústria pode se modernizar e utilizar métodos mais refinados e robustos para produção de jogos, sem ter que se preocupar em espremer algoritmos para encontrar um uso extremamente limpo de memória ou ficar cortando conteúdo por conta da baixa capacidade de armazenamento que era oferecida antigamente.

3. TECNOLOGIAS UTILIZADAS NO PROJETO

Para uma melhor compreensão sobre o que foi visto até aqui, será feita uma implementação com intuito de demonstrar o que foi citado e sua veracidade. Para tanto, serão utilizadas diferentes tecnologias que auxiliarão desde a criação e prototipação do projeto, até o momento de criar a *engine* e o game de fato.

Sendo assim, as seguintes ferramentas serão utilizadas nesse percurso:

Prototipação e Design Artístico:

- Mapa mental: MindMeister;
- Sprites: Paint.net, Asesprite;
- Músicas: BeepBox.

Desenvolvimento:

- Linguagem: Java;
- Bibliotecas: Swing, awt;
- IDE: Eclipse.

3.1 PROTOTIPAÇÃO E PREPARAÇÃO

Para poder mentalizar e formalizar o que se pretende fazer em qualquer projeto, é muito claro dentro da engenharia de software a necessidade de se criar um protótipo do que será o projeto. Nessa linha de discussão foram criados diversos modelos de prototipação para auxiliar o desenvolvimento e manutenção do software.

No entanto, não será utilizado todo este caminho para um game, pois existem requisitos diferentes, que pedem por métodos diferentes de prototipação, que acompanharemos ao longo desta parte do projeto.

3.1.1 Mapa Mental

O mapa mental presente em diversos outros trabalhos e muito recorrente para organização de *brainstorms* em projetos de software, é muito bem-vindo na prototipação de um game também.

Neste trabalho será utilizado o MindMeister, que é um software Web que permite a exportação do mapa desenhado após sua criação.

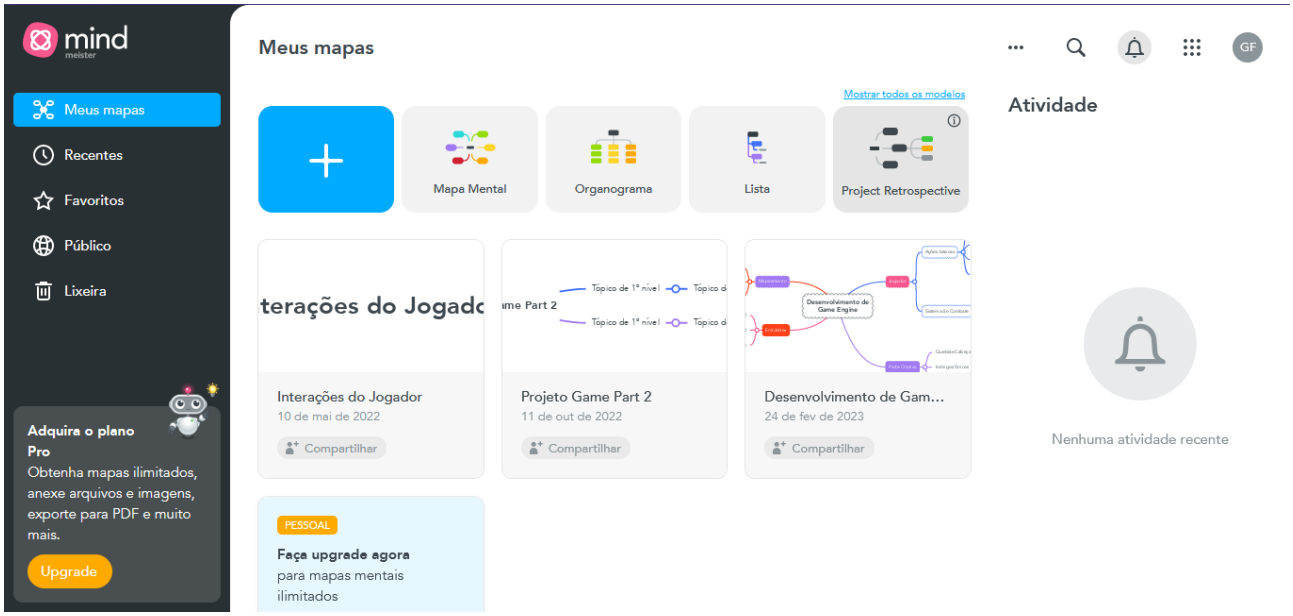

Figura 4: Interface inicial do MindMeister

Para poder acessá-lo, é preciso ter uma conta no site, que por sua vez também oferece versões pagas com diferentes ganhos em sua usabilidade, a versão gratuita serve principalmente para conhecer a ferramenta, mas apesar de ser bem limitada em comparação a paga, já serve para o que precisamos neste trabalho.

3.1.2 Sprites

Existem ferramentas muito úteis para se criar *sprites*, como o Paint.Net, Photoshop, Asesprite. No entanto, para este projeto não serão criadas *sprites* próprias para evitar de se perder muito tempo em desenvolvimento de arte, que apesar de importantes, não é o foco de estudo deste trabalho, e sim a implementação do projeto.

É válido lembrar que aqui são aplicados conceitos já vistos no capítulo anterior como definição da dimensão a ser trabalhada por exemplo. Caso o game a ser criado fosse 3D, ferramentas diferentes seriam utilizadas, como o Blender por exemplo, que comporta a criação de imagens utilizando o eixo Z e uma exploração ampla da geometria analítica para construção de bonecos e cenários.

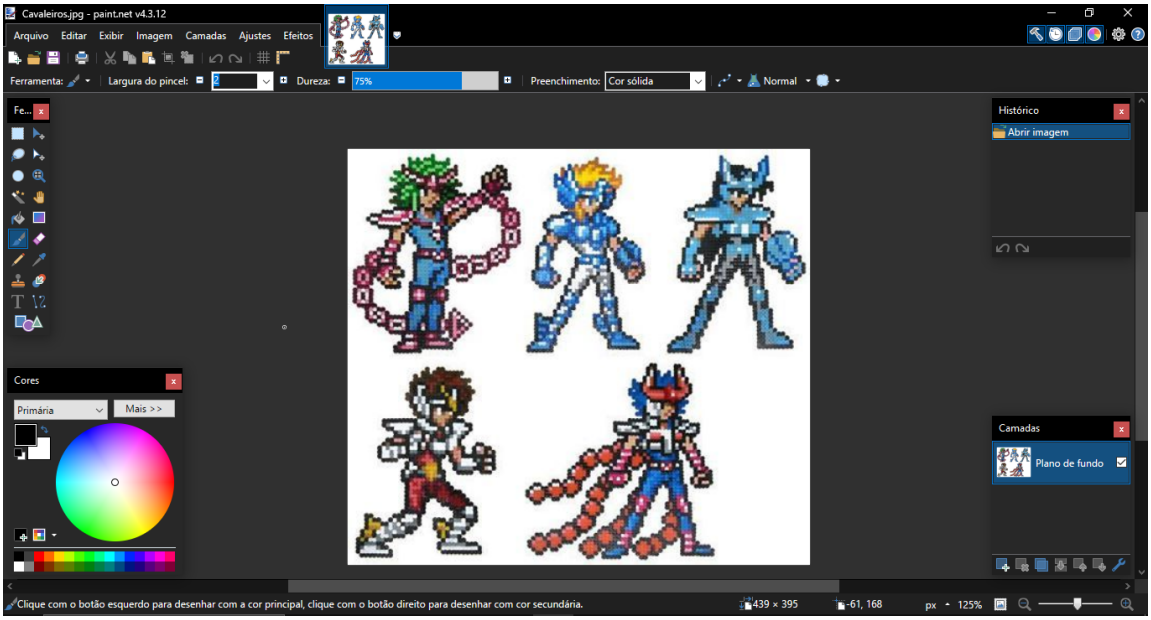

Figura 5: Exemplo de Pixel Art no Paint.NET

O Paint.net e o Asesprite, são ferramentas que possuem um uso já voltado ao 2D, e o que mais agrada nessas ferramentas, é a capacidade de trabalhar pixel a pixel, possibilitando o que é conhecido como Pixel Art. Dessa forma é possível gerar arquivos mais leves e compactos para carregar as imagens, além de que, esse estilo de arte é dono de uma identidade visual extremamente clássica no mundo dos games. Claro que também poderiam ser feitos desenhos de traços maiores, mas o principal fator de uso é esse mesmo.

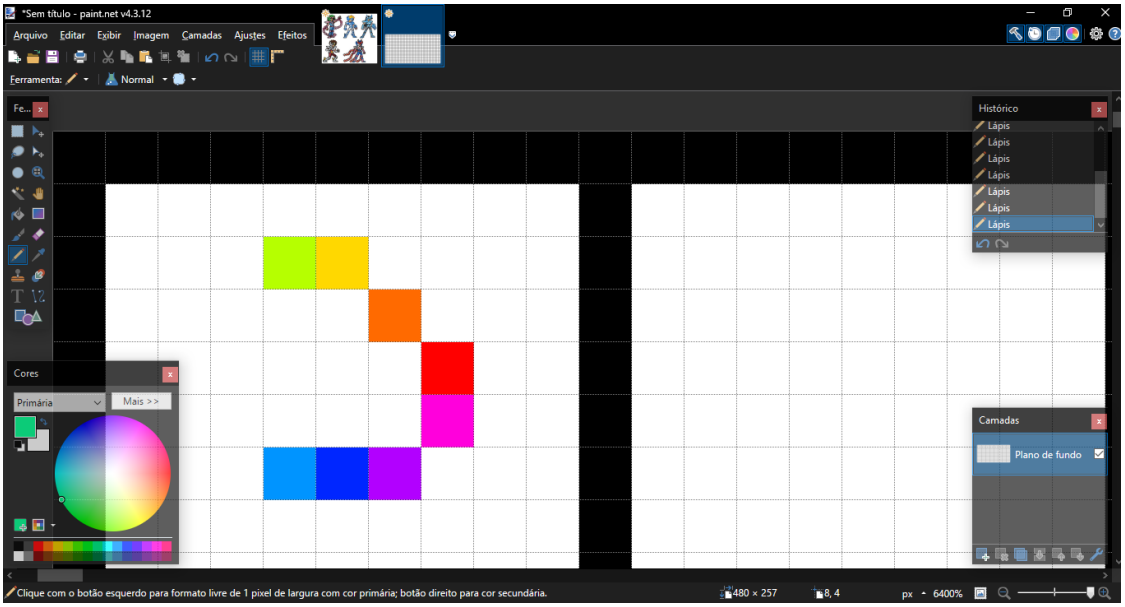

Figura 6: Editando pixeis no Paint.NET

O uso destas ferramentas é bem intuitivo, o usuário pode selecionar um tipo de “material” pode-se assim dizer para iniciar o seu desenho, e a partir disto ele pode rabiscar no quadro o que quer. Para trabalhar com pixeis no caso, o ideal é esboçar uma imagem maior seja no papel ou no computador, e com os quadrados ir aproveitando ao máximo o pequeno espaço para representar a figura planejada, tomando cuidado para não passar do limite estipulado, nem para deixar a figura deformada demais.

3.1.3 Músicas

O som, com certeza, é um fator diferencial dentro dos games, porém como observado, arquivos de áudio tendem a ser pesados, e para jogos mais leves não é interessante inserir faixas inteiras de músicas. Logo, músicas simples e pequenas em *looping*, são uma boa pedida para o desenvolvimento de jogos, e uma ótima ferramenta para se trabalhar com isso é o BeepBox.

O BeepBox é uma ferramenta online, criada para desenhar e compartilhar músicas instrumentais, sendo bem simples e intuitiva em seu uso, apesar de necessitar de conhecimento de teoria musical para se ter um resultado satisfatório (como é de se esperar naturalmente de tal tipo de ferramenta). Ela é totalmente gratuita, contudo, usuários que se sentirem a vontade podem também realizar doações pela criação da ferramenta.

