



**UNIVERSIDADE PAULISTA – UNIP
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS - ICET
CURSO DE CIÊNCIA DA COMPUTAÇÃO
CAMPUS MANAUS**

**DESENVOLVIMENTO DE UM JOGO IMPLEMENTANDO OS CONCEITOS DE
ORIENTAÇÃO A OBJETOS UTILIZANDO A LINGUAGEM JAVA**

MANAUS-AM

2022

**UNIVERSIDADE PAULISTA – UNIP
INSTITUTO DE CIÊNCIAS EXATAS E TECNOLÓGICAS - ICET
CURSO DE CIÊNCIA DA COMPUTAÇÃO
CAMPUS MANAUS**

**TIAGO DA SILVA BRILHANTE – T481104
GUILHERME GUELFY – N015985
EWERTON LUIZ DA SILVA PAIVA – F3447F8**

**DESENVOLVIMENTO DE UM JOGO IMPLEMENTANDO OS CONCEITOS DE
ORIENTAÇÃO A OBJETOS UTILIZANDO A LINGUAGEM JAVA**

Trabalho de Atividade Prática
Supervisionada –UNIP como pré-requisito
para obtenção de nota da disciplina:
Atividade Prática supervisionada – APS
(CC3P34).

Orientador: Prof. Raimundo Fagner Costa

MANAUS-AM

2022

RESUMO

Jogos em geral, são bons exemplos de aplicações desenvolvidas com um forte foco na orientação a objetos. Geralmente são projetados seguindo os princípios e conceitos do paradigma orientado a objetos, que busca modelar o mundo real em termos de objetos interagindo entre si.

No jogo, a estrutura é organizada em torno de classes e objetos, onde cada elemento do jogo é representado por uma instância de uma classe específica. Por exemplo, pode haver classes para representar personagens, itens, cenários e interações.

A orientação a objetos permite criar uma hierarquia de classes, usando conceitos como herança e polimorfismo. Isso significa que é possível definir classes mais gerais e, em seguida, estender essas classes para criar classes mais específicas com características adicionais. Por exemplo, pode haver uma classe "Personagem" geral e classes derivadas como "Herói" e "Inimigo" que herdam as características básicas da classe "Personagem", mas possuem comportamentos e atributos específicos.

Além disso, o encapsulamento é um aspecto fundamental da orientação a objetos. Isso significa que os detalhes internos de cada objeto são ocultados e só podem ser acessados através de métodos públicos. Isso garante que cada objeto possa manter seu estado interno e oferecer uma interface controlada para interagir com outros objetos.

No jogo, a interação entre os objetos é facilitada pelo uso de mensagens e chamadas de métodos. Por exemplo, um personagem pode interagir com um item ao chamar um método específico desse item, resultando em uma ação no jogo.

A orientação a objetos também permite criar jogos mais modulares e extensíveis. Com a separação clara das responsabilidades em diferentes classes, é possível adicionar novos recursos e funcionalidades ao jogo de forma mais fácil, sem afetar o código existente. Em resumo, o nosso jogo "ScrAPS World" é um exemplo de aplicação em Java que utiliza fortemente os conceitos de orientação a objetos. Ele aproveita a estrutura de classes e objetos para modelar o mundo do jogo, permitindo interações, extensibilidade e modularidade. Através desse paradigma, é possível criar jogos mais organizados, flexíveis e de fácil manutenção.

Palavra Chaves: Jogo; Orientação a objetos; Java.

SUMÁRIO

INTRODUÇÃO	5
1. DEFINIÇÕES, OBJETIVOS E IMPORTÂNCIA DO POO	7
1.1 TERMINOLOGIA	9
2. O JAVA COMO LINGUAGEM PARA O DESENVOLVIMENTO DE JOGOS	11
3. O DESENVOLVIMENTO DE UM NOVO JOGO	13
3.1 O INÍCIO DE TUDO	13
3.2 O GDD	16
3.3 A TRILHA SONORA	26
3.4 A ARTE CONCEITUAL	27
3.5 TRATANDO SOBRE OBJETOS	33
3.6 ESTRATÉGIA E DESENVOLVIMENTO DO CÓDIGO	37
3.6.1 A CLASSE GAME.JAVA	37
3.6.2 A CLASSE MUNDO.JAVA e ENTITY.JAVA	42
3.6.3 A CLASSE PLAYER.JAVA	50
4. O CÓDIGO FONTE DO JOGO	52
5. GALERIA DE IMAGENS	52
6. CONCLUSÃO	55
7. REFERENCIAS	56

INTRODUÇÃO

De acordo com o a *Wikipedia*, o conceito de jogo é definido da seguinte forma:

Jogo é toda e qualquer atividade em que exista a figura do jogador (como indivíduo praticante) e regras que podem ser para ambiente restrito ou livre. Geralmente os jogos têm poucas regras e estas tendem a ser simples. Sua presença é importante em vários aspectos, entre eles a regra define o início e fim do jogo. Pode envolver dois ou mais jogando entre si como adversários ou cooperativamente com grupos de adversários. É importante que um jogo tenha adversários interagindo e como resultado de interação exista um vencedor e um perdedor. (WIKIPEDIA - Jogo, 2023).

A *Wikipédia* (2023) deixa em evidência que o jogo deve possuir regras, e que deve existir um vencedor e um perdedor, no entanto essa fórmula não precisa ser necessariamente seguida, já que na atualidade os jogos eletrônicos, também chamados de *vídeo games*, envolvem também um conceito que abrange tão somente a experiência passada ao usuário.

Em tempos antigos, era comum jogar de forma não digital, com os chamados jogos de tabuleiro e jogos de cartas por exemplo, além de existirem os jogos que envolvem a prática de atividades físicas, como futebol e tênis.

A realidade é que os jogos, sejam eletrônicos ou não, despertam na sociedade atual, um sentimento de integração e competição, geralmente de forma salutar, o que acompanhamos em tempos de olimpíadas ou grandes competições dos chamados *e-sports*.

O objetivo do nosso trabalho é o desenvolvimento de um jogo eletrônico, utilizando os conceitos de programação orientada a objetos.

Jogos eletrônicos são formas de entretenimento interativo que envolvem o uso de dispositivos eletrônicos, como computadores, consoles de videogame, dispositivos móveis e outros aparelhos eletrônicos. Eles são projetados para proporcionar aos jogadores uma experiência imersiva e interativa, permitindo que eles assumam o controle de personagens virtuais em cenários digitais.

Os jogos eletrônicos oferecem uma variedade de gêneros e estilos, desde jogos de ação e aventura até jogos de quebra-cabeça, estratégia, esportes, simulação e muito mais. Cada jogo

tem suas próprias regras, objetivos e mecânicas de jogabilidade, criando desafios e situações únicas para os jogadores enfrentarem.

Uma das principais características dos jogos eletrônicos é a interatividade. Os jogadores podem controlar os personagens e tomar decisões que afetam o curso do jogo. Eles podem explorar ambientes virtuais, resolver enigmas, enfrentar inimigos, competir com outros jogadores ou colaborar em desafios multiplayer. A interatividade proporciona uma experiência envolvente, onde os jogadores podem se sentir parte ativa do mundo virtual.

Os jogos eletrônicos também costumam oferecer elementos de progressão e recompensa. Os jogadores podem avançar na história do jogo, desbloquear novos níveis ou conteúdos, ganhar pontos, obter melhorias para os personagens e alcançar metas específicas. Esses elementos motivam os jogadores a continuarem jogando, explorando novas possibilidades e superando desafios cada vez mais difíceis.

Além disso, os jogos eletrônicos podem proporcionar experiências sociais. Muitos jogos oferecem recursos multiplayer, nos quais os jogadores podem interagir e competir com outras pessoas em tempo real. Isso pode ocorrer localmente, com amigos jogando no mesmo dispositivo, ou online, conectando-se a jogadores de todo o mundo. Os jogos multiplayer podem promover a colaboração, o trabalho em equipe e a competição saudável entre os jogadores.

Os jogos eletrônicos se tornaram uma forma popular de entretenimento em todo o mundo, abrangendo diversas faixas etárias e culturas. Eles oferecem não apenas diversão, mas também desafios intelectuais, oportunidades de aprendizado, narrativas envolventes e um meio de se conectar com outras pessoas. Com avanços contínuos na tecnologia, os jogos eletrônicos continuam evoluindo, oferecendo experiências cada vez mais imersivas, realistas e inovadoras para os jogadores desfrutarem.

Em virtude do objetivo dessa APS se tratar de um jogo eletrônico que utilizasse os conceitos do paradigma de orientação a objetos (POO), ficou estabelecido (pela UNIP) o uso da linguagem Java para a execução do projeto.

Java é uma linguagem de programação que tem muitas vantagens quando se trata de

desenvolvimento de jogos eletrônicos. Uma das principais razões é a facilidade de programação orientada a objetos, que é uma metodologia amplamente utilizada na criação de jogos. Com a orientação a objetos, é possível criar estruturas de classes para gerenciar entidades do jogo, como personagens, inimigos, itens, cenários, entre outros. Isso torna o desenvolvimento do jogo mais organizado e fácil de manter.

Além disso, a linguagem Java é conhecida por sua portabilidade, o que significa que um programa Java pode ser executado em diferentes sistemas operacionais, desde que haja uma máquina virtual Java (JVM) instalada. Isso é particularmente útil na criação de jogos, pois permite que o jogo seja executado em diferentes plataformas, como desktops, laptops, smartphones e tablets, sem a necessidade de criar diferentes versões do jogo para cada plataforma.

Outra vantagem da linguagem Java para jogos é o suporte a bibliotecas gráficas poderosas, como o JavaFX e o LWJGL (*Lightweight Java Game Library*). Essas bibliotecas permitem a criação de gráficos e efeitos visuais impressionantes, além de suportar recursos como som e animação.

Além disso, a linguagem Java é conhecida por ser segura e confiável, o que é especialmente importante quando se trata de jogos online e multijogador, pois há riscos de vulnerabilidades de segurança. Java é usado em muitos jogos online populares, como Minecraft, que tem muitos jogadores em todo o mundo.

Em resumo, a linguagem Java é uma escolha popular para a criação de jogos eletrônicos devido à sua facilidade de programação orientada a objetos, portabilidade, suporte a bibliotecas gráficas poderosas e segurança confiável. É uma linguagem que pode oferecer um grande potencial para quem quer desenvolver jogos.

1. DEFINIÇÕES, OBJETIVOS E IMPORTÂNCIA DO POO

De acordo com a *UGO ROVEDA* (2022), temos como definição de POO, o seguinte:

POO é um paradigma de programação que se propõe a abordar o design de um sistema em termos de entidades, os objetos, e relacionamentos entre essas entidades. (ROVEDA, 2022).

A definição torna evidente então qual é o objetivo primário do paradigma de orientação a objetos, que nada mais é do que tentar replicar entidades que encontramos na vida real de forma virtual.

Ainda segundo *ROVEDA* (2022):

Paradigmas de programação existem para responder questões que surgem justamente do processo de pensar na abordagem que será utilizada para a solução de problemas. Enquanto uma linguagem de programação é a implementação de fato de uma ferramenta para desenvolver um software, o paradigma de programação é o modelo conceitual e o conjunto de padrões e metodologias que serão aplicadas no uso de uma linguagem para o desenvolvimento de um software.

O objetivo do paradigma de orientação a objetos é fornecer uma abordagem de desenvolvimento de software que permite a criação de sistemas mais flexíveis, reutilizáveis e fáceis de manter.

A importância do paradigma de orientação a objetos na programação é amplamente reconhecida na indústria de desenvolvimento de software. Essa abordagem se baseia na representação de entidades do mundo real como objetos, que possuem características (atributos) e comportamentos (métodos) associados a eles.

Um dos principais benefícios da orientação a objetos é a modularidade. Por meio da criação de classes e objetos, o código pode ser dividido em módulos independentes e reutilizáveis. Esses módulos encapsulam funcionalidades específicas e podem ser combinados de maneira flexível para construir sistemas complexos. A modularidade facilita a compreensão, manutenção e evolução do código, tornando-o mais organizado e legível (*GeeksforGeeks*, 2023).