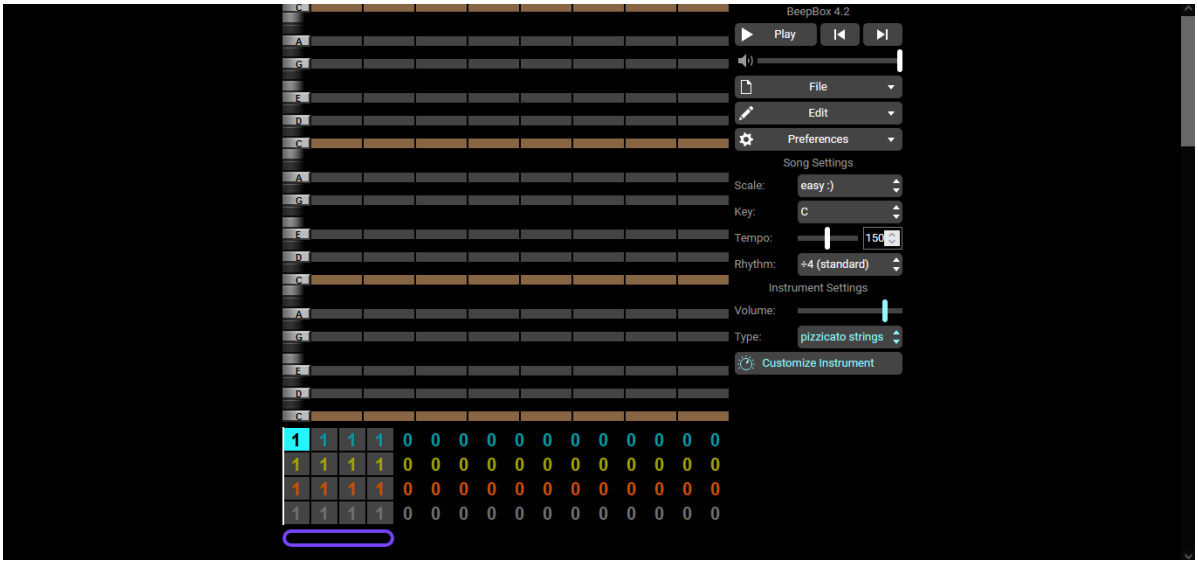

Figura 7: Interface inicial da ferramenta Beepbox

As músicas lá criadas pertencem diretamente aos autores que a produziram, e também não fazem nenhuma espécie de gravação ou redistribuição de músicas, no entanto há uma série de músicas disponibilizadas nos arquivos da comunidade.

O usuário pode selecionar o instrumento que quer reproduzir e as notas são disponibilizadas para ele, então ele pode selecionar o tempo, quais notas ele quer selecionar, o ritmo, a escala de notas, entre diversas variedades de opções, podendo criar desde músicas simples até as mais complexas.

Além de tudo, a ferramenta possui código aberto, notas de atualização quando alguma é lançada, e é reservada pela licença MIT, muito comum em projetos *open source*. Seu criador é John Nesky e o repositório pode ser acessado através do github.

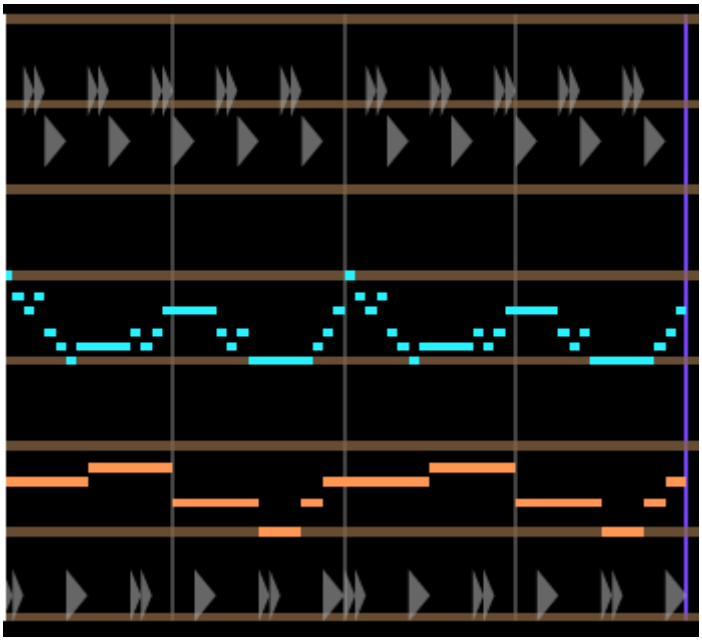

Figura 8: Resultado de uma música criada no Beepbox

3.2 IMPLEMENTAÇÃO DO GAME

A programação do jogo é o grande objetivo deste trabalho, portanto as tecnologias utilizadas nessa etapa são as que levam mais destaque. É válido ressaltar, que já existem diversas ferramentas que agilizam o trabalho do desenvolvedor, que são as *Games Engines*, como a Unity, Unreal e Godot.

No entanto, nenhuma delas serão utilizadas aqui, pois elas já possuem uma abstração de conceitos que serão aplicados dentro deste trabalho e exibidos de forma mais “pura”,

pode-se dizer. Basicamente será mostrado algoritmos que não precisam ser implementados dentro das *engines* pois eles já estão lá.

Tendo levado isso em consideração, segue-se as explicações do porque foi escolhido Java para este desenvolvimento, com suas respectivas funções, facilidades e comparações com outras linguagens de programação que também são utilizadas no mercado.

3.2.1 Linguagem de Programação

É natural que para conseguir criar a implementação do software é necessário algum tipo de linguagem para conseguirmos escrever nossos algoritmos e regras que o jogo receberá. Porém não é tão viável escolher qualquer linguagem de programação para isso, é importante entender a viabilidade dela principalmente por conta da necessidade de trabalhar com baixo nível, acessando recursos de memória, gráficos e áudio.

Durante a história de desenvolvimento de games, vários tipos de linguagens passaram, e o Java foi uma das linguagens optadas, justamente por poder rodar em qualquer tipo de sistema. Seu maior problema no entanto, era a performance, que deixava a desejar, como diz Rabin (2012).

Apesar de existirem outras linguagens que também possuem seu devido valor, como por exemplo o C++, C# e até mesmo Javascript hoje em dia, ainda se tem bons motivos para poder usar Java, mas é claro que essa tomada de decisão também pode ser dada por preferência pessoal, analisando os prós e contras.

Lembrando que o principal motivo para se desenvolver esse projeto diretamente em Java, é para demonstrar o funcionamento de algoritmos essenciais para os jogos e explicar o entendimento da máquina para tais ações.

Caso uma produção real de game estivesse em questão, seria muito mais prático utilizar uma *game engine,* para fazê-lo, pois além de poupar tempo em diversas questões, tal qual um *framework* ou bibliotecas em desenvolvimentos de softwares, a curva de aprendizagem para sua utilização também é mais baixa. Agora vejamos primeiramente alguns detalhes positivos e negativos para a utilização do Java:

Pontos Positivos:

- Mecanismos de serialização de objetos;

- Possibilidade de trabalho com reflexão;
- Permite executar softwares em multiplataformas (JVM);
- Várias bibliotecas existentes para facilitar o desenvolvimento;
- Permite o uso de *threads.*

Pontos Negativos:

- Baixa performance por não poder usar diretamente recursos de hardware;
- Código ainda verboso se comparado com outras linguagens mais modernas;
- Não permite um gerenciamento explícito da memória, deixando o desenvolvedor refém da coleta de lixo automática (*garbage collector*);
- Limitações gráficas 3D.

Como podemos observar, Java possui muitas vantagens em termos de programação, principalmente se comparado a linguagens de mais baixo nível como C/C++ por exemplo, fora o grande destaque da tecnologia, que é a capacidade de poder programar uma vez e executar em quase qualquer plataforma, graças a JVM. No entanto, esse mesmo poder, confere a ele um ponto negativo muito importante para os jogos que possuem um processamento pesado: a performance.

Isso fez de acordo com Rabin (2012), que os jogos em Java fossem mais direcionados para web, de forma que os jogos simples lá encontrados não precisavam de tanto desempenho. Porém as grandes desenvolvedoras não podiam se dar o luxo de perder desempenho em seus games, por isso ele não chegou a cair nas graças de tais estúdios.

É válido também ressaltar, que o Java se manteve no topo das tecnologias utilizadas por muito tempo, recebendo atualizações contínuas. Entre essas atualizações, muitas técnicas de otimização foram acrescentadas a JVM, e também o desenvolvimento em Java hoje permite acesso a código binário nativo, possibilitando o uso do hardware com bibliotecas como OpenGL para gráficos e OpenAL para áudio. Assim, é possível escrever algumas partes de alto desempenho com uma linguagem que compile nativamente e usar esse código através da Java Native Interface (JNI). Contudo, é importante notar que ao usar código nativo, isso significa que a gama de plataformas para quais estão sendo programadas será naturalmente restringida como destaca Rabin (2012).

Outro ponto importante é sobre o g*arbage collector* do Java. Brackeen (2004), cita tal ferramenta como algo positivo, quando está citando sobre as vantagens do Java. Todavia é importante observar, que por mais que o *garbage collector* traga benefícios notáveis para prevenção de vazamentos de memória e redução de erros relacionados a tal, também apresenta alguns problemas para o desenvolvimento de jogos. Ele acaba por operar ao custo de pausas imprevisíveis no programa a fim de recuperar memória não utilizada, leva a fragmentação da memória criando dificuldades em uma alocação eficiente da mesma, o que dificulta muito o desempenho do jogo e como não deixa o desenvolvedor ter um controle direto sobre a memória, pode ocasionar utilizações ineficientes da memória, onde objetos podem ser mantidos por mais tempo que o necessário.

As ações citadas acima, além de limitar o programador, também contribuem para um mau desempenho de um jogo, e por motivos como esse, acredita-se que o Java não é amplamente utilizado no desenvolvimento de jogos.

Com esses pontos em mente, a escolha do Java se dá para este projeto principalmente pela facilidade de ter bibliotecas prontas para poder utilizar recursos gráficos, de áudio, de entrada e saída de dados e recursos para aproveitamento de memória. Dessa forma, podemos lidar diretamente com o funcionamento de algoritmos essenciais, sem ter que entrar em partes de mais baixo nível, lidando com o fluxo principal dos games, além de poder expressar com mais facilidade alguns exemplos de matemática aplicada dentro de bibliotecas já criadas.

3.2.2 Bibliotecas e Ferramentas Auxiliares

Além da linguagem de programação, outras ferramentas auxiliares farão parte de nossa jornada, como as bibliotecas que já possuem métodos e funções prontas para fazer algumas ações que seriam mais complicadas de fazer a baixo nível, portanto mais complexas de explicar e demonstrar neste trabalho como preparar para escutar a entrada de dados via periféricos, ou também ler imagens, abrir um canvas, ler áudio. Além das bibliotecas utilizadas temos também o uso de uma IDE para facilitar nosso trabalho em compilar o código e testá-lo, fora é claro, a possibilidade de contar com um *autocomplete* para códigos, facilitando a escrita da sintaxe.

Neste projeto será utilizado como IDE o Eclipse, que auxilia na estruturação dos arquivos do projeto permitindo a criação de pacotes, que são como as pastas normais de arquivos, e nos permite visualizar de forma hierárquica tendo uma melhor visualização do projeto. Também é importante para facilitar a questão da compilação, como citado antes e escrita de código. Possui extensões dentro de seu ambiente mas não trabalharemos com elas aqui.

Ainda contaremos com o uso de bibliotecas gráficas, como AWT (Abstract Window Toolkit) por exemplo, que possuem classes para facilitar o trabalho que seria feito em mais baixo nível. Uma boa retratação disso é a classe Toolkit em AWT, que possui a capacidade de carregar um arquivo de bitmap, sabendo procurar no caminho do URL (*Uniform Resource Locator*) especificado, utilizando os métodos *getImage()* e *getURL*, como destacado por Harbour (2009).

Assim essas ferramentas que acompanharão este trabalho terão uma boa usabilidade, facilitando tanto a parte de desenvolvimento do projeto para demonstração, quanto a parte de estruturação de arquivos, a fim de não ficar algo confuso de se ler e abstrair.