A reutilização de código é outra vantagem importante da orientação a objetos. Com a criação de classes e hierarquias de herança, é possível compartilhar e estender funcionalidades existentes em diferentes partes do sistema. Isso economiza tempo e esforço de desenvolvimento, pois não é necessário reescrever o mesmo código repetidamente. Além disso,

a reutilização de código promove a consistência e padronização, uma vez que os objetos podem ser utilizados em múltiplos contextos (Guru99, 2023).

Outro aspecto relevante da orientação a objetos é a abstração, que permite simplificar a complexidade do sistema ao focar nos aspectos essenciais. Por meio da definição de interfaces e classes abstratas, é possível modelar conceitos e entidades do mundo real de forma clara e concisa. A abstração permite uma compreensão mais intuitiva do sistema, facilitando o seu desenvolvimento e manutenção (Guru99, 2023).

Além disso, a orientação a objetos promove o polimorfismo, que permite tratar objetos de diferentes classes de maneira uniforme, desde que eles compartilhem uma mesma interface ou herança. Isso traz flexibilidade ao sistema, permitindo a substituição de objetos sem impactar o funcionamento dos demais componentes. O polimorfismo também facilita a criação de código genérico e flexível, que pode lidar com diferentes tipos de objetos de forma dinâmica (Guru99, 2023).

Essas características e benefícios da orientação a objetos contribuem para o desenvolvimento de software mais modular, flexível, reutilizável e de fácil manutenção. Muitas linguagens de programação modernas, como Java, C++, Python e C#, suportam nativamente a orientação a objetos e fornecem recursos avançados para sua implementação (Udacity, 2023).

1.1 TERMINOLOGIA

Para que haja um maior entendimento sobre a programação orientada a objetos, precisamos inicialmente nos familiarizar com alguns termos.

A programação orientada a objetos (POO) é um paradigma de programação que se baseia na organização e estruturação do código em torno de objetos, que são instâncias de classes. Nesse contexto, existem diversos termos e conceitos específicos associados à programação orientada a objetos. Alguns exemplos:

1. Classe: Uma classe é a estrutura que define as características e comportamentos de um objeto. Ela serve como um modelo ou plano para a criação de objetos. (Oracle - The Java™ Tutorials, 2023)

2. Objeto: Um objeto é uma instância de uma classe. Ele possui atributos (variáveis) que descrevem seu estado e métodos (funções) que definem seu comportamento. Uma instância é uma ocorrência específica de um objeto, criada a partir de uma classe. Cada instância pode ter seu próprio estado, mas compartilha os comportamentos definidos pela classe. (*MDN Web Docs - JavaScript* , 2023)

3. Encapsulamento: O encapsulamento é o princípio de esconder os detalhes internos de um objeto e fornecer uma interface pública para interagir com ele. Isso garante a segurança e integridade dos dados, além de facilitar a manutenção do código. (*Tutorials Point - Encapsulation in Java* , 2023)

4. Herança: A herança é um mecanismo que permite que uma classe herde características e comportamentos de outra classe. Isso permite a reutilização de código e a criação de hierarquias de classes. (*W3Schools - Java Inheritance*, 2023)

5. Polimorfismo: O polimorfismo permite que um objeto seja tratado de diferentes formas, dependendo do contexto em que é usado. Isso possibilita o uso de métodos com o mesmo nome, mas com comportamentos diferentes em classes diferentes. (*Baeldung - Polymorphism in Java*, 2023)

6. Abstração: A abstração é o processo de identificar características essenciais de um objeto e ignorar os detalhes irrelevantes. Ela permite criar classes abstratas e interfaces que definem contratos e fornecem uma visão simplificada dos objetos. (*Tutorialspoint - Object-Oriented Analysis & Design* , 2023)

Esses são alguns dos termos mais comuns relacionados à programação orientada a objetos. Cada termo possui uma gama de conceitos e aplicações mais detalhadas, e as fontes mencionadas fornecem informações mais abrangentes e exemplos práticos para aprofundar o conhecimento sobre cada um deles.

2. O JAVA COMO LINGUAGEM PARA O DESENVOLVIMENTO DE JOGOS

A decisão de usar a linguagem Java para a criação de jogos é baseada em diversas vantagens e recursos oferecidos por essa linguagem. Vamos explorar essa decisão, destacando os termos específicos da linguagem:

- **Portabilidade:** Uma das principais vantagens do Java para jogos é sua portabilidade. Através do conceito "*write once, run anywhere*" (escreva uma vez, execute em qualquer lugar), os jogos desenvolvidos em Java podem ser executados em diferentes plataformas, como Windows, MacOS, Linux e até mesmo dispositivos móveis. Essa característica é possível graças à Máquina Virtual Java (JVM), que interpreta o código Java em tempo de execução. (*Oracle - Java Gaming*, 2023)
- **Orientação a Objetos:** A linguagem Java é amplamente baseada no paradigma de programação orientada a objetos (POO). Isso significa que os jogos desenvolvidos em Java podem ser estruturados em torno de classes, objetos e seus relacionamentos, o que permite uma organização modular e reutilização de código. Os conceitos de herança, encapsulamento, polimorfismo e abstração são fundamentais na criação de jogos complexos e bem estruturados em Java. Fonte: (*Oracle - The Java™ Tutorials*, 2023)
- **Bibliotecas e Frameworks:** A comunidade Java oferece uma ampla variedade de bibliotecas e frameworks específicos para jogos, facilitando o desenvolvimento de diferentes aspectos dos jogos, como gráficos, som, física, rede e IA. Exemplos populares incluem a libGDX (<https://libgdx.com/>) e o JavaFX (<https://openjfx.io/>). Essas bibliotecas fornecem abstrações e funcionalidades prontas para uso, permitindo aos desenvolvedores concentrarem-se na lógica do jogo em vez de implementar funcionalidades básicas do zero. (*DZone - Top Java Game Development Libraries*, 2023)
- **Ecossistema e Comunidade:** A linguagem Java possui uma comunidade ativa e um ecossistema maduro, o que significa que existem muitos recursos disponíveis para suporte, aprendizado e compartilhamento de conhecimento. Fóruns, tutoriais,

documentação e exemplos de código são amplamente disponibilizados pela comunidade Java, facilitando a resolução de problemas e o aprendizado contínuo. Fonte: *Oracle - Java Community* (<https://community.oracle.com/community/developer>)

Ao escolher o Java como linguagem para a criação de jogos, os desenvolvedores podem aproveitar a portabilidade, a orientação a objetos, as bibliotecas específicas para jogos e o suporte de uma comunidade vibrante. Esses fatores contribuem para uma experiência de desenvolvimento mais eficiente e permitem a criação de jogos robustos e de alta qualidade em Java.

3. O DESENVOLVIMENTO DE UM NOVO JOGO

A proposta de trabalho da APS, fala o seguinte:

Pede-se aos alunos que desenvolvam um software/jogo com interface gráfica utilizando a linguagem Java. O tema do jogo tem que conter a educação ambiental tendo como base a vida numa grande metrópole.

E como observações importantes, destacamos o seguinte:

- 1) Quaisquer componentes gráficos poderão ser utilizados, desde que desenvolvidos na linguagem Java.
- 2) A escolha do número de jogadores que participarão do jogo fica a cargo do grupo.
- 3) O grupo deverá fazer uma dissertação sobre todos os elementos utilizados no desenvolvimento do projeto, assim como o efeito desse trabalho na sua formação e discutir a interdisciplinaridade envolvida no mesmo.

Após a análise do pedido principal, chegou-se à conclusão que, para o bem do desafio acadêmico, seria proposta a criação um jogo com o mínimo de bibliotecas de terceiros, e se valendo principalmente do que a linguagem Java traz de forma nativa.

Além disso, a rigorosa limitação do tema proposto pela UNIP se constituiu em um grande desafio, pois a comunidade de jogadores existente na atualidade não tem muito interesse em jogos com propostas voltadas ao meio ambiente.

Com base nisso, foi iniciado o processo de desenvolvimento do jogo “*ScrAPS World*”.

3.1 O INÍCIO DE TUDO

Inicialmente, entendemos que um jogo que possui um tema proposto, precisa de certa forma conter elementos chave no seu desenvolvimento.

Podemos citar como elementos principais em um jogo, o seguinte:

1. **Conceito e Tema:** Todo jogo começa com um conceito e um tema. O conceito define a ideia principal do jogo, enquanto o tema define a ambientação, história ou estilo visual do jogo. Esses elementos dão forma e identidade ao jogo.
2. **Mecânicas de Jogo:** São as regras e interações que definem como o jogador irá interagir com o jogo. Inclui elementos como controles, movimento, ações do jogador, sistema de pontuação, combate, quebra-cabeças, entre outros. As mecânicas de jogo são responsáveis por criar desafios e proporcionar diversão ao jogador.
3. **Gráficos e Design Visual:** Os elementos visuais do jogo, como gráficos, animações, personagens, cenários e efeitos visuais, desempenham um papel importante na imersão e na experiência do jogador. O design visual deve ser atraente, coeso e contribuir para a narrativa do jogo.
4. **Áudio e Trilha Sonora:** O áudio é um elemento essencial na criação de jogos, pois contribui para a atmosfera, emoção e feedback ao jogador. Isso inclui efeitos sonoros para ações, diálogos, trilha sonora e até mesmo dublagens. O áudio imersivo ajuda a criar uma experiência envolvente.
5. **Inteligência Artificial:** A IA é responsável por controlar o comportamento de personagens não jogáveis (NPCs) e criar desafios para o jogador. Ela pode ser utilizada para criar inimigos inteligentes, aliados controlados pelo computador, ajustar a dificuldade do jogo e oferecer respostas realistas às ações do jogador.
6. **Progressão e Sistema de Recompensas:** Um jogo bem projetado possui um sistema de progressão, no qual o jogador é recompensado por suas conquistas e avança na história ou níveis do jogo. Isso pode incluir desbloqueio de novas habilidades, itens, áreas ou narrativa. A progressão e as recompensas incentivam o jogador a continuar jogando e oferecem um senso de satisfação e realização.

7. Interface do Usuário (UI): A UI abrange os elementos de interface gráfica que permitem ao jogador interagir com o jogo, como menus, barras de vida, indicadores de pontuação, mapas e demais elementos visuais que facilitam a compreensão do jogo e a navegação pelo seu conteúdo.

8. Testes e Ajustes: Os jogos devem passar por testes rigorosos para identificar bugs, problemas de desempenho e ajustar o equilíbrio do jogo. Os testes permitem refinar e aprimorar a jogabilidade, garantindo uma experiência de jogo suave e satisfatória.

É importante considerar todos esses elementos de forma integrada para oferecer uma experiência de jogo envolvente e cativante aos jogadores.

Em um primeiro momento, entendemos que o jogo poderia se passar no espaço, pela simplicidade e facilidade de construção de cenários e física, no entanto isso não atenderia ao tema imposto pela UNIP, foi então que ficou decidido que o estilo de jogo seria PLATAFORMA, e iniciou-se o desenvolvimento do GDD.

GDD é a sigla para "*Game Design Document*" (Documento de Design de Jogo, em português). O GDD é um documento utilizado na indústria de jogos para descrever e documentar todos os aspectos do design e desenvolvimento de um jogo.

O GDD serve como um guia abrangente que contém informações detalhadas sobre o conceito do jogo, mecânicas de jogo, personagens, história, arte, som, níveis, interface do usuário, requisitos técnicos e qualquer outra informação relevante para o desenvolvimento do jogo.

O objetivo principal do GDD é fornecer uma referência centralizada para toda a equipe de desenvolvimento do jogo, incluindo designers, artistas, programadores e outros membros da equipe. Ele auxilia na comunicação, alinhamento de visão e tomada de decisões durante todo o processo de desenvolvimento.

O conteúdo exato de um GDD pode variar dependendo do estúdio de desenvolvimento e do projeto específico do jogo, mas geralmente inclui seções como visão geral do jogo,

mecânicas de jogo, personagens e história, arte e áudio, níveis e progressão, interface do usuário e requisitos técnicos.

O GDD é uma ferramenta valiosa para ajudar a garantir que todos os aspectos do jogo sejam considerados e planejados de forma adequada, desde a concepção até a implementação. Ele fornece uma estrutura organizada para o desenvolvimento do jogo e auxilia na tomada de decisões consistentes e informadas ao longo do processo.