4. PROTOTIPAÇÃO DO PROJETO

Muito visto dentro da disciplina de engenharia de software, a prototipação é um elemento crucial para o sucesso de diversos projetos nas mais diferentes áreas, e é claro que os jogos não são exceção. Dentro desse capítulo será mostrada as abordagens utilizadas para a prototipação do projeto, definições em cima do que já foi estudado e escolhas sobre como o jogo será, fechando mais o escopo do que teremos no jogo, a fim dele ser um produto viável.

4.1 LEVANTAMENTO DE REQUISITOS

Como o software a ser desenvolvido é um game, os requisitos tendem a ser bem ligados primariamente a questões artísticas, ao qual não será o foco. No entanto, permanece a importância de se ter um mínimo desenvolvimento artístico sobre o projeto, para assim ter a possibilidade de se criar uma visualização das implementações ocorrendo na tela de forma mais clara e objetiva.

Também é importante se pensar em como o jogo será aproveitado pelo usuário, então a abordagem em cima de sua jogabilidade e a inteligência artificial dentro do jogo também é importante.
E por último, é válido também considerar o fluxo dentro do jogo que teremos, isto é: desenvolveremos um jogo focado em história linear com fases para serem passadas, ou um jogo de mundo aberto onde o jogador terá livre acesso a todo mundo desde o início? Isso também afeta a decisão acima inclusive.

Todas essas decisões são pensadas principalmente em função a experiência do jogador, e qual o objetivo que queremos entregar. E como dito no início, essas decisões sobre os requisitos tendem a ser artísticas e de design.

Levando essas questões em consideração, foi feito o seguinte levantamento de requisitos para o desenvolvimento do game de demonstração:

Requisitos relacionados ao mapa e ações de fundo:

- *Sprites* do personagem de demonstração;
- *Sprites* para habilidades e ações do personagem de demonstração;
- Cenário de Fundo;

- Músicas pequenas e que compõe repetição para usar como música de fundo;
- Sistema de menu;
- Sistema de IA para demonstração de diálogos com NPC’s;
- Sistema de IA para inimigos de demonstração.

Requisitos relacionados ao jogador:

- Sistema de movimentação e ação;
- Sistema de habilidade;
- Sistema de colisão;
- Sistema de combate;
- Sistema de mapa;
- Sistema de Inventário.

Ao final o que é esperado, é um game simples, com um menu, um personagem para fins de demonstração que tem capacidade de se mover verticalmente e horizontalmente, usar habilidades, realizar ataques padrão e interagir com o cenário pegando objetos ou falando com NPC’s.

4.2 MAPA MENTAL

O mapa mental serve como um norte para poder separar as ideias para criação do software separando-o em partes.

Como no nosso caso, desenvolveremos uma *game engine* como forma de demonstração do estudo realizado, as características do software estão representadas de forma genérica, de forma a buscar uma exemplificação clara do funcionamento de um jogo, utilizando um modelo já conhecido para poder estabelecer a comparação em termos de programação e prática.

A *engine* é construída a partir de um modelo de jogo em 2D, adotando o estilo “*Top-Down* RPG” (*Role-Playing Game*) encontrado em títulos como Legend of Zelda e Stardew

Valley. A figura 9 exibe um mapa mental que abrange todos os componentes vitais da lógica do jogo. Cada elemento será minuciosamente elucidado nas seções deste capítulo.

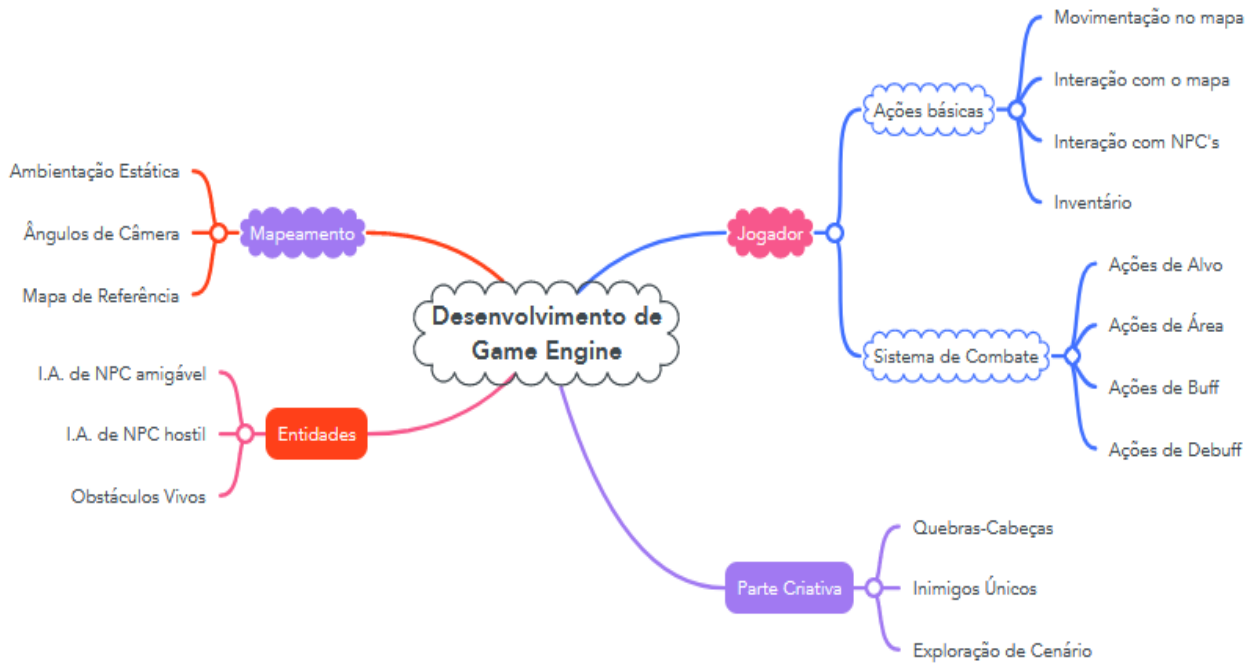

Figura 9: Mapa mental do game de demonstração

4.3 JOGADOR

O jogador é a parte do jogo que conecta o usuário com o universo que está sendo mostrado. É por meio dele que são criadas as interações, e é possibilitada a exploração do que foi criado.

Neste jogo, o jogador será uma peça chave, pois ele será o foco da jogabilidade principal e a ponte por onde o usuário interagirá com todas as outras coisas. Dentro dele temos mais dois aspectos importantes também para se levar em conta.

Por ações básicas, entende-se principalmente as ações cotidianas durante o jogo que estarão presentes durante a maior parte do jogo, sendo elas essenciais para dar vida ao jogador. Ela foi dividida em 4 partes que compõe uma base de interações necessárias para esse tipo de abordagem de jogo.

4.3.1 Movimentação

Ao se pensar em um plano bidimensional (que será o plano do jogo, e um assunto abordado com mais detalhes adiante), nota-se que será necessário o uso de matrizes 2x2 para poder se ter a localização das coordenadas, nada mais do que um plano cartesiano, matematicamente falando. A movimentação do jogador, se baseia justamente em qual coordenada ele está atualmente e qual a próxima, de acordo com o dado de entrada que o usuário inseriu, constituindo um fundamento essencial para a noção de progressão e movimento dentro do game.

Assim construímos uma movimentação em 8 sentidos:

- Horizontal = Esquerda e Direita = 2 sentidos;
- Vertical = Cima e Baixo = 2 sentidos;
- Diagonal = Superior Esquerda/Direita e Inferior Esquerda/Direita = 4 sentidos.

4.3.2 Interação com o mapa

Além de movimentar o jogador pelo mapa, é importante que ele respeite as leis da física, pelo menos a física estabelecida para o jogo, para fazer sentido no que está acontecendo. Sendo assim, a interação com o mapa leva em consideração toda a parte de colisão e reconhecimento do jogador com o cenário ao redor dele.

Seja tomando dano ao colidir com espinhos ou andar mais devagar devido ao terreno ser mais difícil, são bons exemplos da interação da figura do jogador com o mapa.

Além disso, existe também outras coisas no mapa como itens que estarão a deriva para ser recolhidos e passagens que podem ser destruídas. Todo esse tipo de ação é uma relação do jogador com o mapa em que ele está inserido.

4.3.3 Interação com NPC’s

Essa seção é mais simples, no entanto, fundamental. Normalmente são os NPC’s que dão imersão em um jogo, que fazem aquele universo ser vivo. É por meio deles que muitas histórias são contadas, objetivos são dados e itens são recebidos, portanto, pensar em como essa interação será feita é algo importante.

Seja apertando uma tecla, ou fazendo outra ação, é importante que o jogador possa interagir com NPC’s (aliados ou não).

4.3.4 Inventário

Por último em nossas ações básicas, um item muito importante para se incluir nesse escopo é um inventário para gerenciar e manter os itens que são adquiridos dentro do jogo. Claro que é importante também pensar em um limite para o inventário, não só para evitar um cansaço ao ver diversos itens nele, mas também para evitar do jogador ter acesso a todos os itens em qualquer momento.

4.4 SISTEMA DE COMBATE

Um tema clássico que empolga muito em diversos gêneros de jogos é um sistema de combate. Novamente é necessário olhar para o contexto que se quer incluir o projeto, para poder fazer algo que faça sentido, e nesse contexto bidimensional, foi-se pensado um combate de ação livre e também foram separadas 4 tipos de ações para o sistema de combate.

4.4.1 Ações de Alvo

Essas ações se baseiam em acertar um oponente só de forma direta. Elas podem ser articuladas tanto de maneira *point-and-click* (onde clicamos em cima do alvo) como também em uma arquitetura de *skill-shot* (onde apontamos para onde queremos que a ação saia), o importante é que a ação atinja um alvo apenas.

4.4.2 Ações de Área

Como o próprio nome sugere, todas ações que se baseiam em afetar uma determinada área. Isso significa que quando acionada, algo acontecerá em uma área determinada ou selecionada pelo jogador, como por exemplo: invocar um círculo onde quem estiver nele recebe dano em seus pontos de vida ou levantar uma parede em uma linha determinada. Essas ações naturalmente terão interação com o ambiente ao redor, logo, elas terão uma relação direta com o mapa que o jogador está incluso.

4.4.3 Ações de Buff

Buff, se trata de uma fala comumente usada entre os jogos que vem do verbo *to buff,* que por sua vez significa algo como aperfeiçoamento. No mais, pode-se entender *buffs*, como ações que melhoram os status ou características específicas do jogador, oferecendo mais defesa, dano, velocidade ou coisas do tipo. *Buffs* normalmente também podem ser passados para aliados, mas é claro que não faz sentido ser possível “buffar” um inimigo.

4.4.4 Ações de Debuff

Se *Buffs* são aperfeiçoamentos, *Debuffs* são o contrário, causando prejuízos as entidades atingidas. Como exemplos, pensa-se de forma contrária aos *buffs*: menos dano, defesa, lentidão, silenciamento (proibição de usar habilidades por um período), atordoamento (restrição de movimentos) e por aí vai. Existem diferentes formas de *debuffs* sendo até mais extenso que os *buffs*, basicamente então, podemos dizer que se enquadra em qualquer ação que vai lesar de alguma forma a entidade.

4.5 MAPEAMENTO

Os mapas dentro dos games funcionam não somente como plano de fundo, mas também para dar sentido ao que o jogador está fazendo, para onde ele está indo, onde ele está. Os mapas dão contexto, e funcionam para denotar a posição do jogador e de outras entidades no universo do jogo.

Existem várias maneiras de se construir um mapa, e isso vai depender bastante da dimensão em que se é trabalhada. Como este projeto será trabalhado em 2D, será utilizado um mapa que identifica os lugares com cores em pixeis, para representar cada ponto específico do mapa, dessa forma dependendo da cor do pixel, será exibido um *tileset* diferente para o jogador na renderização do game.

4.5.1 Ambientação Estática:

A ambientação no game, será mais um plano de fundo, como observado em muitos games antigos. Não se planeja para este trabalho uma interação forte com o cenário, ou ele tendo uma vida (como árvores balançando ou coisas do gênero), ele servirá apenas

para se ter uma noção de ambiente, também servirá para delimitarmos a área aonde as entidades poderão ir, onde teremos as colisões.