Em resumo, o GDD é um documento essencial que descreve todos os detalhes e especificações do design de um jogo, fornecendo uma base sólida para o desenvolvimento e garantindo a coesão e qualidade do produto final.

3.2 O GDD

Com base em experiências da equipe, propomos as seguintes perguntas iniciais:

- Como será o design do personagem principal?
- Quem é ele?
- Quais são seus traumas?
- Qual sua missão?
- Qual sua motivação?
- Qual é o seu objetivo?
- Como é o mundo no qual ele vive?

Acreditamos que para gerar a ambientação necessária para uma boa imersão, essas perguntas eram importantes, e precisavam de respostas.

Ao final de uma análise inicial, chegamos ao seguinte:

Qual é o tempo em que se passa o jogo: Futuro (apocalíptico)

Ambientação: O planeta terra foi devastado por causa da ação humana. O nível de poluição se tornou caótico e irreversível.

Por causa de práticas predatórias e pouco cuidado com o meio ambiente, a humanidade vive um período em que caminha para a sua própria extinção.

Muitos já morreram por causas de desastres naturais em consequência da atual situação.

Muitos já morreram por causa das guerras travadas em busca dos poucos territórios onde a vida é mais propícia.

Doenças de toda sorte assolam os vivos. A sobrevivência não é fácil.

Quem é o player: Um ex-piloto militar que tenta sobreviver catando scrap(lixo) em uma grande metrópole arrasada, mas ainda sim, o último grande reduto da humanidade.

Quais são seus traumas: Ele perdeu toda a família, e é uma pessoa solitária. Tem medo de se relacionar com alguém e perder a pessoa.

Como é a rotina do player: Ele anda pela cidade, coletando lixo, que em um primeiro momento tem a finalidade de tentar propiciar uma melhoria da situação inicial em que se encontra.

Qual é sua missão: Sobreviver, e tentar ajudar na reconstrução da sociedade local em que vive

Qual é a motivação do player: O player apenas anseia por sobrevivência e procura melhorar as coisas da forma que pode.

Quais são as ameaças contra o player:

- (em momentos iniciais) Animais selvagens, monstros mutantes, bandidos.
- (avançado no game) Alienígenas.

Figura 1: Perguntas iniciais

Com base nisso, passamos ao desenvolvimento de uma progressão que fizesse sentido nesse contexto apresentado.

Ficou decidido que a arte / design do jogo seria feita em *Pixel Art*.

A *Pixel Art* é uma forma de arte digital que se originou nas décadas de 1970 e 1980, com o surgimento dos primeiros jogos eletrônicos. Ela se caracteriza pelo uso de pixels individuais para criar imagens de baixa resolução, com um estilo visual único e distintivo (*Dan Whitehead, 2022, JOE MATAR, 2020*)

A principal característica da *Pixel Art* é a sua estética retro, que remete aos jogos clássicos de arcade e consoles antigos. Ela utiliza uma grade de pixels para representar personagens, objetos, cenários e outros elementos visuais do jogo. Cada pixel é cuidadosamente posicionado e colorido para criar uma imagem detalhada dentro das limitações de resolução (*JOE MATAR, 2020*).

A *Pixel Art* é valorizada por muitos desenvolvedores de jogos e entusiastas por várias razões. Em primeiro lugar, ela evoca nostalgia e sentimentos de familiaridade, lembrando a era dos jogos clássicos que muitas pessoas cresceram jogando. Além disso, a simplicidade visual da *Pixel Art* permite uma representação clara e distintiva dos objetos, facilitando o reconhecimento e a identificação dos elementos do jogo. (*Dan Whitehead, 2022, JOE MATAR, 2020*).

A criação de *Pixel Art* requer habilidades técnicas e artísticas. Os artistas de *Pixel Art* utilizam ferramentas específicas, como editores de gráficos baseados em pixels, para criar e editar suas imagens. Eles trabalham com uma paleta de cores limitada, geralmente composta por 8 ou 16 cores, para alcançar o estilo retro autêntico. (*JOE MATAR, 2020*).

A *Pixel Art* também se destaca pela sua capacidade de transmitir emoções e contar histórias de maneira eficaz. Mesmo com suas restrições de resolução e número limitado de pixels, os artistas conseguem criar personagens e ambientes cativantes, transmitindo expressões faciais, movimento e atmosfera através de detalhes sutis. (*Dan Whitehead, 2022*).

O programa decidido para a criação da arte seria o Aseprite e a paleta de cores a ser utilizada seria a AAP – 64 pela ótima variedade de cores.



Figura 2: Software para criação da arte

AAP-64



Figura 3: Paleta de cores AAP-64 (fonte: Aseprite)

Passamos então a tentar estabelecer o desenvolvimento da história.

Nessa fase, passamos a responder as seguintes perguntas sobre o design e a mecânica do jogo:

- Qual é o nome do jogo?
- Como o player coleta o lixo?
- Como o player ataca os inimigos?
- Qual é o nome do player?
- Quais são as mecânicas de deslocamento?
- Como é feita a progressão?
- Como seria a pontuação do jogo?
- Como será a política de *save*?
- Como será a política *life/vidas*?

Vale a pena ressaltar que nesse momento ainda não existia uma linha de código sequer, nem UML tratando sobre objetos e métodos. Apenas conceitos relativos a como o jogo será criado.

Um aspecto importante é que responder essas perguntas antes de produzir qualquer coisa (inclusive a arte), facilita o trabalho de desenvolvimento, e oferece o foco necessário para a execução das diversas tarefas.

Ficou estabelecido o seguinte:

Qual é o nome do jogo?

Chamaríamos o jogo de ScrAPS World, justamente para criar uma brincadeira com o nome Scrap (que em inglês significa lixo), APS (que é a sigla de Atividade Prática Supervisionada) e world (que é mundo em inglês)

Como o player coleta o lixo?

O lixo ficaria espalhado pela fase, e o jogador passaria para a próxima fase após recolher todo o lixo existente.

Como o player ataca os inimigos?

Poderia atacar com um pedaço de cano que ele leva consigo ou com uma arma de fogo (pistola), no entanto essa pistola pode falhar alguns tiros, e possui munição limitada.

Qual é o nome do player?

Ficou decidido que o nome do player seria Elidan.

Quais são as mecânicas de deslocamento?

O jogador pode andar para a esquerda e direita, pode subir e descer escadas, e pode pular para a esquerda e direita.

Como é feita a progressão?

A progressão é feita coletando o lixo nas diversas fases.

Como seria a pontuação do jogo?

Cada lixo recolhido é equivalente a um ponto

Como será a política de save?

Em cada fase existirão placas, que quando em contato com o player, salva a posição.

Como será a política energia /vidas?

O player iniciará o jogo com 100 pontos de energia e com 3 vidas. Ao entrar em contato com entidades inimigas ou certos locais no mapa, perde a energia com velocidades distintas. Caso sua energia chegue a zero, o player perde uma vida, se perder as 3 vidas, o jogo se encerra.

Figura 4: Perguntas adicionais e sobre mecânica

Com essas perguntas sobre a mecânica respondidas, passamos ao design e *level design*.

Pensamos inicialmente em uma tela inicial abrindo direto no menu, onde teríamos as seguintes opções:

- Jogar – Ao acessá-la, seria explicada a história do jogo para que em seguida iniciasse o *level 1*.
- Leaderboards – Mostraria as 3 melhores pontuação do jogo
- Controles – Mostraria as teclas usadas para jogar
- Sobre – Mostraria os elementos do grupo da APS que criaram o jogo
- Sair – Sairia do jogo

Em seguida, pensamos em como as fases se desenvolveriam.

Vale a pena ressaltar que o escopo inicialmente pensado abrangeria muito mais do que realmente fora implementado. E deixo como crítica à própria UNIP, o fato dela demandar por um jogo, que se trata de algo extremamente complexo, sendo que em sala, só foram tratados sobre os conceitos de POO, e que apenas no semestre seguinte teríamos a matéria de Aplicação de Linguagem de Orientação a Objetos.

Graças a minha experiência de cerca de 20 anos de desenvolvedor, foi possível chegar a um jogo com as características apresentadas, no entanto a falta de tempo e de elementos capacitados no grupo, obrigaram a uma redução drástica do escopo inicial, que agora apresentarei:

FASE 1 (deserto- arredores da cidade)

- O Player passa a primeira fase se ambientando com os comandos básicos do jogo...
- São apresentadas as mecânicas básicas de movimento e ataque.
- Nenhum inimigo aparece por aqui.
- A fase se passa nos arredores da metrópole (deserto).
- A fase termina quando ele consegue coletar todos os lixos.
- A música é desoladora.
- Muito espaço de tela para o player.
- Diálogos falando sobre a situação atual e as dificuldades para sobreviver.

FASE 2 (a cidade atual)

- O player retorna para a metrópole, para trocar o lixo recolhido por créditos
- Pela primeira vez é possível ver a situação em que se encontra a cidade.
- A música é triste
- Diálogos sobre a situação da cidade e do planeta.
- Diálogos que buscam contar a história do player.
- Um dia se passa...
- Os créditos (obtidos pela troca do lixo) são poucos e mal dá pra alimentar.
- O player sente fome... tudo poderia ser diferente se eles tivessem cuidado mais do planeta
- Ele chega à conclusão que precisa coletar mais lixo...
- Ele dorme
- Ele sonha

FASE 3 (a terra de antes)

- O player acorda (no sonho) num mundo vivo... a terra como era antes
- Não há lixo nem inimigos (só a beleza do local)
- Diálogos mostrando a importância de cuidar do meio ambiente
- o player passa por lugares que possuem rio, flora (sonho)...
- O player acorda por causa da fome...
- Ele decide que tem que coletar mais lixo...

FASE 4 (deserto – arredores da cidade)

- O player sai para coletar lixos.
- A fase ainda é o deserto.
- É apresentada a mecânica de combate
- Inimigos aparecem
- Existem diálogos do player explicando a natureza dos inimigos (porque bandidos, por que animais selvagens)
- A música se agita um pouco por causa da presença de inimigos
- A fase termina quando os lixos são coletados.

FASE 5 (a cidade atual)

- O player volta para a cidade...
- Na troca de lixos por créditos ele percebe um cartaz na parede
- O governo está recrutando pessoas para uma missão de busca por um novo planeta habitável com a finalidade de realocar a raça humana.
- O player lembra da história dele como ex-piloto e dada as circunstâncias, decide participar do recrutamento.
- Aqui são apresentados 2 amigos do player...
- Amigo 1
- Amigo 2
- Diálogos sobre o panfleto...
- O amigo 1 também é ex-piloto, o amigo 2 é técnico de armas do exército, o player chama os dois para embarcar nessa nova jornada.
- Ele vai até a agência de treinamento de Cadetes espaciais com os amigos.
- Diálogos sobre a missão.

- Eles são separados...
- O player chega no alojamento dele.
- Muitos estão por lá também... ele chega na cama e dorme.

FASE 6 (Cidade – Espaço)

- O player acorda, é apresentado a sua nave de treinamento.
- A nave tem um nome (NOME_DA_NAVES)
- É apresentado o robô ajudante da nave...
- É explicada a mecânica de save da nave
- A nave é uma gambiarra só...
- O robô é cômico.
- Ele embarca e vai para o espaço iniciar seu treinamento.
- É explicada a mecânica de uso da nave...
- Os inimigos nesse momento são alvos ... ou lixo espaciais...
- Destruído o último alvo... eles sofrem um ataque do nada.
- Boa parte da frota é destruída... mas ele sobrevive...
- As naves que não foram destruídas recuam para a terra...
- O Player recebe ordem de perseguir a ameaça para fornecer informações para a base na terra...
- A base informa que vai mandar assim que possível a nave de suporte ao combate para ajudar o player (nave mãe)
- O player está só na missão.... e ele parte em rumo ao desconhecido...

FASE 7 (espaço aberto – proximidades do sistema solar)

- O player está no espaço e começa a interceptar naves inimigas...
- Diálogos com o robô sobre o que está acontecendo
- Alguns espaços sobre contemplação do espaço, a solidão de estar lá... o robô não gosta, pois ele está lá com o player (ou seja, o player não está só... está com o robô).
- O robô tenta animar o player...
- Mais inimigos aparecem
- A nave as vezes apresenta alguns barulhos estranhos... e o player começa a perceber que a nave é uma porcaria... mas é o que ele tem pra hoje
- Inimigos e mais inimigos

Depois de um tempo aparece o primeiro BOSS

BOSS 1 – explicação sobre o boss 1

- O boss é destruído, e o player vê que eles estão entrando em um cinturão de asteroides tentando despistar o player.
- O player avisa a terra que informa que a nave mãe deve chegar em instantes na posição que ele está agora e determina que ele , por ser menor, entre no cinturão de asteroides para continuar a perseguição.