4.5.2 Ângulos de Câmera:

Aqui não teremos muito mistério, a câmera terá um ângulo fixo para poder acompanhar o personagem do jogador. Como já dito, existem diversas formas para resolver as questões de câmera para um jogo, tendo câmeras fixas por ambiente como observado em jogos como Resident Evil**,** ou câmeras que você possui movimento livre, observado em jogos de gênero MOBA como League of Legends**.** No entanto, para jogos *Top-Down* 2D, a câmera presa ao jogador funciona muito bem, exemplos desse tipo de câmera, são encontrados em jogos como Graveyard Keeper**.**

4.5.3 Mapa de Referência:

A orientação do lugar em que está é muito importante quando nos locomovemos na vida real. O mesmo se repete nos games. Muitas vezes o termo “mapa” é usado para referenciar todo o espaço que é possível se locomover, e como os mapas podem ser grandes, médios ou pequenos e até mesmo confusos, é importante ter um mapa de referência, onde o jogador possa localizar onde está para ir andando e achar novos ambientes. Seja com uma seta que se atualiza a posição do jogador no mapa, até um mapa tradicional que só mostra os lugares, já vai ajudar bastante o jogador a explorar o universo construído.

4.6 ENTIDADES

Cada “ser vivo” pode-se assim dizer, dentro de um jogo eletrônico, vai criar uma sensação maior de imersão e vida para o universo criado. Dessa forma, o jogador não tem a impressão de que está sem objetivos, ou sozinho no mundo (mesmo jogando um jogo *single-player,* isto é: para um jogador).

Estes seres que habitam o jogo e interagem com o jogador das mais variadas formas, são as entidades. Elas podem ser reproduzidas tanto como um NPC aliado, que dará itens ou missões, como um inimigo que enfrentará o jogador. Também poderá ser apenas um NPC neutro que está de passagem e nem interage com o jogador, apenas “vivendo” sua rotina programada.

De uma forma ou de outra, as entidades são indispensáveis dentro de um game, pois sem elas, o produto final ficaria vazio e perderia muito de seu potencial imersivo e valioso, já que seu contexto iria se basear apenas no jogador. Desde os primeiros games (como Pong ou Space Invaders) já existiam entidades, que eram presentes em uma I.A. simplificada, agora faria ainda menos sentido tirar elas do processo vital de construção de um game.

4.6.1 I.A. de NPC Amigável

Os NPC's amigáveis tem diferentes tipos de utilidades. Dentro de um game, eles podem ajudar em combate, podem dar itens e até mesmo passar tarefas diferentes para o jogador (e recompensas ao terminar estas). Nesta implementação, a I.A. será feita de uma forma simples, onde nesse tópico, tratamos de entidades que não são hostis ao jogador, isto é: não o ataca, e que podem conversar com ele de alguma forma para demonstrar um pouco esse funcionamento de interação com NPC's.

4.6.2 I.A. de NPC Hostil

Sendo um game onde existe combate, precisamos de NPC's que ofereçam perigo ao jogador. Eles irão atacá-lo de diferentes formas, e é interessante fazer padrões de ataques e animações diferentes. É claro que na implementação de demonstração não será exibido detalhes complexos, até porque muito disso vem da parte artística, com animações e inimigos diferentes. O mais interessante, é ter inimigos especiais como os famosos "*Bosses*" ou "Chefes de fases", que são mais resistentes, possuem uma movimentação única e especial e dano maior do que inimigos comuns. A partir daí é basicamente a criatividade para fazer conteúdos que realmente prendam o jogador.

4.6.3 Obstáculos Vivos

Há certas regiões do mapa, que podem também agir como entidades, mas não fazem de fato parte do plano de fundo. Um exemplo disso são as armadilhas, que fazem de fato parte do ambiente, mas elas são ativadas quando o jogador passa por elas e atingem ele de alguma forma.

Levando isso em consideração, entende-se que esses obstáculos, partes do mapa que acabam interagindo com o jogador, são também entidades vivas, pelo menos em termos

de programação, pois eles possuem um funcionamento separado do mapa tradicional que nada mais é que uma base de fundo (como já dito anteriormente), para dar mais imersão e contexto para quem está jogando.

4.7 PARTE CRIATIVA

Todo jogo possui algum conceito artístico/criativo que torna ele diferente de outros, mesmo que as mecânicas ou implementações sejam extremamente parecidas com outros games já criados. Sendo assim este recurso se apresenta como algo extremamente fundamental para o sucesso de vendas do projeto, pois também carrega consigo o dever de dar ao consumidor a imagem do que o jogo é, e o que ele representa.

No entanto, como não é o foco do presente trabalho, a parte criativa do game será reduzida apenas aos *sprites* e *tilesets* para formar uma representação do que podemos ter em um jogo e suas devidas formas de implementação. Claro que mesmo estes, serão usados de uma forma limitada pois estamos trabalhando com uma vertente de games 2D, que por sua vez ainda possuem vários outros espectros de jogos que podem ser alcançados, mas não há como retratar todos em um só projeto.

Por esse motivo, foi escolhido *sprites* do estilo bidimensional *top-down*, pois é possível explorar diversos tipos de mecânica já implementadas dentro desse gênero, seja com o uso de ações em áreas como habilidades especiais, exploração total do plano cartesiano, perspectiva mais completa de animação (cima, baixo, direita, esquerda), aliado ao fator que podemos explorar uma construção mais tradicional de games.

4.7.1 Quebra-Cabeças

Puzzles são muito bem-vindos dentro dos games, trazendo desafios diferentes para prender a atenção do jogador. E eles podem ser introduzidos de diferentes formas, desde coisas simples como um caça-palavras tradicional, jogo da forca, coisas do tipo, até mesmo projetos mais complexos que dependem de solucionar enigmas com informações e pistas deixadas pelos cenários.

Essa é uma das partes que variam de acordo com a criatividade imposta sobre o jogo e a implementação como de costume dentro do jogo, vai se adequar a cada pedida.

4.7.2 Inimigos Únicos

Quando falamos de NPC's hostis acima, foram citados os "*Bosses*". Pois bem, eles se enquadram exatamente na parte criativa do game, pois o desafio para quem vai criar o jogo, é estar preparando diferentes tipos de movimentos, ataques e defesas. E o limite da forma como o embate se dará é denotado pelos artistas que criam, o papel do desenvolvedor será seguir com a arte criada e o funcionamento pensado para o inimigo e dar vida a eles. Normalmente o desafio em programar esses inimigos é o fato de que não há como generalizá-los, já que precisam ser únicos e memoráveis, então gasta-se uma boa parte do tempo no projeto para esse desenvolvimento.

4.7.3 Exploração de Cenário

Por fim, temos a implementação do contato direto do jogador com o universo do jogo. E a exploração desse universo pode se dar de muitas formas, podendo ser gradativa com áreas fechadas que vão sendo liberadas conforme certos objetivos vão sendo completados, ou o mapa aberto já para o jogador explorar a bel prazer, ou até mesmo com o usuário sendo imposto a passar por alguma situação para poder ter acesso a tal cenário. A forma como o jogador vai ter contato com esse ambiente também é decisão criativa da equipe e o que ele pode ter de contato com o ambiente principalmente, já que mais interatividade significa mais processamento e mais implementação também.

4.8 PROTOTIPAÇÃO DO GAME

Devido à natureza do processo de desenvolvimento de um game ser diferente de um software padrão, a metodologia de criação e planejamento naturalmente se mantém diferente.

Para facilitar e agilizar com a parte visual, serão utilizadas *sprites* e *tilesets* já construídos por alguns artistas, que serão mencionados logo em seguida da apresentação de seus trabalhos nesse projeto.

4.8.1 Características Básicas

Os games tendem a ter algumas características em comuns como uma obra, ou projeto qualquer, que vão assimilar ele a outros tipos de obras. Portanto, a prototipação é um

bom momento para decidir o que será feito de fato, já saindo do mapa mental com as ideias que foram jogadas na caixa e definindo o que será de fato implementado no projeto.

4.8.1.1 Dimensão

A abordagem optada para o desenvolvimento desse projeto será 2D, principalmente pela facilidade de trabalho em relação ao 3D, mas também por exigir menos em termos de poderes gráficos e computacionais.

4.8.1.2 Tipo de Jogo

Como o projeto é apenas uma demonstração de implementação de um game em um estado mais puro, isto é, sem ser em cima de uma *engine* pronta, usaremos como base um estilo de game que apesar de simples pode usufruir de muitos conceitos, então foi um escolhido o estilo *Top-Down* RPG, estilo encontrado em jogos como Stardew Valley e Graveyard Keeper por exemplo.

4.8.1.3 Jogabilidade

Não há como incluir no projeto apresentado neste trabalho, inúmeras mecânicas de jogabilidade devido ao cunho do mesmo. Portanto, serão apresentadas mecânicas que são básicas e muito replicadas nos games como:

- Movimentação (cima, baixo, direita e esquerda);
- Combate (ação de atacar, defender e uso de habilidades/magias);
- Interação com NPC’s amigáveis;
- Interação com objetos do cenário;
- Sistema de Inventário atrelado ao personagem e itens equipados.

4.8.1.4 Mapa

O mapa de um jogo 2D é criado em duas etapas, portanto será criado uma representação em um arquivo menor de onde serão representadas certas estruturas, e em outra representação reconstituiremos esses elementos com artes já prontas, uma estratégia utilizada para não precisar ficar fazendo uma leitura em um arquivo imenso, de imagem.

4.8.1.5 Inteligência Artificial

Por fim, o planejamento da I.A. do game, que acaba variando de entidade para entidade, sendo assim, foi optado por utilizar simples diálogos programados com NPC's amigáveis para fins demonstrativos e inimigos com uma postura simples de combate, apenas para poder demonstrar um pouco as diferentes utilidades dessa característica dentro de um game.

4.8.2 Sprites e Tilesets

A arte do jogador (e até dos NPC's), normalmente é contida em um arquivo só, que reúne todas as posições necessárias para fazer a animação, que são os *spritesheets*. A mesma ideia é reproduzida para o mapa, só que em vez de um *sprisheet*, temos um *tileset*, pois vamos apenas replicar diversas vezes em lugares específicos, a arte que queremos dentro do nosso *tileset*, tornando o jogo mais otimizado por não consumir um arquivo enorme.
No projeto, foram selecionadas artes já feitas para facilitar o trabalho, mas normalmente esse tipo de arte em projetos 2D podem tomar dois tipos de processos:

- *Pixel Art*: Onde cada pixel é valorizado, tendo como resultado arquivos mais leves, onde personagens, por exemplo, tomam um espaço de cerca de 16 x 16 píxeis.
- Desenhos tradicionais: basicamente aqui se incluí outras formas de desenhos que não tem essa valorização, onde no final obtém-se resultados mais complexos e detalhados, no entanto, mais pesados.

Com isso em mente, foi-se procurado *sprites* e *tilesets* que sigam a ideia de *pixel art*, justamente para ficar mais leve o jogo, e também para explorar mais a ideia de como funciona a leitura de imagem pixel a pixel. Nos itens a seguir, serão demonstradas as artes selecionadas para o desenvolvimento deste trabalho.

4.8.2.1 Sprites de Entidades

Uma entidade, normalmente é um boneco representando alguma figura humana ou não, tendo como característica principal, que ela é um objeto vivo no jogo, isto é: interage com o ambiente de alguma forma.

Sendo assim, ela possui diferentes ações como caminhar, correr, atacar, extrair algum item, coisas do tipo, o que automaticamente implica que caso essas ações sejam implementadas, será necessário a criação de uma animação para cada uma delas.

E a animação, ela vai ser composta da mesma figura em posições parecidas, mas formando um movimento. Quando esses desenhos forem unidos em uma implementação que fará o algoritmo percorrer a imagem e mostrá-la como um desenho só, o usuário terá a impressão de que o boneco estará justamente fazendo aquela ação, seja correr, caminhar, atacar, escalar, extrair, ou qualquer outra.