FASE 8 – (Cinturão de asteroides)

- Dificuldade de movimentar a nave em meio a tanta pedra
- Inimigos
- Diálogos entre o robô e o player

- Player repete que a nave é uma porcaria espacial...
- Inimigos...
- Os canhões de tiro começam a falhar e possuem um fator aleatório de 50% de sair o tiro...
- Em determinado momento a nave perde completamente os canhões de tiro e só podem se desviar dos tiros inimigos.
- O robô se empenha em consertar tudo, e depois de um tempo consegue...
- Diálogos cômicos sobre a aeronave... nunca duvidei...
- Inimigos...

BOSS 2 – explicação sobre o boss 2

- O boss morre, e o player informa a nave mãe que está tudo limpo até a posição onde ele se encontra, que a nave pode avançar, no entanto ele precisará usar o motor de dobra espacial quântica para continuar a perseguição, pois percebeu aeronaves inimigas dando o salto de velocidade hiperluz e vai perder o contato por um breve instante...
- A base dá o ok e ele prossegue...

FASE 9 – (a hipervelocidade)

- O player se encontra em hipervelocidade... as estrelas parecem riscos...
 - Inimigos aparecem...
 - Diálogos sobre a hipervelocidade...
 - O escudo começa a descer até ficar com 1 ponto (o máximo é 100)
 - O robô informa que o sistema de defesa está inoperante... se tomar um hit morre...
 - Inimigos...
 - O robô consegue ajustar o sistema de defesa e a nave volta ao normal...
- Inimigos...

BOSS 3 – Explicação sobre o boss

- O boss morre e o player termina o salto de hipervelocidade...
- Ele informa a nave mãe a posição atual e manda ele ir pra lá...
- Fala sobre a manutenção da nave...
- Informa que viu os inimigos se aproximarem de uma estrela para tentar dificultar a perseguição... e você vai atras... Prossiga na missão, diz a nave mãe...

FASE 10 – (A estrela 54f37-H)

- A estrela é grande (similar ao nosso sol), a parte de baixo da tela agora aparece com plasma (fogo) da estrela, se o player encostar lá morre imediatamente...
- Falar sobre o calor... Irradiação térmica...
- Lembrar do efeito estufa...
- Inimigos...
- Diálogos genéricos
- Inimigos

BOSS 4 – Explicação sobre o boss

- O boss morre mas ao morrer ele explode e danifica o robô...

- O player percebe e avisa a base que vai precisar fazer reparos nele , pois sem ele é impossível prosseguir (demonstração de laço afetivo)...
- O player sai do raio de influência da estrela... e sobe para uma posição segura...
- Vemos a nave mãe chegar e o player pouso nela...

FASE 11 – (a nave mãe)

- O player começa a jornada para consertar o robô...
- Passa em vários lugares da nave mãe para coletar as peças...
- E ao coletar todas, volta para o laboratório
- DIALOGOS...
- O player encontra os 2 amigos, e conversam com eles...
- O robô é consertado... (daquele jeito...)
- O chefe da missão avisa que interceptou a localização das aeronaves alienígenas, e parece que eles (inimigos) se encontram na orbita do planeta natal deles e que o planeta é habitável!
- A nave mãe convoca todos os pilotos disponíveis para o ataque...
- O player se prepara para partir...
- O player percebe que o robô está estranho...
- O player parte...

FASE 12 – (a orbita do planeta inimigo)

- O Robô está estranho...
- A nave começa a falhar na movimentação... (ainda sem inimigos na tela)
- A movimentação da nave falha de vez...
- Os comandos ficam invertidos...
- Inimigos... poucos
- Inimigos... médio
- O robô se reinicia e começa a se autorreparar... e em seguida repara os comandos da nave e volta ao normal.
- O player fica feliz (laços afetivos)...
- Inimigos
- Os amigos se juntam na jornada (não comandados por você)...

BOSS 5 – explicação sobre o boss

- O inimigo morre... e você recebe ordem de pousar no planeta... e você vai...

FASE 13 – (o planeta inimigo)

- Você pouso e verifica que o planeta é muito bonito... tem atmosfera...
- Você parte para a exploração e encontra a cidade inimiga....
- Enfrenta alguns deles e entra na “casa” do BOSS

BOSS FINAL...

- Você é atacado de cara... não há nada a ser dito aqui... é matar ou morrer...
- O boss morre (na verdade fica com 1 de vida)
- Diálogos sobre a motivação...

- O boss fala que viu o que a raça humana fez com o seu planeta, e que só estava tentando se proteger...
- Que a única forma de manter a sua raça era impedindo que os humanos chegassem até NOME_DO_PLANETA....
- O inimigo cai...
- Dilema do player...
- O player volta para a nave mãe e diz que é melhor procurar outro planeta, pois aquele não possui condições de suportar a vida...

FIM

Figura 5: Extrato do GDD

Devido ao prazo dado para a execução da APS, nos vemos obrigados a reduzir o escopo e implementar apenas a fase 4.

E assim concluímos o GDD, e passamos para a fase de desenvolvimento propriamente dito.

3.3 A TRILHA SONORA

Trilhas sonoras são essenciais na imersão do jogador, então iniciamos o desenvolvimento de músicas autorais com a finalidade de enriquecer a experiência.

Foi utilizado o software GarageBand na confecção da música do menu, e da primeira fase.

O GarageBand é um software de produção musical desenvolvido pela Apple Inc. Ele é uma parte do pacote de aplicativos iLife para MacOS e iOS. O GarageBand é projetado para permitir que usuários com ou sem experiência em música possam criar, gravar e misturar suas próprias músicas. (Apple. *GarageBand*, 2023).

Foram criadas 5 músicas, das quais 2 foram utilizadas inicialmente.

As músicas procuram estimular os sentidos dos jogadores, procurando transparecer uma atmosfera de solidão em alguns momentos, e de agitação durante as fases que envolve contato com inimigos.

3.4 A ARTE CONCEITUAL

De posse da paleta de cores AAP-64 e com o uso do software Aseprite, iniciamos os sprites base para que o código do jogo pudesse refletir as diversas mecânicas pretendidas.

Estabelecemos que a resolução do gráfico do jogo seguiria o mesmo padrão do console Super Nintendo com o Mode7 ativado (512 x 448), com uma única diferença, ao invés de adotar o padrão de tela de 4:3, buscaríamos implementar a resolução *widescreen* (16:9), o que resultou em uma resolução final de 520 x 292 com um *scale* de 2x para se adequar as telas atuais.

Desta forma, ficou decidido que para fins de testes, o player inicialmente teria um espaço de 16 x 16 pixels (que depois seria substituído por 32 x 48) e as demais entidades deveriam possuir 32 x 32 pixels.

Tínhamos em mente que ao desenvolver o código do jogo, ele deveria se adequar facilmente a qualquer medida utilizada nos sprites (elementos gráficos do jogo).



Figura 6: Arte inicial do Player (para menus)



Figura 7: Sprite do player em 16 x 16 pixels



Figura 8: Sprites de entidades do jogo 32 x 32 px



Figura 9: Sprites de inimigos 32 x 32 px

Da mesma forma, criamos as artes das telas de Menu, História, Controles, LeaderBoard e Sobre:



Figura 10: Arte da Tela de Menu

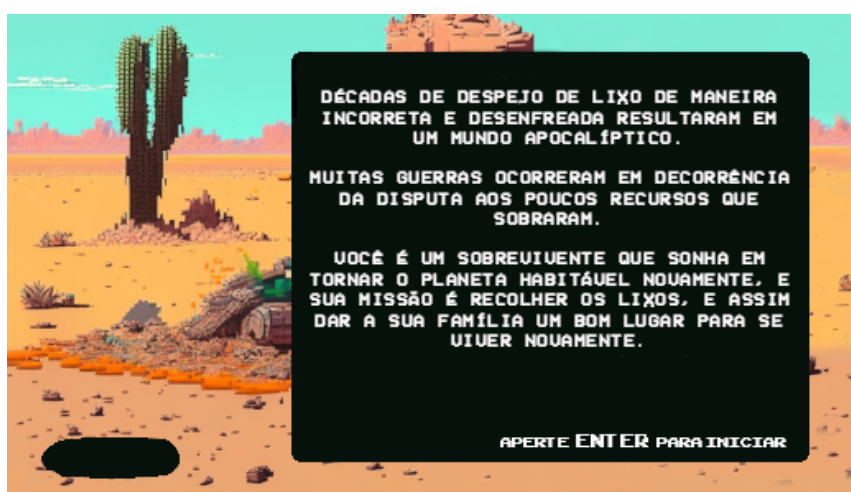


Figura 11: Arte da tela de história



Figura 12: Arte da tela LEADERBOARDS

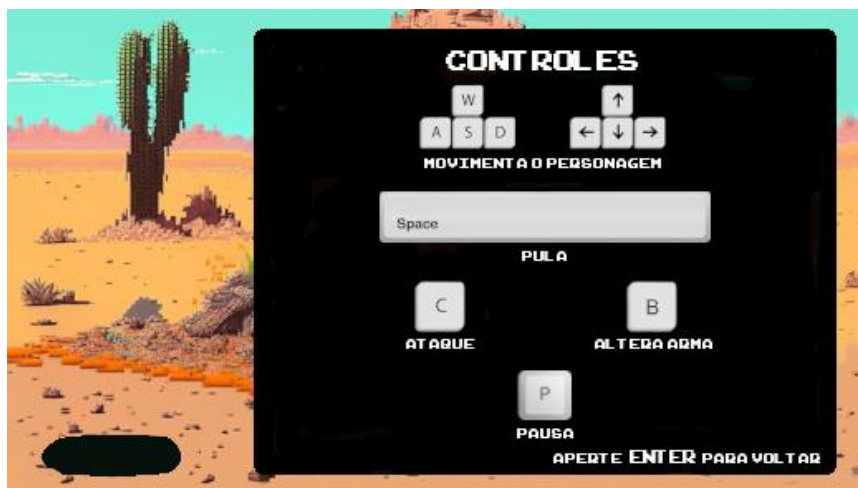


Figura 13: Arte da tela de controles



Figura 14: Arte da tela Sobre

Ainda fora criados elementos como o fundo da tela, vetores de nuvens e diversos elementos de cenários.

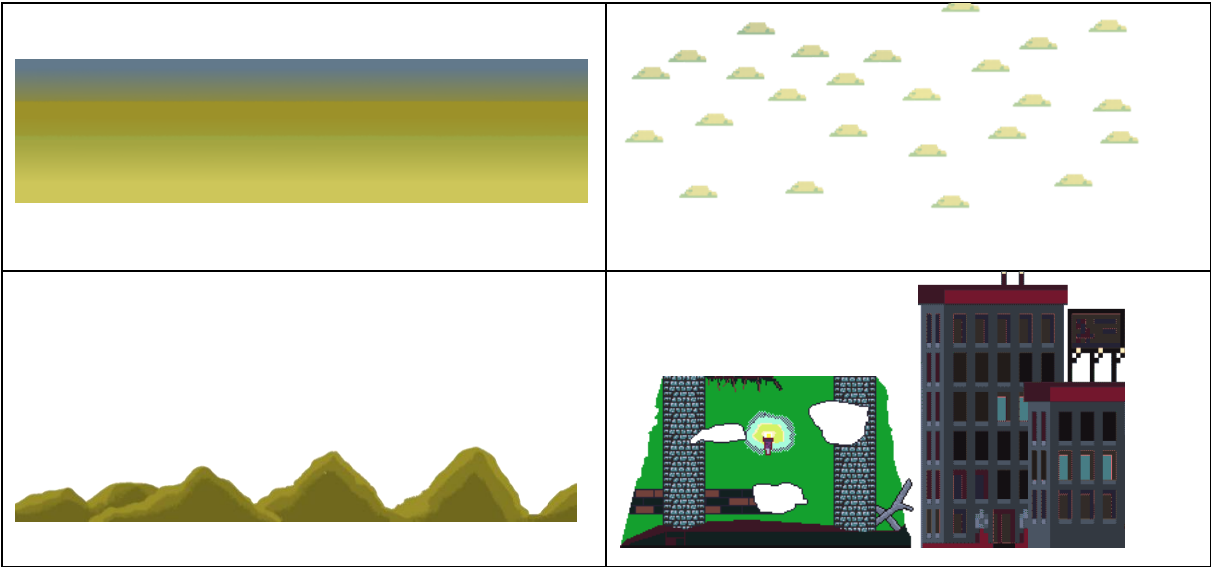


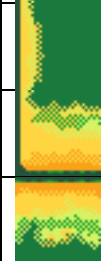

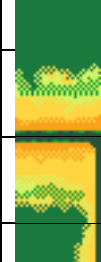



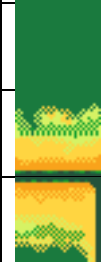

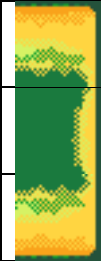

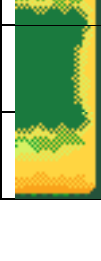





































































Figura 15: Imagens diversas de Fundo

Foi criado o mapa de entidades com base na mecânica de mapeamento decidida para o jogo, e que será explicada mais adiante:

		18b229	corChaoEsquerdoTopo
		Ac4839	corChaoEsquerdo
		946d4a	corChaoEsquerdoFundo
		6A91a4	corChaoBaseTopo
		086910	corChaoNucleo
		4A2420	corChaoBaseFundo
		CD3420	corChaoDireitoTopo
		AC6920	corChaoDireito
		6722AC	corChaoDireitoFundo

		7845AC	corChaoIsoladoTopo
		45ACAC	corNucleoBifurcaChaoIsoladoTopo
		D57D29	corNucleoBifurcaChaoIsoladoFundo
		AAD5C0	corChaoIsoladoFundo
		55d595	corNucleoConverteDireitaChaoIsoladoTopo
		D5D580	corNucleoConverteDireitaChaoIsoladoFundo
		D55555	corNucleoConverteEsquerdaChaoIsoladoTopo
		392420	corNucleoConverteDireitaChaoIsoladoFundo
	Player	FFFF00	corPlayer
	Inimigo 1	494900	corInimigo1
	Inimigo 2	606000	corInimigo2
		Ee8529	corTijoloDeserto
		ee8fbe	corEscadaTopo
		C4759d	corEscada
		9b5d7c	corEscadaBase
		38385d	corGrama
		4D4D80	corGalhoSeco
		4D8080	corEspinhas
		808033	corPlacaSave
		F6efef	corKitHealth
	Ceu/Fundo	639bFF	corCeu

		Void	000000	
			F600f6	corTrashBag
			FF9d52	corJuncaoTopoEsquerda
			FF713d	corJuncaoFundoEsquerda
			FF4551	corJuncaoTopoDireita
			FF73F8	corJuncaoFundoDireita
			61ff88	corJuncaoDupla1
			C8FF52	corJuncaoDupla2
			496372	corJuncaoSimplesLateralTopoDireita
			793b34	corChaoIsoladoDireita
			355240	corChaoIsoladoEsquerda
			F22778	corJuncaoSimplesFundoDireita
			0E5050	corFundoDarkBrickBase
			646464	corFundoDarkBrickEsquerdo
			494949	corFundoDarkBrickDireito
			0e1052	corFundoDarkBrickBrokenBase1
			2d3425	corWallFundo1
			585392	corJuncaoBuEsquerdaBaixo
			666663	corJuncaoBuDireitaBaixo
			666248	corBuSimples
			157920	corPredioFundo1














		2d2c7a	corChaoIsoladoMeioVertical
		1a3917	corPedra1
		97df67	corVidaExtra
		827719	corAmmuBox
		3e7682	corMountainParalax
		203766	corJuncaoUmBlocoDireita
		364366	corJuncaoUmBlocoEsquerda

Figura 16: Mapping Tiles

Posteriormente a arte do player foi alterada para ocupar 32 x 48 pixels.

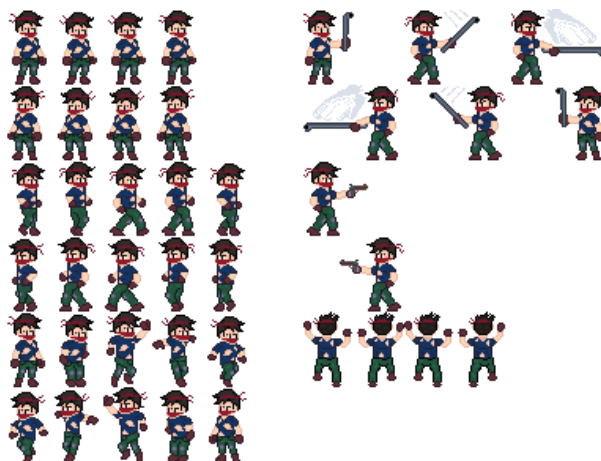


Figura 17: Sprite do player em 32 x 48px

Partimos então para a definição dos objetos do jogo.

3.5 TRATANDO SOBRE OBJETOS

Para criar um jogo em Java, precisávamos criar os objetos que fariam parte do software, bem como os métodos pretendidos e as relações entre os objetos.

Para confeccionar o UML, pensamos inicialmente da seguinte forma.

- Entity -Tudo no jogo é uma entidade , essa é a super classe

- Entidades Interativas – tudo aquilo que de uma forma ou outra tem interação com o player (Inimigos, Munição extra, *Check Point*, Escadas, Espinhos, Kit de Vida, Tiros , Lixo e Vida Extra).

- Entidades não solidas – Compõem o cenário (Céu, tijolos de fundo, galhos secos, Montanhas (paralaxe), nuvens, Partículas, prédios, poeira, e paredes de fundo).

- Player – é a entidade controlada pelo jogador.

- Entidades Solidas – São aquelas que são afetadas pela gravidade e possuem efeitos de travamento de movimento.

- Spritesheet seria a classe que controlaria a parte de gráficos do jogo.

- Game – Classe padrão, que faz o jogo rodar.

- Controles, Game Over, História, LeaderBoard, Menu e Sobre - classes de controle de estado do jogo.

- UserInterface – Classe para controlar as informações disponíveis para o player.

- Audio – controlador de sons do jogo.

- Camera – controla o foco da tela no player.

- Empty – responsável pela renderização do vazio do jogo.

- Tile – responsável pelo gerenciamento dos itens de um mundo (level).

- Mundo – responsável pela criação da fase em si.

Seria criado um pacote chamado res (abreviação de *resources*) onde ficariam os arquivos de sprite e sons.

Inicialmente, trabalhamos com os atributos de forma pública, pois gostaríamos de facilitar a integração com as diversas classes, no entanto ao longo do projeto mudaríamos alguns para *protected* e *private*.

Devido ao grande número de atributos e métodos das classe, construímos o UML de forma separada para facilitar a visualização.

Com relação as entidades, definimos o seguinte UML:

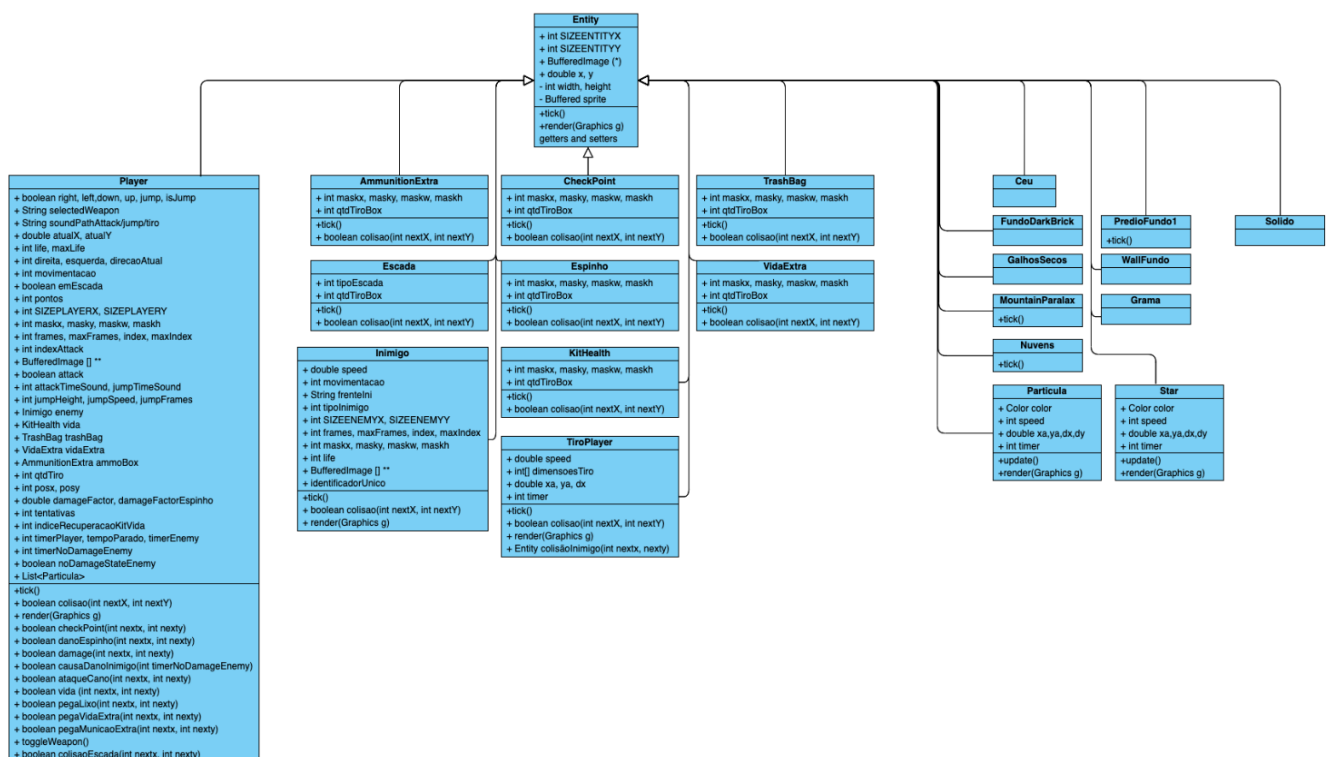


Figura 18: UML de entidades

Por fim foi feito o UML das demais Classes:

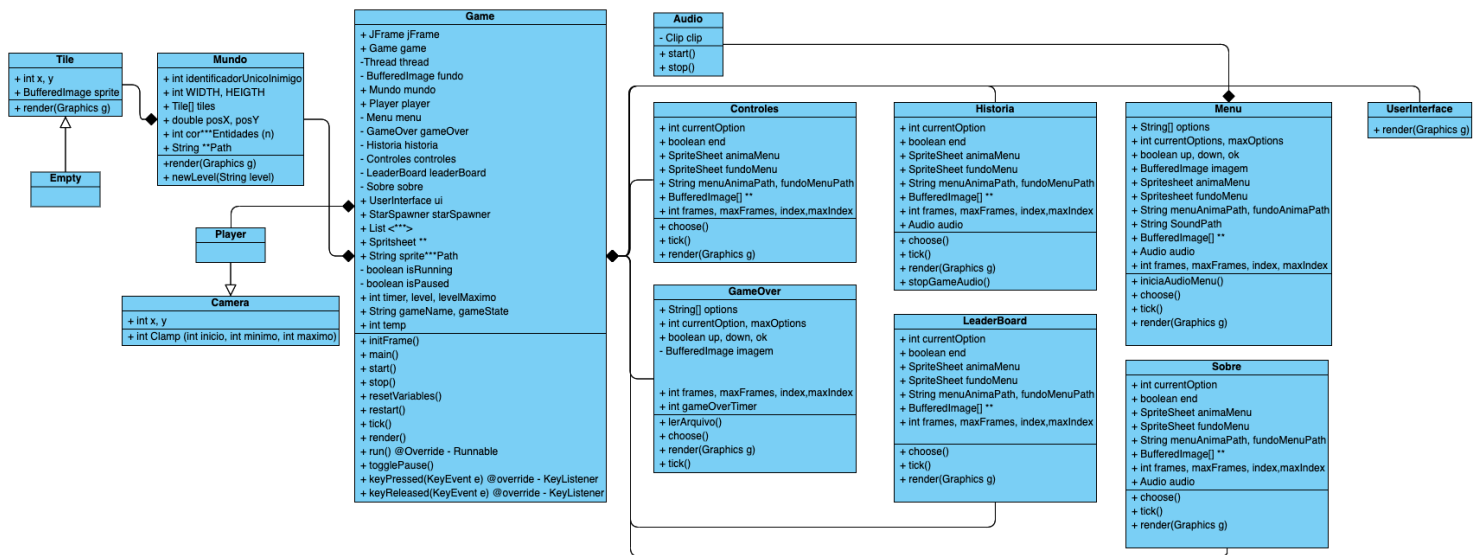


Figura 19: UML das demais classes

Uma das peculiaridades da linguagem Java é a sua abordagem em relação à quantidade de arquivos de código produzidos. Em Java, é comum que cada classe seja definida em um arquivo separado, seguindo uma convenção estrita de nomenclatura.