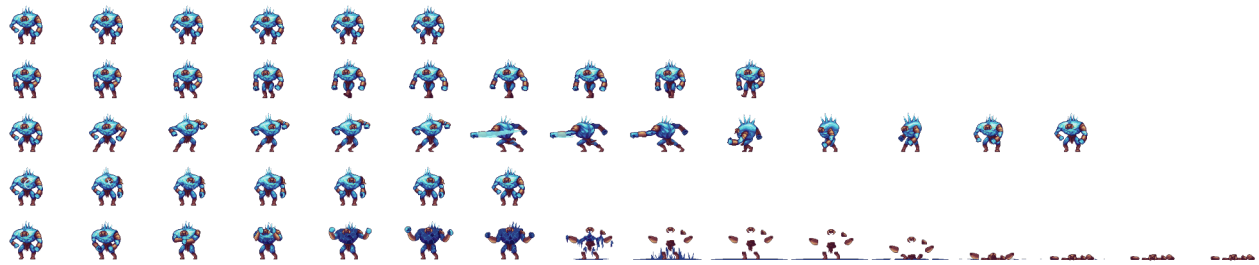

Figura 10: Exemplo de *Spritesheet* com animações alinhadas

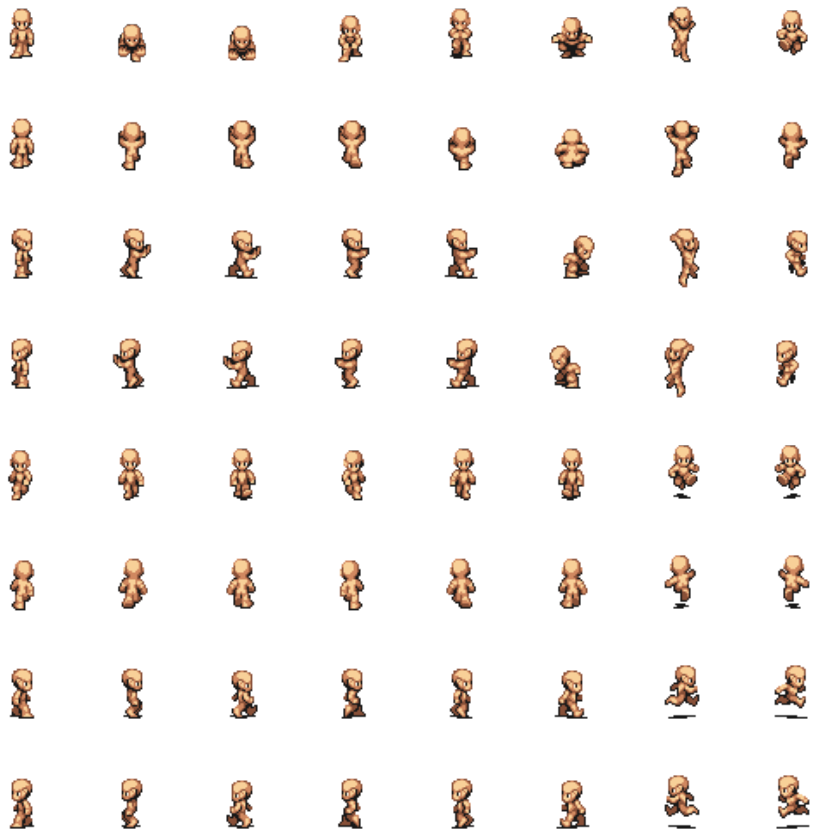

Figura 11: *Spritesheet* de personagem para o jogador

O mais importante nesse momento, é fazer esses desenhos pelo menos de cada personagem em um arquivo só, que será o *spritesheet*, dessa forma além de não ficar tão pesado, nos facilita na hora de implementar, pois só será necessário percorrer o arquivo e passar as posições dos pixeis que são precisos para exibir a animação em questão.

Uma boa prática nesse sentido, é organizar a *spritesheet* de tal forma que seja fácil depois localizar a sequência correta da animação, pois dentro desse arquivo existirá diversas animações relacionadas ao *sprite*, então é importante essa organização para evitar problemas e confusões na hora da implementação.

4.8.2.2 Tilesets

Diferentemente dos *sprites*, os *tilesets* não precisam criar uma animação. Sua principal função é a de preencher um cenário, utilizando de artes que irão formando um conjunto e que podem ser implementadas de forma parecida ao que vemos nos *sprites*. Esse cenário pode ser composto por diversos tipos de formatos: quadrados, retangulares, triangulares, hexagonais, o que importa é ele estar bem definido para se fazer sentido e cumprir seu propósito.

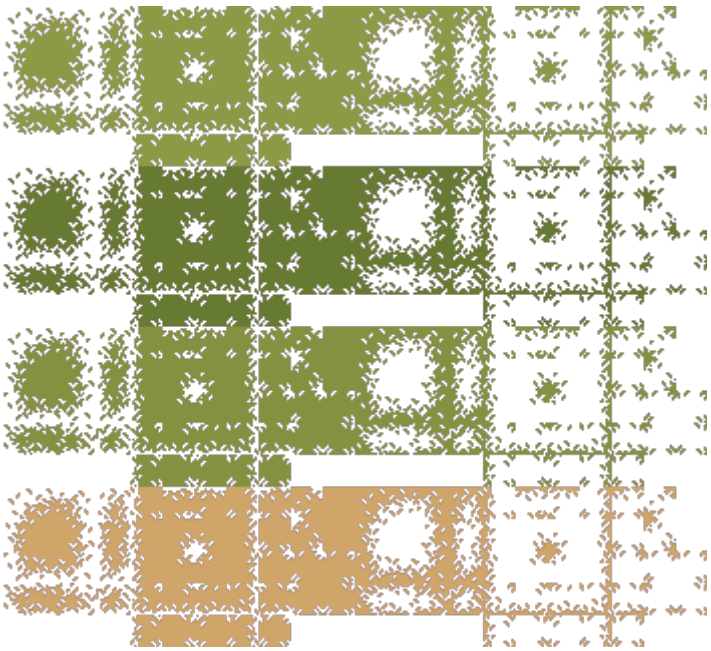

Figura 12: *Tilesets* para cenários de grama e areia

Contudo, cada tipo de *tileset* vai ter uma especificação diferente de acordo com o que ele atende, podendo formar um chão, uma parede, uma escada, um lago… mas dificilmente conseguirá representar uma estrutura inteira, pois o cada *tile* (tijolo), é desenhado pensando em um tipo de estrutura, seja ele uma parede, um chão, ou até mesmo uma escultura.

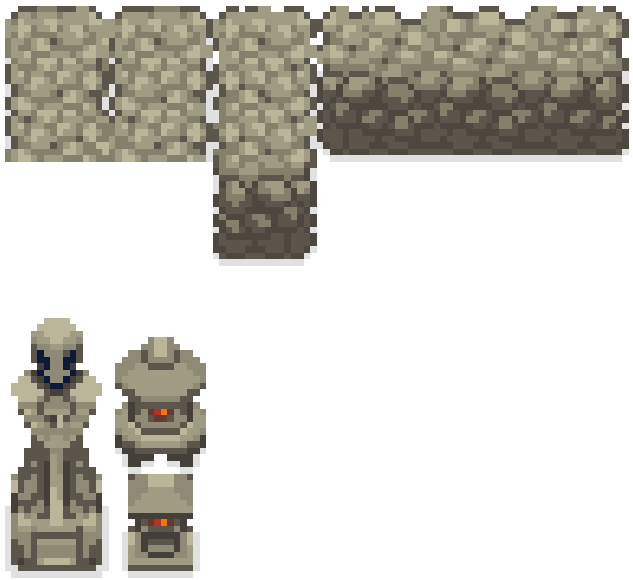

Figura 13: *Tilesets* para paredes e esculturas

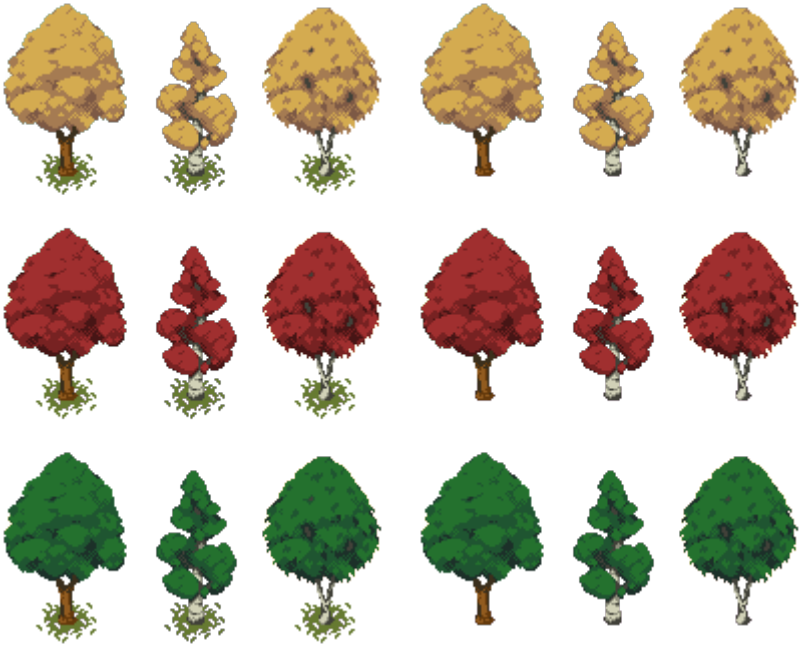

Figura 14: *Tilesets* de árvores

Além de representar essas estruturas, eles podem ser combinados entre si, formando planos de fundo para o jogo ou até mesmo formando entre si estruturas, apesar de ser difícil conseguir representar uma utilizando um agrupamento de *tiles* mas não é impossível.

A implementação futura, utilizará esses desenhos apenas como referência para ir preenchendo o mapa, então existe uma forte possibilidade nesses casos de que não sejam utilizados todos os *tiles* projetados. Tendo isso em mente, é interessante pegar o arquivo novamente e tirar os *tiles* não utilizados para não ficar ocupando espaço no projeto. Não é obrigatório essa parte claro, mas é interessante.

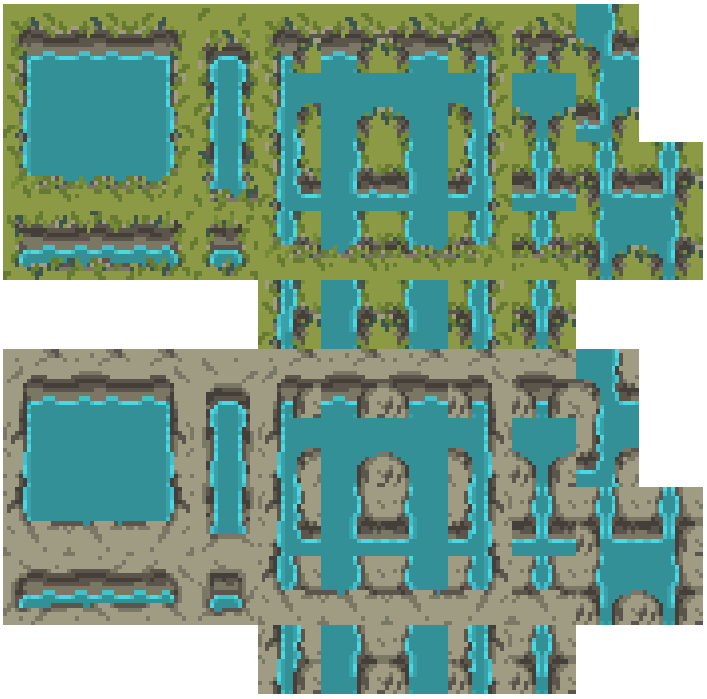

Figura 15: *Tilesets* com a representação de lagos

4.8.3 Áudio: Músicas e outros sons

Um dos elementos mais fundamentais em qualquer jogo é o áudio. Ele realça e dá uma vida forte a experiência, no entanto, sua produção é um verdadeiro desafio. Isso pode ser observado pois se para cada ação é necessário um som, então precisa-se de fazer um trabalho para implementar o som para essas ações, e outro (e mais cansativo) que é de produzir esses sons.