Diferentemente de algumas outras linguagens de programação, onde é possível ter várias classes definidas em um único arquivo, o Java adota a abordagem "uma classe, um arquivo". Isso significa que cada classe Java deve ser definida em um arquivo com o mesmo nome da classe, seguido da extensão ".java".

Essa abordagem promove uma organização mais clara e estruturada do código fonte. Cada arquivo .java contém a definição de uma única classe, tornando mais fácil localizar, entender e manter o código. Além disso, essa prática está alinhada com o princípio de encapsulamento, em que cada classe deve ser responsável por uma única funcionalidade.

Quando um programa Java é compilado, cada arquivo .java é transformado em um arquivo .class correspondente, contendo o *bytecode* executável. Esses arquivos .class podem ser agrupados em um arquivo JAR (Java Archive) ou serem disponibilizados individualmente.

Embora essa abordagem de um arquivo por classe possa resultar em uma maior quantidade de arquivos em projetos Java maiores, ela oferece benefícios significativos em termos de organização e manutenção do código. Também é possível criar pacotes e diretórios para organizar as classes relacionadas de forma lógica.

O nosso projeto contaria com 36 classes inicialmente, tendendo a reduzir ou aumentar a medida que o projeto fosse expandido e refinado.

3.6 ESTRATÉGIA E DESENVOLVIMENTO DO CÓDIGO

Nesta seção, trataremos sobre as classes mais importantes do jogo, e as estratégias adotadas para que ele funcionasse de forma satisfatória. Vale ressaltar que existem inúmeras possibilidades para a execução de tarefas no Java, e que um planejamento adequado, realizado com mais tempo, poderia resultar em um código mais limpo e eficiente.

Devido ao tempo reduzido e em virtude da maioria do grupo não ter conhecimento da linguagem Java (até mesmo porque só fora ensinado os conceitos básicos de POO no período considerado), o jogo sofre com relação ao refinamento, mas apresenta consistência em sua execução.

3.6.1 A CLASSE GAME.JAVA

Como estratégia, definimos que a classe main ficaria no Game, e dele seria a responsabilidade de renderizar o Frame principal onde o jogo aconteceria.

Além disso também seria responsabilidade da classe, receber e repassar para as demais classes os eventos relativos ao uso do teclado no jogo, através da implementação (interface) `KeyListener`.

A classe `Game.java` faria também o gerenciamento do *tick* (ou *update*) usando *Threads*, fixando a execução dos *ticks* (termo usado em programação, que faz referência a um milissegundo) em 60 FPS (*frames per second*).

Todas as listas de entidades usadas no jogo seriam carregadas assim que o main instanciasse pelo construtor padrão um novo jogo.

Vale ressaltar que o jogo faria o gerenciamento do seu status de execução através das classes auxiliares: Menu, LeaderBoard, Controles, Sobre, Historia e GameOver, através da variável gameState, que mudaria conforme a necessidade.

O construtor padrão de Game, criaria então a lista de todas as entidades e por fim, instanciaría o Mundo do jogo (referente a fase inicial, que seria 1).

Para fins de organização o jogo começaria com o Menu, onde o usuário poderia escolher um novo jogo, ver as pontuações máximas, ver os controles, ver os membros da equipe de desenvolvimento e por fim, sair do jogo.

No caso de seleção de um novo jogo, seria aberta a interface de história, para que em seguida o jogo (level1) fosse carregado.

```
public Game() {
    menu = new Menu();
    gameOver = new GameOver();
    historia = new Historia();
    controles = new Controles();
    sobre = new Sobre();
    leaderBoard = new LeaderBoard();
    // Listener de teclado
    addKeyListener(this);
    // ajusta a preferência do tamanho do container do jogo
    this.setPreferredSize(new Dimension(WIDTH * SCALE, HEIGHT * SCALE));
    // inicia o Frame
    initFrame();
    // instancia a ui
    ui = new UserInterface();
    // instancia o fundo
    fundo = new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
    // define o sprite a ser usado pelas entidades
    //sprites para as entidades base
    sprite = new Spritsheet(spriteGamePath);
    //sprite para jogador
    spritePlayer = new Spritsheet(spritePlayerPath);
    // sprite para inimigos
    spriteEnemy = new Spritsheet(spriteEnemyPath);
    // sprite para ceu
    ceu = new Spritsheet(spriteCeuParam);
    // sprite de paralaxe
    mountain = new Spritsheet(spriteMountainPath);
    // Sprite de fundo(cenário)
    wallFundo1 = new Spritsheet(spriteFundo1Path);
    predioFundo1 = new Spritsheet(spriteFundoPredio1Path);
    //sprite de nuvem
```

```

nuvens = new Spritsheet(spriteNuvemPath);
// lista de entidades
entidades = new ArrayList<>();
// lista de ceu
ceuVetor = new ArrayList<>();
// lista de montanhas
mountainVetor = new ArrayList<>();
// lista de vetor de fundo
wallFundo1Vetor = new ArrayList<>();
predioFundo1Vetor = new ArrayList<>();
// lista de nuvens
nuvemVetor = new ArrayList<>();
// lista de health kit
kitHealth = new ArrayList<>();
// lista de trash bags
trashBags = new ArrayList<>();
vidasExtras = new ArrayList<>();
ammunitionExtras = new ArrayList<>();
// lista de savepoints
checkPoints = new ArrayList<>();
// lista de inimigos
inimigo = new ArrayList<>();
// lista de grama
grama = new ArrayList<>();
espinhos = new ArrayList<>();
// lista de escadas
escada = new ArrayList<>();
// lista de darkBrick
darkBricksFundo = new ArrayList<>();
tirosPLayer = new ArrayList<>();

// instancia o player (de acordo com a posição inicial no sprite)
player = new Player(0, 0, Player.SIZEPLAYERX, Player.SIZEPLAYERY, spritePlayer.getSprite(0, 0,
Player.SIZEPLAYERX, Player.SIZEPLAYERY));
// adiciona o player em entidades (só pode haver 1)
entidades.add(player);
// spawner de poeira
starSpawner = new StarSpawner();
// por fim carrega o mundo....
mundo = new Mundo(levelPath);
}

```

Figura 20: construtor de Game

O método main, instanciará o novo jogo e o faria rodar:

```

public static void main(String[] args) {
    game = new Game();
    game.start();
}

```

Figura 21: Método main

Código responsável por criar o *frame* onde o jogo seria iniciado:

```

public void initFrame() {
    JFrame = new JFrame(gameName);
    JFrame.add(this);
    // não permito que a janela seja redimensionada
    // poderá mudar ao longo do tempo, mas temos apenas 3 meses para terminar o jogo
}

```

```

jFrame.setResizable(false);
jFrame.pack();
// aqui eu ajusto a posição relativa da janela (geralmente no meio da tela)
jFrame.setLocationRelativeTo(null);
// comportamento esperado quando eu aperto o botão de fechar a janela
jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
jFrame.setFocusableWindowState(true);
jFrame.addWindowListener(new WindowAdapter() {
    @Override
    public void windowActivated(WindowEvent e) {
        jFrame.requestFocusInWindow();
    }
});
jFrame.setFocusable(true);
// pede o foco para a janela assim que ela aparecer
KeyboardFocusManager.getCurrentKeyboardFocusManager().clearGlobalFocusOwner();
KeyboardFocusManager.getCurrentKeyboardFocusManager().clearFocusOwner();
jFrame.requestFocusInWindow();
jFrame.toFront();
jFrame.requestFocus();
// torno visível a janela
jFrame.setVisible(true);
}

```

Figura 22: método responsável pelo frame

O método `start()` inicia a `Thread` de forma sincronizada.

Isso significa que apenas uma *thread* pode executar esse método por vez, evitando que várias threads entrem em conflito ao chamar esse método simultaneamente.

```

public synchronized void start() {
    isRuning = true;
    thread = new Thread(this);
    thread.start();
}

```

Figura 23: método `start`

O ciclo de atualizações (*ticks*) fica a cargo do método `run`.

```

@Override
public void run() {
    long lastTime = System.nanoTime();
    double amountOfTicks = 60.0f;
    double ns = 1000000000L / amountOfTicks;
    double delta = 0;
    double timer = System.currentTimeMillis();

    while (isRuning) {
        long now = System.nanoTime();
        delta += (now - lastTime) / ns;
        lastTime = now;
        while (delta >= 1) {
            if (!isPaused) { // verificação de pausa
                tick();
                render();
            }
            delta--;
        }
    }
}

```



```

    }
    delta--;
    }
    if (!isPaused) { // verificação de pausa
    if (System.currentTimeMillis() - timer >= 1000) {
    timer += 1000;
    }
    }

    // loop de espera para limitar o uso da CPU
    try {
    Thread.sleep(1);
    } catch (InterruptedException e) {
    e.printStackTrace();
    }
    }
    stop();
}

```

Figura 24: Método run

O método run() é uma implementação da interface Runnable. Ele representa o código que será executado em paralelo em uma thread separada quando essa thread for iniciada.

Analisando o método, temos o seguinte:

1. Inicialização de variáveis:

- lastTime: armazena o tempo atual em nanossegundos.
- amountOfTicks: representa a quantidade de atualizações por segundo desejadas (neste caso, 60 atualizações por segundo).
- ns: calcula o intervalo de tempo em nanossegundos entre cada atualização.
- delta: controla o tempo decorrido desde a última atualização.
- timer: armazena o tempo atual em milissegundos.

2. Loop principal:

- Enquanto a variável isRunning for verdadeira, o loop continuará executando.
- Calcula o tempo decorrido desde a última atualização e adiciona esse valor a delta.
- Dentro de um segundo loop, enquanto delta for maior ou igual a 1, realiza a atualização lógica do jogo chamando os métodos tick() e render(). Esses

métodos provavelmente estão implementados na mesma classe ou em outra classe e representam a lógica de jogo e a renderização dos elementos na tela.

- Verifica se um segundo se passou (usando a variável timer) e, se passou, atualiza o timer.
- Realiza um pequeno atraso de 1 milissegundo para limitar o uso da CPU e evitar um processamento excessivo.

3. Encerramento:

- Quando o loop principal é interrompido (a variável isRunning se torna falsa), o método stop() é chamado, encerrando a execução adequada da thread.

O método run garante que as atualizações sejam realizadas em uma taxa consistente, limitando o uso da CPU e permitindo um controle mais preciso do tempo de atualização.

3.6.2 A CLASSE MUNDO.JAVA e ENTITY.JAVA

Essas duas classes são responsáveis por trazer todos os elementos do jogo, haja vista que tudo dentro do jogo pode ser considerado como uma entidade, e o mundo renderiza as entidades existentes.

O conceito de mundo está relacionado ao nível do jogo (level) isso porque o mundo apresenta várias fases e cabe a ele imprimir essa fase no frame.

A estratégia principal aqui foi utilizar um arquivo do tipo png (*portable network graphics*) que contém pixels de diversas cores, e de acordo com o MappingTiles, seriam convertidos em entidades do jogo.

O mundo faria a leitura do arquivo pixel a pixel e interpretaria a cor existente, inserindo nas arrays de entidades o tile respectivo.



Figura 25: exemplo de level design



Figura 26: Mapping Tiles

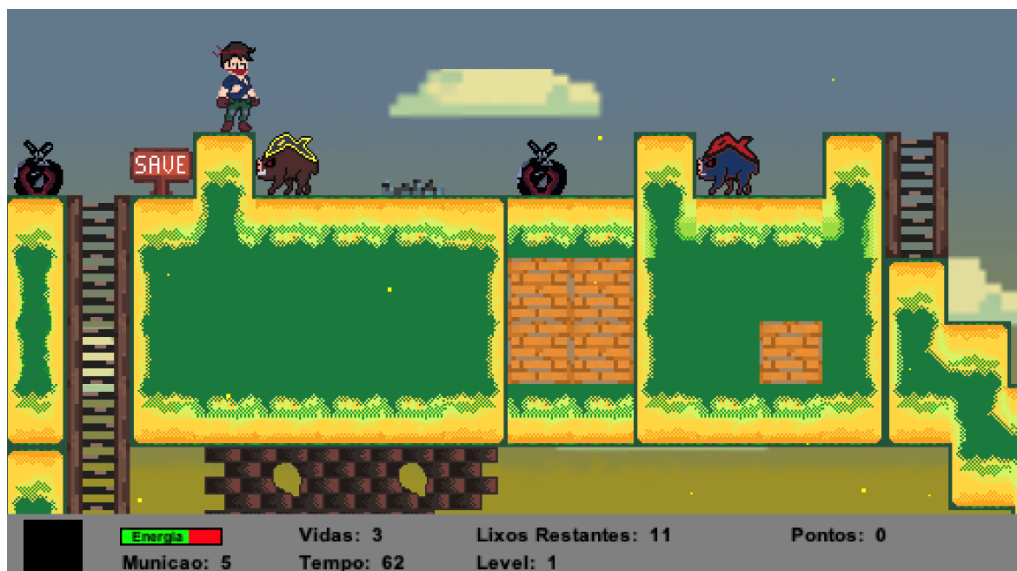


Figura 27: Renderização da fase

Dessa forma é possível observar que cada pixel existente em level1.png é substituído por uma entidade.

```
public int corEscadaTopo = 0xFFee8fbe, corEscada = 0xFFc4759d, corEscadaBase = 0xFF9b5d7c, corChaoNucleo = 0xFF086910, corChaoEsquerdo = 0xFFac4839, corChaoEsquerdoTopo = 0xFF18b229, corChaoEsquerdoFundo = 0xFF946d4a, corJuncaoTopoEsquerda = 0xFFff9d52, .....
```

Figura 28: extrato de referência das cores

Construtor do Mundo.java

```
public Mundo(String path) {
    try {
        // mapeamento do mundo (de acordo com os pixel da base da fase)
        BufferedImage level = ImageIO.read(Objects.requireNonNull(getClass().getResource(path)));
        int[] pixels = new int[level.getWidth() * level.getHeight()];
        tiles = new Tile[level.getWidth() * level.getHeight()];
        // tamanho da fase (arquivo .png)
        WIDTH = level.getWidth();
        HEIGHT = level.getHeight();
        // pega as cores dos pixel do arquivo base e prepara para a conversão
        level.getRGB(0, 0, level.getWidth(), level.getHeight(), pixels, 0, level.getWidth());
        // passa pelo exito x e y do arquivo de fase
        for (int x = 0; x < level.getWidth(); x++) {
            posX = x;
            for (int y = 0; y < level.getHeight(); y++) {
                posY = y;
                int pixelAtual = pixels[x + (y * level.getWidth())];
                // popula os tiles vazios
                tiles[x + (y * WIDTH)] = new Empty(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.empty);
                if (pixelAtual == corFundoDarkBrickBase) {
                    FundoDarkBrick fundoDarkBrick = new FundoDarkBrick(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY,
                        Entity.SIZEENTITYX, Entity.SIZEENTITYY, Entity.fundoDarkBrickBase);
                    Game.darkBricksFundo.add(fundoDarkBrick);
                }

                else if (pixelAtual == corFundoDarkBrickBrokenBase1) {
                    FundoDarkBrick fundoDarkBrick = new FundoDarkBrick(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY,
                        Entity.SIZEENTITYX, Entity.SIZEENTITYY, Entity.fundoDarkBrickBrokenBase1);
                    Game.darkBricksFundo.add(fundoDarkBrick);
                }

                else if (pixelAtual == corFundoDarkBrickEsquerdo) {
                    FundoDarkBrick fundoDarkBrick = new FundoDarkBrick(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY,
                        Entity.SIZEENTITYX, Entity.SIZEENTITYY, Entity.fundoDarkBrickEsquerdo);
                    Game.darkBricksFundo.add(fundoDarkBrick);
                }

                else if (pixelAtual == corFundoDarkBrickDireito) {
                    FundoDarkBrick fundoDarkBrick = new FundoDarkBrick(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY,
                        Entity.SIZEENTITYX, Entity.SIZEENTITYY, Entity.fundoDarkBrickDireito);
                    Game.darkBricksFundo.add(fundoDarkBrick);
                }

                else if (pixelAtual == corPlacaSave) {
                    CheckPoint checkPoint = new CheckPoint(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
                        Entity.SIZEENTITYY, Entity.save);
                    Game.checkPoints.add(checkPoint);
                }

                else if (pixelAtual == corEscadaTopo) {
                    Escada escada = new Escada(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
                        Entity.SIZEENTITYY, 3, Entity.escadaTopo);
                    Game.escada.add(escada);
                } else if (pixelAtual == corEscada) {
                    Escada escada = new Escada(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
                        Entity.SIZEENTITYY, 2, Entity.escada);
                    Game.escada.add(escada);
                } else if (pixelAtual == corEscadaBase) {
                    Escada escada = new Escada(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
                        Entity.SIZEENTITYY, 1, Entity.escadaBase);
                    Game.escada.add(escada);
                } else if (pixelAtual == corChaoIsoladoTopo) {
                    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
                        Entity.SIZEENTITYY, Entity.chaoIsoladoTopo);
                    Game.entidades.add(solido);
                }
            }
        }
    }
}
```

```

}
else if (pixelAtual == corChaoIsoladoEsquerda) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoIsoladoEsquerda);
    Game.entidades.add(solido);
}
else if (pixelAtual == corChaoIsoladoDireita) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoIsoladoDireita);
    Game.entidades.add(solido);
}
else if (pixelAtual == corChaoIsoladoMeioVertical) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoIsoladoMeioVertical);
    Game.entidades.add(solido);
}
else if (pixelAtual == corPedra1) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.pedra1);
    Game.entidades.add(solido);
}
else if (pixelAtual == corChaoIsoladoFundo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoIsoladoFundo);
    Game.entidades.add(solido);
} else if (pixelAtual == corChaoEsquerdo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoEsquerdo);
    Game.entidades.add(solido);
} else if (pixelAtual == corChaoDireito) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoDireito);
    Game.entidades.add(solido);
} else if (pixelAtual == corChaoEsquerdoTopo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoEsquerdoTopo);
    Game.entidades.add(solido);
} else if (pixelAtual == corChaoEsquerdoFundo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoEsquerdoFundo);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoBuEsquerdaBaixo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoBuEsquerdaBaixo);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoBuDireitaBaixo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoBuDireitaBaixo);
    Game.entidades.add(solido);
}
else if (pixelAtual == corBuSimples) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.buSimples);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoTopoEsquerda) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoTopoEsquerda);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoFundoEsquerda) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoFundoEsquerda);
    Game.entidades.add(solido);
}

```

```

}
else if (pixelAtual == corJuncaoTopoDireita) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoTopoDireita);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoUmBlocoDireita) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoUmBlocoDireita);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoUmBlocoEsquerda) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoUmBlocoEsquerda);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoFundoDireita) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoFundoDireita);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoDupla1) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoDupla1);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoDupla2) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoDupla2);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoSimplesLateralTopoDireita) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoSimplesLateralTopoDireita);
    Game.entidades.add(solido);
}
else if (pixelAtual == corJuncaoSimplesFundoDireita) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.juncaoSimplesFundoDireita);
    Game.entidades.add(solido);
}
else if (pixelAtual == corChaoDireitoTopo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoDireitoTopo);
    Game.entidades.add(solido);
} else if (pixelAtual == corChaoDireitoFundo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoDireitoFundo);
    Game.entidades.add(solido);
} else if (pixelAtual == corChaoNucleo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoNucleo);
    Game.entidades.add(solido);
} else if (pixelAtual == corChaoBaseTopo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoNormalTopo);
    Game.entidades.add(solido);
} else if (pixelAtual == corChaoBaseFundo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.chaoNormalFundo);
    Game.entidades.add(solido);
} else if (pixelAtual == corNucleoConverteDireitaChaoIsoladoTopo) {
    Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
    Entity.SIZEENTITYY, Entity.nucleoConverteDireitaChaoIsoladoTopo);
    Game.entidades.add(solido);
} else if (pixelAtual == corNucleoConverteDireitaChaoIsoladoFundo) {

```

```

Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.nucleoConverteDireitaChaoIsoladoFundo);
Game.entidades.add(solido);
} else if (pixelAtual == corNucleoConverteEsquerdaChaoIsoladoTopo) {
Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.nucleoConverteEsquerdaChaoIsoladoTopo);
Game.entidades.add(solido);
} else if (pixelAtual == corNucleoConverteEsquerdaChaoIsoladoFundo) {
Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.nucleoConverteEsquerdaChaoIsoladoFundo);
Game.entidades.add(solido);
} else if (pixelAtual == corNucleoBifurcaChaoIsoladoTopo) {
Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.nucleoBifurcaChaoIsoladoTopo);
Game.entidades.add(solido);
} else if (pixelAtual == corNucleoBifurcaChaoIsoladoFundo) {
Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.nucleoBifurcaChaoIsoladoFundo);
Game.entidades.add(solido);
} else if (pixelAtual == corTijoloDeserto) {
Solido solido = new Solido(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.tijoloDeserto);
Game.entidades.add(solido);
} else if (pixelAtual == corKitHealth) {
KitHealth kitHealth = new KitHealth(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.kitHealth);
Game.kitHealth.add(kitHealth);
} else if (pixelAtual == corTrashBag) {
TrashBag trashBag = new TrashBag(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.trashBag);
Game.trashBags.add(trashBag);
} else if (pixelAtual == corVidaExtra) {
VidaExtra vidaExtra = new VidaExtra(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.vidaExtra);
Game.vidasExtras.add(vidaExtra);
}
else if (pixelAtual == corAmmuBox) {
AmmunitionExtra ammunitionExtra = new AmmunitionExtra(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY,
Entity.SIZEENTITYX, Entity.SIZEENTITYY, Entity.ammunitionExtra);
Game.ammunitionExtras.add(ammunitionExtra);
}
else if (pixelAtual == corGrama) {
Grama grama = new Grama(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.grama);
Game.entidades.add(grama);
} else if (pixelAtual == corEspinhas) {
Espinha espinha = new Espinha(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.espinha);
Game.espinhas.add(espinha);
} else if (pixelAtual == corGalhoSeco) {
GalhosSecos galho = new GalhosSecos(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYY, Entity.galhoSeco);
Game.entidades.add(galho);
} else if (pixelAtual == corInimigo1 || pixelAtual == corInimigo2) {
int tipoInimigo = 0;
if (pixelAtual == corInimigo1) {
tipoInimigo = 1;
} else if (pixelAtual == corInimigo2) {
tipoInimigo = 2;
}
}

// aqui é um incremento para por o inimigo no lugar correto
// tem que reajustar quando os inimigos passarem a ocupar 32 pixels ao invés de 16
int incremento = 1;
if (Inimigo.SIZEENEMYX == 16){

```

```

incremento = 2;}
Inimigo inimigo = new Inimigo(identificadorUnicoInimigo,x * Inimigo.SIZEENEMYX*incremento, y *
Inimigo.SIZEENEMY*incremento, Inimigo.SIZEENEMYX, Inimigo.SIZEENEMY, tipoInimigo, Entity.inimigo);
identificadorUnicoInimigo++;
Game.inimigo.add(inimigo);
} else if (pixelAtual == corCeu) {
Ceu ceu = new Ceu(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY, Entity.SIZEENTITYX, Entity.SIZEENTITYX,
Entity.ceu);
Game.ceuVetor.add(ceu);
}
else if (pixelAtual == corMountainParalax) {
MountainsParalax mountainsParalax = new MountainsParalax(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY,
Entity.SIZEENTITYX, Entity.SIZEENTITYX, Entity.mountainParalax);
Game.mountainVetor.add(mountainsParalax);
}
else if (pixelAtual == corWallFundo1) {
WallFundo1 wallFundo1 = new WallFundo1(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY,
Entity.SIZEENTITYX, Entity.SIZEENTITYX, Entity.wallFundo1);
Game.wallFundo1Vetor.add(wallFundo1);
}
else if (pixelAtual == corPredioFundo1) {
PredioFundo1 predioFundo1 = new PredioFundo1(x * Entity.SIZEENTITYX, y * Entity.SIZEENTITYY,
Entity.SIZEENTITYX, Entity.SIZEENTITYX, Entity.predioFundo1);
Game.predioFundo1Vetor.add(predioFundo1);
}

else if (pixelAtual == corPlayer) {
Game.player.setX(x * Player.SIZEPLAYERX);
Game.player.setY(y * Player.SIZEPLAYERY);
}

}
}
}
Nuvens nuvem = new Nuvens(Entity.SIZEENTITYX, Entity.SIZEENTITYY, Entity.SIZEENTITYX,
Entity.SIZEENTITYX, Entity.nuvens);
Game.nuvemVetor.add(nuvem);
} catch (IOException e) {
e.printStackTrace();
}
}
}

```

Figura 29: Construtor do mundo

É possível observar no código acima o laço de repetição de acordo com a imagem fornecida. Com base no que foi apresentado, não importa o tamanho nem a largura do arquivo `level1.png`, pois esse laço garante que independente disso, todas as entidades serão renderizadas.