Seja um personagem caminhando, cortando uma árvore ou soltando uma bola de fogo, o áudio precisa ser criado para cada uma dessas ações. O responsável por essa árdua tarefa seria o sonoplasta, mas nesse projeto não será feito tal trabalho, mas vale a citação pois está presente em qualquer jogo produzido.

Agora em relação as músicas, é importante notar que para evitar que muito espaço de armazenamento seja reservado apenas para conseguir guardar as músicas de um jogo, elas possam ser criadas de forma a serem curtas e se possível com uma espécie de conexão para ficarem em *loop.*

Dessa forma é possível criar uma trilha sonora para um jogo, que não consuma tanto armazenamento na máquina de quem for jogar, apesar de gerar um certo trabalho para implementação e criação, mas este é um trabalho que vale a pena.

Uma outra consideração que se pode fazer em relação ao áudio, é que por termos hardwares mais avançados nos dias atuais, também pode-se inserir faixas maiores de músicas e combinações mais complexas, mas se o intuito do seu desenvolvimento é fazer um jogo mais simples, não faz sentido encher ele de músicas enormes pois vai torná-lo mais pesado para o armazenamento. Então essa questão de fazer uma música curta ou seguir normalmente com arquivos grandes, vai também de acordo com a intenção do desenvolvimento do projeto.

5. IMPLEMENTAÇÃO DO PROJETO

Como foi visto até aqui, um game tem várias etapas para ser construído e vários formatos diferentes que vão resultar em propostas bem diferentes entre si. Tendo definido no capítulo anterior o que será desenvolvido como forma de demonstração, neste capítulo será descrito os passos principais para a construção do game e como sua implementação foi realizada.

5.1 ELEMENTOS FUNDAMENTAIS

A primeira parte fundamental para construção do game, é formar os algoritmos base onde as instruções passarão o tempo inteiro. Sendo assim, neste primeiro momento temos a criação de algoritmos “principais” que regem sobre todo o game, e logo após, teremos uma espécie de fragmentação destes algoritmos em outros elementos que fazem parte do game mas que precisam do mesmo comportamento, como por exemplo: atualização e renderização. A seguir, será mostrado qual o funcionamento e a ideia por trás desses algoritmos, e como foi realizada sua construção no projeto de demonstração.

5.1.1 Game Looping

Como um programa que se mantém ativo o tempo inteiro e que só finalizará seu ciclo quando o usuário assim decidir, é necessário criar uma estrutura onde as funções principais aconteçam em tempo real, e estabelecer essa condição para encerrar de fato o programa.

Então, é uma prática já conhecida ter o que se chama de *Game Looping*, que consiste em criar um laço com um *while*, onde dentro deste, serão estabelecidas algumas informações importantes para localização do momento interno do jogo, e posteriormente as duas funções mais importantes do jogo: a de atualização (onde ocorrerá todo processamento interno do que ocorreu de fato para o computador) e a renderização (onde será desenhado na tela para o jogador entender o que está acontecendo).

Tudo isso acontecerá várias e várias vezes em apenas um segundo, e gerando quadros de atualização. A taxa de quadros por segundos renderizados na nossa tela chamamos de FPS (*Frames Per Second* – Quadros Por Segundo).

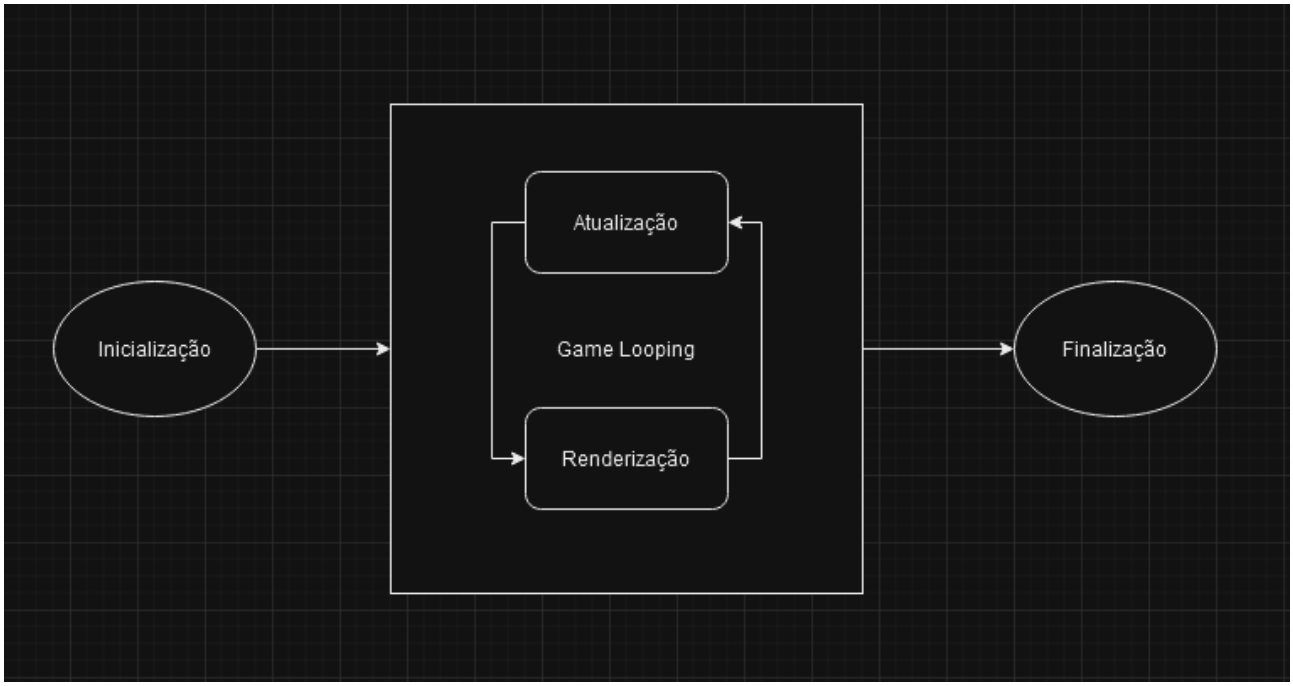

Figura 16: Representação em diagrama do *Game Looping*

Uma coisa interessante de se observar, é que durante a execução, também haverá a entrada e saída de dados, que também alterará no resultado da atualização e renderização, sendo outro detalhe que impacta bastante também na performance dos jogos. Como é necessário que ele faça sua execução em tempo real, *games* mais elaborados tendem a se tornar muito mais pesados para se processar do que o comum, exigindo uma placa gráfica poderosa para realizar esse processo a parte. Por isso ter acesso aos recursos nativos de uma máquina é tão interessante para o desenvolvimento de jogos eletrônicos.

5.1.2 Função de Atualização (tick)

Independente estado em que o jogo estiver, ele sempre verificará o estado atual para saber se houve alguma mudança, e se necessário, processá-la de fato. Seja em um menu ou dentro de jogo, com algum tipo de entrada por parte do jogador ou não, essa função sempre estará sendo executada de acordo com o estado atual do jogo.

Quando o jogador executa uma ação de movimento por exemplo, a função de atualização do jogador fará o cálculo da posição dele na matriz com a adição ou subtração dependendo de como foi esse movimento dele (o que será descrito com mais detalhes no

capítulo 5.3.1) do valor relacionado a sua velocidade, e então, será entregue o resultado com a nova posição do jogador, um valor atualizado.

Este tipo de ação ocorre o tempo inteiro em um jogo, mas ele não ficará esperando o usuário passar algum valor de entrada, ele irá naturalmente realizar o que deve fazer de acordo com o estado em que se encontra. Nos menus, normalmente não há muito o que se fazer, mas dentro de jogo, normalmente temos inimigos andando, querendo nos atacar, itens com um tempo limite para existir, ou até mesmo efeitos com tempo limite, que ao esgotar, sumirão do jogo até serem utilizados novamente. Todas essas coisas acontecem internamente sob o controle de uma função de atualização.

Essa função vai ser encontrada em várias partes do software, seja na classe principal, de um jogador, de um inimigo, de um efeito, pois para ser computada, ela será chamada pelo método de atualização principal que fica dentro do *Game Looping*, que por sua vez chamará os métodos de atualização que são necessários de acordo com o estado atual em que o jogo se encontra.

Alguns usos mais inteligentes de arquitetura para jogos, é criar interfaces ou classes para serem herdadas, onde eles já possuam esse método. Depois a classe que implementa ou herda, poderá utilizar esse método novamente sem ter que ficar o reescrevendo toda hora. Contudo, seria mais interessante o uso de interfaces para poder implementar o método de atualização da forma que lhe for conveniente de acordo com a classe, visto que teremos diferentes classes com comportamentos diversos.

E por final, é importante se atentar que nessa função se concentrará cálculos e atribuições de valores que são referência para o jogo, então é natural que existam funções de atualização maiores dentro da implementação do software, mas nada impede que se criem outras funções privadas para realizar diversas ações e depois apenas chamá-las nessa função de atualização, resultando em uma melhor organização do código.

5.1.3 Função de Renderização

Tal qual a atualização, a função de renderização é uma parte essencial do desenvolvimento de jogos, pois o resultado do que foi atualizado internamente, será desenhado na tela por ela.

A ideia principal do funcionamento desta função é muito parecida com a de atualização, o que muda é apenas o seu propósito. Também teremos diversas funções entre as classes, podendo utilizar interfaces para implementação ou classes para esquemas de herança, todavia, é importante relembrar que o uso de interfaces acaba por ser mais interessante por obrigar a presença do método na classe que o implementa, mas dando a possibilidade de reescrevê-lo de forma independente em cada classe.

O desenho das informações na tela, vai acontecer utilizando os conceitos que vimos no capítulo 2.1.2, de matrizes e geometria, estudando as melhores maneiras de representar uma forma, via figuras geométricas. Em um jogo 2D como no nosso caso, a implementação de uma função de renderização, consistirá percorrer o arquivo que disponibilizamos como uma matriz 2D. A forma como ele fará isso, será descrito mais a frente, tanto para o mapa quanto para entidades.

Um outro adendo interessante sobre a função de renderização, é que sua construção é resultado das chamadas das funções de atualização, o que significa que ela jamais pode ser executada antes de uma atualização, e sim, após esta. Dessa forma, se garante que as posições, cálculos e outras medidas internas que precisam ser feitas estejam prontas e justamente atualizadas para serem desenhadas na tela. Se o contrário fosse feito, teríamos um problema, pois todo desenho na tela estaria atrasado em relação ao que aconteceu, pois ele só percorreria a renderização antes da atualização, causando um pequeno *delay* no processo da informação.

A implementação dessa função exige o uso de bibliotecas gráficas, neste projeto, foi utilizado o *java.awt.Graphics*, que fará todo o trabalho de poder percorrer uma imagem e desenhá-la no *Canvas,* e sempre veremos essa função junto a de atualização nas classes em que são implementadas, pois tudo que é atualizado, tende a ser renderizado.

5.1.4 Estabelecimento do Quadro do Game (Canvas)

Na seção anterior foi falado um pouco sobre o *Canvas*. E como se nota, ele realmente é um quadro, assim como sua denotação na vida real, onde uma pintura será feita.

É dentro deste quadro que o jogo será exibido, pois tudo que é desenhado, é criado dentro dele. Como de praxe em softwares desktop, o *Canvas* também tende a ser utilizado para poder desenhar o programa, disponibilizando interfaces, menus, onde o usuário navega, a grande diferença aqui, é que os desenhos não são estáticos, eles são carregados na memória e mudam de posição de acordo com a programação interna, e ainda por cima realizam isso com uma atualização do desenho (afinal, ficaria muito estranho se apenas mudasse da posição original sem nenhuma animação).

Contudo, essa função de representar o que está dentro como vimos, não é do *Canvas* e sim da função de renderização. O quadro receberá apenas os valores relacionados a sua configuração como o tamanho, permissões de redimensionamento, alteração de posição e visibilidade. É comum entre os jogos deixarem também o jogador selecionar o que é melhor para ele dentro de um menu de opções, mas neste projeto não faremos tal implementação, já que sabemos as configurações exatas que se precisam.

Essas configurações serão passadas ao início do programa e antes de entrar no *Game Looping*, pois assim se garante que ele não ficará abrindo janelas, várias e várias vezes. É interessante também a criação de uma classe para tratar as opções relacionadas ao quadro, a ideia seria dentro dessa classe, utilizar a chamada dessa função de inicialização do quadro porém, passando valores diferentes do padrão, onde o usuário escolhe (o ideal é deixar opções listadas e limitadas para não se quebrar o *Canvas*), quais são as melhores opções para ele.

5.1.5 Definição de Estados do Jogo

Dentro da arquitetura de um *game* é muito comum se ter uma variável para se controlar os estados em que o jogo se encontra (podendo ser até mesmo um Enum), e como pode ser observado, já foi citado algumas vezes sobre o estado do jogo, mas como se percebe ele não uma função, e sim uma variável. E isso ocorre, porque é justamente através do fluxo do jogo que essa variável agirá (tendo seu valor alterado conforme a necessidade),

conduzindo quais funções de atualização e renderização estarão sendo ativadas por meio de validações do seu valor atual.

Na prática, isso significa que quando um usuário seleciona em um menu por exemplo “Começar jogo”, o estado do jogo irá de “Menu”, para “Jogo”, ativando as classes referentes para iniciá-lo (Mapa com estruturas, caminhos, objetos, entidades e tudo mais). Se no meio do jogo o usuário resolver pausar, o estado irá de “Jogo” para “Pausa”, congelando o processamento do que acontece no jogo e deixando em um estado parecido com o menu, mas ao retomar, as entidades ainda ficam carregadas.

Essa abordagem sobre os estados é interessante, pois assim temos a possibilidade de criar diferentes cenas para o jogo também (como se fosse um filme mesmo), poupando processamento de um mapa inteiro por exemplo, ou de deixar diversos objetos vivos na memória, além de também fornecer uma estrutura concisa sobre o fluxo do jogo.

Neste projeto foi optado por deixar a variável de estado como uma *String* mesmo, recebendo um texto. É uma opção válida, uma vez que elas podem ser facilmente validadas também, mas é interessante como mencionado anteriormente o uso de Enum para esse tipo de estrutura também, ou inteiros, enfim. O que o desenvolvedor achar melhor e que dê para listar vários tipos de estados de forma que fique claro para ele o que cada um representa é válido.

5.1.6 Definição e Estabelecimento de Entidades

Eis aqui um ponto crítico para boa parte dos jogos: as entidades. Poucos são os gêneros onde não se é preciso o estabelecimento desse tipo de abordagem, pois as entidades estão para um *game*, tal qual o ser vivo está para a vida real podemos dizer. Isso é dito, baseado na função de uma entidade: representar o jogador e os NPC’s.

Isso significa que todo “boneco” ou representação de um robô amigo ou inimigo dentro do jogo, tem como sua configuração base que ele é uma entidade, inclusive o próprio jogador.

As entidades também recebem informações básicas que é compartilhado entre toda classe que é uma entidade. Toda entidade vai ter essas usabilidades dentro dela:

- Posição dentro do mapa (x,y);
- Imagem para se carregar (*spritesheet*);

- Largura e Altura (que será usado para definir o tamanho da entidade).

Com esses itens estando definidos dentro da classe de entidades, se constrói a possibilidade de criar classes para o jogador e inimigos que herdem a classe Entidade, e assim podemos não só evitar a reescrita de código, como estruturar melhor também o nosso software, uma vez que essas características estarão presentes em todas as classes que estão herdando Entidade. Apenas é importante tomar cuidado para utilizá-la nos lugares corretos e evitar problemas de estruturação de código.

5.2 IMPLEMENTAÇÃO DE SPRITES E TILESETS

Uma vez que se foi implementado a principal parte do jogo, agora é necessário preparar uma outra parte que está mais relacionada a visualização do jogo, que são os *sprites* e *tilesets*.

Isso significa, que é a hora de pegar os arquivos que foram preparados na parte da prototipação e incluir eles de fato dentro do *game,* além de preparar também a parte de como ele será lido.

Existem algumas abordagens diferentes em relação a como implementar essa parte, no entanto como foi planejado no capítulo 4, será abordado algumas tratativas em relação ao 2D principalmente, mas é válido citar que dentro de uma abordagem 3D, as *sprites* iriam dar o seu lugar a modelos 3D que são formados por diversos polígonos. Após isso, texturas iriam se incorporar aos modelos, tornando a figura algo menos abstrato e que o jogador possa reconhecer de fato como algo do universo em que ele está inserido.

5.2.1 Tilesets e Construção do Mapa

Para viabilizar a implementação dos *tilesets*, uma boa forma de abordagem, é a construção de uma representação de mapa em um pequeno arquivo com cores que em cada pixel que representarão diferentes objetos no mapa. Dessa forma é possível representar os *tilesets* que serão incluídos no mapa e futuramente colocar algum tipo de lógica como colisão dentro dele de uma maneira fácil.

Outro uso interessante dessa abordagem, é aproveitar que os pixeis serão usados como referência para se implementar os *tilesets* e utilizar também para localizar e implementar as entidades como *players* e inimigos e até mesmo para a localização de itens.

Como a construção desse mapa é criada somente para ter uma referência, ele não será um arquivo enorme, muito pelo contrário, ele pode ser construído com poucos bytes pois utilizará poucos pixeis (já que cada pixel representa uma parte muito maior no C*anvas*). Após a construção do mapa, cada pixel contido nele será lido, de forma que dependendo da cor utilizada, alguma arte seleciona no *tileset* ou algum tipo de personagem aparecerá. Por exemplo: onde tem azul-escuro será o *Spawn* do personagem do jogador, onde é preto é o chão e onde é branco será desenhada uma parede.

Com isso temos a construção de um algoritmo que fará uma busca por esse mapa de referência e passará a informação de como deverá ser feita a renderização do mapa de forma bem simples e organizada, com o adendo, de que essas cores devem estar muito bem separadas e organizadas no projeto. Depois no código, basta pegar o valor hexadecimal da cor e referenciá-la dessa forma para fazer a validação sem riscos de correr erros (normalmente utilizando uma ferramenta de conta-gotas tem como se obter esse valor, é assim que funciona no Paint.NET por exemplo).

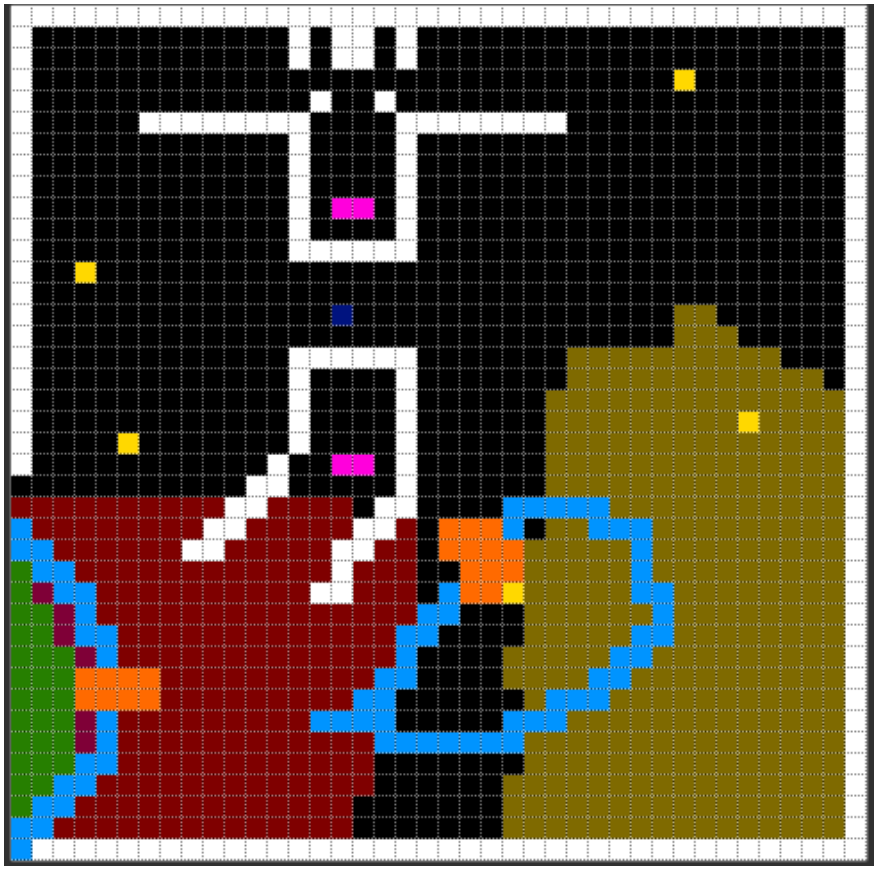

Figura 17: Mapa de representação para implementação de *tilesets*

Também existe a opção de ir renderizando um mapa diretamente, já desenhando ele por inteiro e lendo a estrutura dele, porém é um processo muito mais pesado e complexo para implementação, uma vez que a complexidade de cores e desenhos contidas nesse tipo de mapa dificulta entender o que é uma estrutura, o que é um chão e o que é um item, o que resulta também em uma posterior dificuldade de implementação de colisões e regras de física no universo do jogo.

5.2.2 Spritesheets de entidades

A implementação dos s*pritesheets* das entidades não costuma ter muito segredo, pois elas basicamente vão se estruturar em cima de conceitos base de animação, que é de repetir a mesma figura com leves alterações para que elas formem em conjunto uma sensação de movimentação.

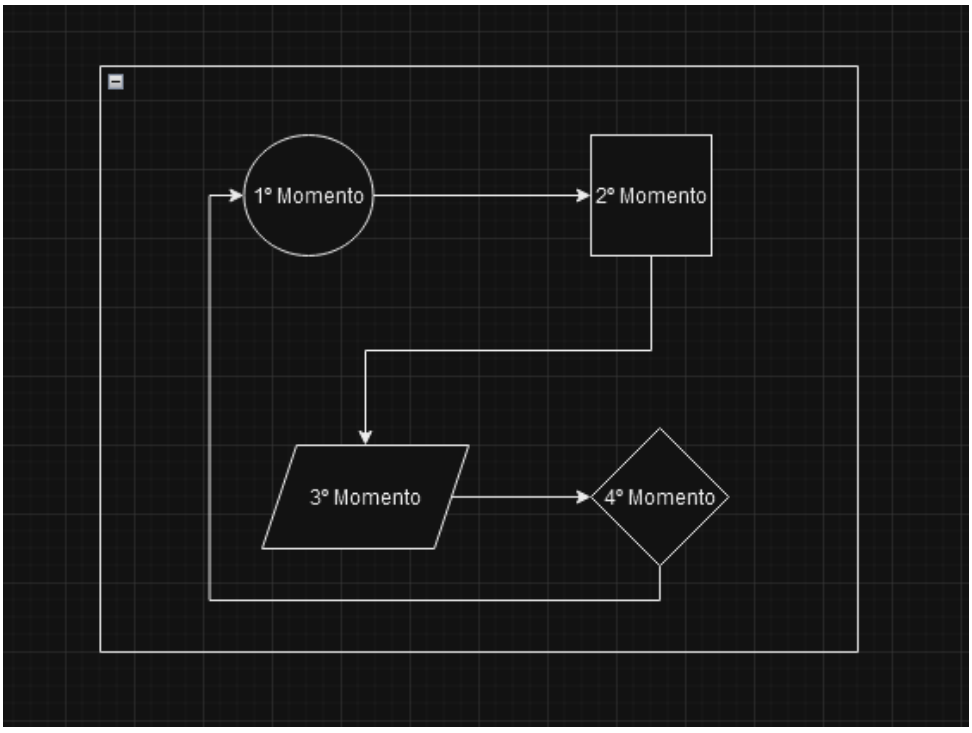

Figura 18: Diagrama representando laço para animação

Neste caso, o ideal é pegar o *spritesheet* que se tem, e verificar quais são as coordenadas de início e fim para cada imagem. Após isso, criar um *looping* para quando as condições de movimentação forem atingidas (ou de qualquer outra animação como

ataque, extração de item, uso de habilidade), começar a percorrer o arquivo com as medidas que formam a figura utilizada. Caso aconteça de chegar até o final da animação e ainda necessitar continuar como em uma ação de movimento, ele apenas retorna ao início e continua novamente até a condição estabelecida ser negada.

Como se pode ver na figura 16, ao chegar no último momento, não necessariamente se encerra o ciclo, apenas zera-se o índice e volta ao início. As condições normalmente são ditas de acordo com o planejamento da ação. Como exemplos podemos citar a ação de movimentação. Enquanto o usuário pressiona a tecla “D”, o personagem irá se mover a direita, logo, quando chegar no 4º momento da animação, ele não parará reproduzir a figura, apenas voltará ao início, dando a impressão de uma movimentação contínua.

Todavia, isso não é uma regra para todo tipo de animação. Em um ataque por exemplo ou uma habilidade, é interessante que mesmo com o usuário apertando várias vezes o botão para tal ação, a animação e sua atualização só ocorra uma vez, enquanto o jogador só vai poder executá-la novamente quando ela acabar.

E a mesma ideia se aplica aos robôs que estarão presentes no *game*, com o diferencial de que o acionamento de suas ações podem ser tanto aleatório, conforme a decisão de uma pequena implementação de I.A. ou até mesmo ser programado para em situações específicas realizar tais ações.

Além de todas essas questões de localidade no mapa, é importante realizar também os cálculos referentes ao resultado de uma ação. Por exemplo: se o jogador realiza um ataque contra um inimigo, primeiro ele deve verificar se o ataque acertou ou não, e caso positivo, calcular o dano que o inimigo recebeu para perder a vida. Além das questões de animação relacionados a renderização há também as questões de atualização interna, e elas se darão pelo contato entre duas entidades diferentes em casos como esse exemplo, ou apenas em na própria entidade em questão caso não aja uma espécie de interação com outras entidades.

5.3 IMPLEMENTAÇÃO DE JOGABILIDADE

Talvez uma das partes mais importantes pois é o que dará a imersão para o usuário, seja também uma das mais simples para se implementar. Na verdade se olhar um pouco mais a fundo, existe toda uma questão complexa que é a de observação de dados de entrada, isto é, perceber se o usuário está informando alguma coisa para o programa. Porém, essa

questão é facilitada já em Java por exemplo por meio da interface *Key Listener*, que pode ser encontrada na biblioteca java.awt.event.

A função desse tipo de implementação é observar os periféricos do computador pelo Sistema Operacional, e caso algum dado seja passo ele entra em ação. A interface não só faz esse trabalho de observação como também possui métodos já prontos para quando uma tecla for pressionada, solta ou digitada, inclusive nos dando o poder de acessar variáveis que confirmam de fato qual tecla é por meio do *KeyEvent*.

Tendo isso em mente, a seguir será comentado como podemos aproveitar o uso de tais ferramentas e lógica para implementar a jogabilidade para o jogador e as outras entidades.

5.3.1 Implementação do Jogador

A implementação do funcionamento do jogador, está totalmente interligada aos periféricos citados anteriormente. Isso se deve ao fato de que para jogar de fato, o usuário deve utilizar meios para poder passar os comandos até o boneco do jogador, que fará as ações de acordo com esses comandos.

Os comandos serão previamente especificados pelo desenvolvedor, e como de praxe temos a movimentação do jogador (onde é muito comum utilizar as setas do teclado ou as teclas “W”, “A”, “S” e “D”), onde nesse projeto o jogador pode ir para as direções cima, baixo, esquerda, direita e diagonais. Se o jogador terá algum limite por conta de perspectiva ou algo do tipo, fica a critério da prototipação como sempre.

No projeto foi aproveitado a biblioteca para fazer a implementação da movimentação e outras ações, mas a ideia principal é que enquanto o jogador está pressionado uma tecla, ele irá se movimentar para aquela direção, logo, ao manter tal tecla pressionada ele alterará um valor na classe do jogador, fazendo com que sua atualização calcule sua nova posição e ele seja animado andando na direção apontada.

Por exemplo: enquanto o jogador pressiona a tecla “S”, ele irá para baixo, alterando o valor de Y positivamente (nesse plano o Y positivo está para baixo e o negativo para cima, mas o X mantém-se positivo a direita e negativo a esquerda), e animando a movimentação do jogador se movendo em tal direção. Assim que ele parar de pressionar a tecla, isto é, ao soltá-la, o boneco não se moverá mais.

Isso implica que para toda ação que o jogador for fazer, antes existirá um sinal vindo do *KeyListener*, para saber se de fato o usuário está passando alguma entrada válida. Então a mesma ideia irá se replicar para qualquer outra coisa que ele fizer, desde falar com NPC's, usar habilidades, atacar, até abrir o inventário e consumir um item. As animações e ações serão executadas a medida que elas são chamadas, tornando uma implementação até mais intuitiva de se realizar do que a de outras entidades que compõe o universo do jogo.

5.3.2 Implementação de NPC's Hostis

A implementação de inimigos é algo mais complexo, pois é necessário construir uma espécie de inteligência artificial já que essa entidade agirá por conta própria.

Existem várias formas de se implementar esse tipo de comportamento, mas para fazer um bom filtro foi escolhido 3 para se falar a respeito.

A primeira forma é fazer o NPC hostil apenas correr atrás do jogador, e caso ele encontre-o fará algum ataque a ele (seja a distância ou colado). Essa implementação terá uma mentalidade apenas de "mate o jogador", não se importando qual o melhor caminho, ou se organizando com outras entidades do mesmo tipo a fim de cercar o jogador para complicar sua vida, mas pode ser realmente complicadas de lidar quando tiver uma boa quantidade atrás do jogador, ou inserindo quantidades altas de dano.

A segunda forma é programando decisões específicas e aleatórias, para quando o jogador entrar em uma área de contato com a entidade, ela fazer alguma ação hostil a ele porém aleatória, tornando o combate mais imprevisível e desafiador. Contudo é importante ressaltar que fazer vários tipos de ataques diferentes e torná-los aleatórios, além de se tornar cansativo pode quebrar um pouco o combate para o jogador, que deverá adivinhar qual o próximo movimento.

A terceira forma, é programando manualmente como o esse NPC irá se comportar de acordo com o momento do combate, configurando ataques e movimentações únicas a fim de tornar tal encontro único também. É uma boa abordagem para lidar com personagens especiais, mas é a mais trabalhosa das três, apesar de tornar pelo trabalho a jogabilidade muito mais interessante.

5.3.3 Implementação de NPC's Amigáveis

Um NPC amigável pode servir para várias coisas, inclusive ajudar em combate, com uma implementação similiar aos NPC's hostis, com a diferença de que eles atacarão qualquer entidade hostil ao *player*.

No entanto, também é possível utilizar esses NPC's para outro tipo de função, como conversar para dar algum tipo de informação ou missão, dar algum item, ou até mesmo curar o jogador.

Não é necessário ir muito longe aqui, pois apesar de importantes, a implementação é de fato parecida a seção anterior, podendo apenas ter esse adendo comentado.

Para fazer uma implementação de fala ou de uso, é bem simples, basta colocar que ao entrar em contato com o jogador por meio de uma ação (como o jogador chegar do lado do NPC e apertar a tecla "E"), ele executa sua função, que pode ser qualquer uma das citadas acima.

Uma outra possibilidade seria de criar apenas esses NPC's para ficarem perambulando aleatoriamente pelo mapa, fazendo ele trocar de direção a cada 10 segundos por exemplo, ou até mesmo, tracejando uma rota específica para ele cumprir, são ideias simples mas práticas de se implementar e funcionam bem.

5.4 IMPLEMENTAÇÃO DE ÁUDIO DENTRO DO JOGO

Essa é a última forma de implementação que será abordada neste trabalho, e é bem única. Isso se deve porque o áudio ele é muito importante para dar vida a um jogo, ao mesmo passo, que tende a criar arquivos extremamente pesados. Por isso as implementações mais antigas, criavam pequenas músicas e as colocavam em *loop.* Assim se evitava o consumo de muito espaço de armazenamento.

A implementação do áudio, consiste em ler um arquivo de áudio e reproduzi-lo quando necessário. Uma música pode ficar tocando de acordo com a *gameplay*, e um som de caminhada, de ataque vai ser reproduzido de acordo com a situação. O que é mais complexo nisso tudo seria de fato conseguir a leitura desse arquivo de áudio.

Para realizar essa tarefa no projeto desenvolvido, foi utilizado um recurso do Java, que como mencionado nas seções anteriores, já possui bibliotecas que permitem esse tipo de manipulação. Nesse caso, foi utilizado o javax.sound.sampled e o java.io.

Com o uso dessas bibliotecas, a parte mais baixo nível que seria ler os bytes do arquivo e reproduzir o som, fica simplificada, e a partir disso basta carregar os arquivos de áudio separados para o processo e escrever a lógica de qual momento deverá ser tocado, como por exemplo, ao iniciar o jogo ou trocar de fase.

6. CONCLUSÃO

Neste trabalho, foi explorado sobre como os jogos funcionam para um computador, citando exemplos de matemática aplicada aos *games* e também demonstrando a utilização da Teoria da Computação e seus modelos matemáticos para facilitar o entendimento do processo de um software como um todo e sua relação natural com a matemática.

Apesar de não ter como abordar esse assunto de forma muito profunda devido a sua complexidade natural, essa pesquisa já funciona como uma forma de esclarecer brevemente como que os jogos funcionam em sua virtude computacional, mas para poder abstrair esses conhecimentos de uma forma mais concisa, também foi abordado a implementação de um trabalho prático, utilizando-se de uma forma mais “antiga” pode-se assim dizer de desenvolvimento que é apenas uma linguagem de programação em seu estado natural, que foi o Java, e não uma *game engine* que já possui várias ferramentas voltadas a construção do jogo e que abstraem para si alguns detalhes mais complexos.

Sendo assim, se pôde demonstrar também todo um processo de desenvolvimento que inclui desde as escolhas de tecnologias a serem utilizadas e prototipação do projeto, até sua implementação de fato. Apesar de ser comum essa abordagem no desenvolvimento de diversos softwares, a produção de um *game* precisa respeitar as peculiaridades e objetivos da obra em questão.

E como perspectiva de contribuição, é esperado que aqueles que lerem este trabalho, possam não só compreender melhor a forma natural como um jogo é desenvolvido, como viabilizar uma pesquisa mais profunda, destacando conceitos matemáticos e computacionais que possam melhorar a técnica para produção de jogos eletrônicos.

6. REFERÊNCIAS

BARBOZA, Eduardo Fernando Uliana. **A evolução tecnológica dos jogos eletrônicos: do videogame para o newsgame**. 2014. 16p. 5º Simpósio Internacional de Ciberjornalismo – Universidade Federal do Mato Grosso do Sul, Campo Grande, 2014.

BRACKEEN, David; BARKER Bret; VANHELSUWÉ, Laurence. **Developing games in Java** – New Riders, 2004.

CAPCOM. **RE: Engine**. Osaka. Disponível em <https://en.wikipedia.org/wiki/RE_Engine>. Acesso em 07 jun.2022.

DIVERIO, Tiarajú Asmuz. **Teoria da Computação – Máquinas Universais e Computabilidade**: Volume 5. Bookman, 2011.

EPIC GAMES. **Unreal Engine**. Cary. Disponível em <https://www.unrealengine.com/en-US>. Acesso em: 07 jun. 2022.

HARBOUR, Jonathan S. **Programação de games com Java** – Tradução da 2ª edição norte-americana. Cengage Learning, Brasil, 2009.

PITTOL, Gabriel Luís Duarte. **A história e contribuição dos jogos e consoles de videogames para a sociedade e a computação**. 2019. 72p. Monografia – Instituto de Informática – Universidade Federal do Rio Grande do Sul, Porto Alegre, 2019.

RABIN, Steve. **Introdução ao Desenvolvimento de Games** – Volume 2 – Programação: técnica, linguagem e arquitetura – Tradução da 2ª edição norte-americana. Cengage Learning, Brasil, 2012.

TONÉIS, Cristiano N. **Matemática Aplicada aos Games**. Clube de Autores, 2016.

UNITY TECHNOLOGIES. **Unity**. São Francisco. Disponível em <https://unity.com/pt>. Acesso em 07 jun. 2022.