Já a classe `Entity.java`, realiza a distinção quanto ao tipo de bloco a ser inserido, categorizando-o em sólido, interativo, player, cenário dentre outros.

Também é responsabilidade da classe `Entity`, localizar a posição no sprite específico para a realização das renderizações, e por fim, sendo ela a super classe, atribuir os base para as demais sub-classes.


```

public class Entity {

    // posições das entidades no sprite x, y
    static int[] posChaoNucleo = {32, 64}, posChaoIsoladoTopo = {96, 0}, posChaoIsoladoFundo = {96, 96},
    posNucleoBifurcaChaoIsoladoTopo = {96, 32}, posNucleoBifurcaChaoIsoladoFundo = {96, 64},
    posNucleoConverteDireitaChaoIsoladoTopo = {128, 32}, posNucleoConverteEsquerdaChaoIsoladoTopo = {160, 32},
    posNucleoConverteDireitaChaoIsoladoFundo = {128, 64}, posNucleoConverteEsquerdaChaoIsoladoFundo = {160, 64},
    posChaoEsquerdoTopo = {0, 32}, posChaoEsquerdo = {0, 64}, posChaoDireitoTopo = {64, 32},
    posChaoDireito = {64, 64}, posChaoNormalTopo = {32, 32}, posChaoNormalFundo = {32, 96},
    posJuncaoBuEsquerdaBaixo = {224, 96}, posJuncaoBuDireitaBaixo = {288, 96}, posBuSimples = {256, 96},
    posChaoEsquerdoFundo = {0, 96}, posChaoDireitoFundo = {64, 96}, posTijoloDeserto = {32, 0},
    posEmpty = {0, 0}, posGramma = {0, 160}, posEscadaTopo = {0, 192}, posEscada = {32, 192},
    posEscadaBase = {64, 192}, posEspinho = {0, 128}, posGalhoSeco = {32, 160}, posKitHealth = {64, 160},
    posSavePoint = {64, 128}, posTrashBag = {32, 128}, posJuncaoTopoEsquerda = {192, 32},
    posJuncaoFundoEsquerda = {192, 64}, posJuncaoTopoDireita = {224, 32}, posJuncaoFundoDireita = {224, 64},
    posJuncaoDupla1 = {256, 32}, posJuncaoDupla2 = {256, 64}, posJuncaoSimplesLateralTopoDireita = {192, 96},
    posChaoIsoladoEsquerda = {160, 0}, posChaoIsoladoDireita = {128, 0}, posJuncaoSimplesFundoDireita = {64, 0},
    posFundoDarkBrickEsquerdo = {96, 128}, posFundoDarkBrickDireito = {160, 128}, posFundoDarkBrickBase = {128, 128},
    posFundoDarkBrickBrokenBase1 = {96, 160}, posCahoIsoladoMeioVertical = {160, 96}, posPedra1 = {128, 96},
    posVidaExtra = {128, 160}, posAmmunitionExtra = {160, 160}, posJuncaoUmBlocoDireita = {288, 32},
    posJuncaoUmBlocoEsquerda = {288, 64};

    public static int SIZEENTITYX = 32, SIZEENTITYY = 32;
    // buffer de todas as entidades e seus posicionamentos no sprite)
    public static BufferedImage chaoNucleo = Game.sprite.getSprite(posChaoNucleo[0], posChaoNucleo[1], SIZEENTITYX,
    SIZEENTITYY);
    public static BufferedImage chaoIsoladoTopo = Game.sprite.getSprite(posChaoIsoladoTopo[0], posChaoIsoladoTopo[1],
    SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage save = Game.sprite.getSprite(posSavePoint[0], posSavePoint[1], SIZEENTITYX,
    SIZEENTITYY);
    public static BufferedImage chaoIsoladoEsquerda = Game.sprite.getSprite(posChaoIsoladoEsquerda[0],
    posChaoIsoladoEsquerda[1], SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage chaoIsoladoDireita = Game.sprite.getSprite(posChaoIsoladoDireita[0],
    posChaoIsoladoDireita[1], SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage chaoIsoladoFundo = Game.sprite.getSprite(posChaoIsoladoFundo[0],
    posChaoIsoladoFundo[1], SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage nucleoBifurcaChaoIsoladoTopo =
    Game.sprite.getSprite(posNucleoBifurcaChaoIsoladoTopo[0], posNucleoBifurcaChaoIsoladoTopo[1], SIZEENTITYX,
    SIZEENTITYY);
    public static BufferedImage nucleoBifurcaChaoIsoladoFundo =
    Game.sprite.getSprite(posNucleoBifurcaChaoIsoladoFundo[0], posNucleoBifurcaChaoIsoladoFundo[1], SIZEENTITYX,
    SIZEENTITYY);
    public static BufferedImage nucleoConverteDireitaChaoIsoladoTopo =
    Game.sprite.getSprite(posNucleoConverteDireitaChaoIsoladoTopo[0], posNucleoConverteDireitaChaoIsoladoTopo[1],
    SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage nucleoConverteEsquerdaChaoIsoladoTopo =
    Game.sprite.getSprite(posNucleoConverteEsquerdaChaoIsoladoTopo[0],
    posNucleoConverteEsquerdaChaoIsoladoTopo[1], SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage nucleoConverteDireitaChaoIsoladoFundo =
    Game.sprite.getSprite(posNucleoConverteDireitaChaoIsoladoFundo[0], posNucleoConverteDireitaChaoIsoladoFundo[1],
    SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage nucleoConverteEsquerdaChaoIsoladoFundo =
    Game.sprite.getSprite(posNucleoConverteEsquerdaChaoIsoladoFundo[0],
    posNucleoConverteEsquerdaChaoIsoladoFundo[1], SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage chaoIsoladoMeioVertical = Game.sprite.getSprite(posCahoIsoladoMeioVertical[0],
    posCahoIsoladoMeioVertical[1], SIZEENTITYX, SIZEENTITYY);
    public static BufferedImage pedra1 = Game.sprite.getSprite(posPedra1[0], posPedra1[1], SIZEENTITYX,
    SIZEENTITYY);

    .....
}

```

Figura 30: Extrato da classe Entity

3.6.3 A CLASSE PLAYER.JAVA

A classe `Player.java` contém grande parte da lógica do jogo, e pode ser considerada como uma das mais importantes.

Dentre as características mais importantes, podemos citar as seguintes:

- **Movimentação básica:** O jogador pode se mover para a direita, esquerda, para baixo e para cima.
- **Armas selecionadas:** Há uma variável chamada `"selectedWeapon"` que indica a arma selecionada pelo jogador.
- **Sons:** Há caminhos de arquivos de áudio para diferentes sons, como ataque, pulo e tiro.
- **Vida:** O jogador tem uma vida inicial e uma vida máxima, representadas pelas variáveis `"life"` e `"maxLife"`.
- **Direção e movimentação:** Há variáveis para controlar a direção atual e o estado de movimentação do jogador.
- **Gravidade e pulo:** Há lógica para aplicar a gravidade e permitir que o jogador pule.
- **Ataque:** O jogador pode atacar inimigos, tanto com um "cano" quanto com tiros (se a arma selecionada permitir).
- **Sprites:** Há vários arrays de `BufferedImage` para representar os diferentes sprites do jogador em diferentes estados, como movimento, pulo e ataque.
- **Colisões:** Há lógica para verificar colisões com objetos do jogo, como escadas e inimigos.
- **Outros elementos:** A classe também inclui outras entidades relacionadas ao jogador, como inimigos, kits de vida, sacos de lixo, vida extra e munição extra.

Com o uso da `KeyListener`, as variáveis de controle, como *left*, *right*, *up* e *down*, todas do tipo *boolean* assumem valor *true* quando as respectivas teclas são pressionadas e *false* caso sejam soltas.

Além disso, por se tratar de um jogo de plataforma, e por sofrer ação simulada da gravidade, que puxa o player para uma entidade sólida caso não esteja em contato com uma, os comandos de *up* e *down*, só fazem efeito caso exista uma colisão com alguma entidade do tipo Escada.

Foi implementado um sistema de armas com munição limitada para a pistola (arma de fogo) e com uso livre (cano), tudo isso visando gerar uma dificuldade no gerenciamento de recursos, pois o ataque a distância em teoria sempre garante maior segurança ao player.

Foi implementado um sistema de falha na arma de fogo, no qual o disparo pode ou não acontecer, e o tiro ao colidir com qualquer entidade sólida ou inimigo é removido do jogo.

Os inimigos causam dano ao tocar no player e o player causa dano nos inimigos apenas ao conseguir executar um ataque válido.

Foram implementados diversos testes que permitem as diversas interações do inimigo com o mundo ao seu redor.

```
// Gravidade
// Situação de ação da GRAVIDADE e MOVIMENTO de PLAYER em ESCADAS (pois desafia a gravidade)
if (!colisao((int) x, (int) (y + 1)) && !isJump && !emEscada) {
    // não existe a colisão não estou em pulo e nem em escada
    // situação normal do player
    // se não existe colisão, eu sou afetado pela gravidade
    // quando estou em uma escada, a gravidade não me afeta (posso subir e descer)
    // quando estou em pulo, eu desafio a gravidade pelo tempo do pulo
    y += 4; // índice de velocidade de queda
} else {
    // regras para movimento em escadas
    if (emEscada && !colisao((int) x, (int) (y))) {
        if (down) {
            y += speed;
            y = (int) y;
        }
        if (up) {
            y -= speed;
            y = (int) y;
        }
        movimentacao = 1;
        timerPlayer = 0;
    } else {
        // nesse caso, eu tento me colocar no nível y da entidade sólida ao qual eu colido
        // evita que o player fique travado no chão
        Rectangle playerRect = new Rectangle((int) x + maskx, (int) y + masky, maskw, maskh);

        for (int i = 0; i < Game.entidades.size(); i++) {
            Entity entidade = Game.entidades.get(i);
            if (entidade instanceof Solido) {
```

```

Rectangle solido = new Rectangle(entidade.getX() + maskx, entidade.getY() + masky, Entity.SIZEENTITYX,
Entity.SIZEENTITYY);
if (playerRect.intersects(solido)) {
this.y = entidade.getY() - (SIZEPLAYERY);
}
}
}
}
}
}

```

Figura 31: Extrato do teste de controle de movimento e gravidade

4. O CÓDIGO FONTE DO JOGO

Em virtude da grande extensão do código do jogo, e por causa da característica do Java de trabalhar com um arquivo por classe, tornamos disponível os conteúdo total dessa aps no seguinte endereço:

<https://github.com/tiagobrilhante/jogoapsjava>

Bastando baixar o conteúdo da pasta scr, e por fim rodar o Game.java

5. GALERIA DE IMAGENS



Figura 32: Tela de menu

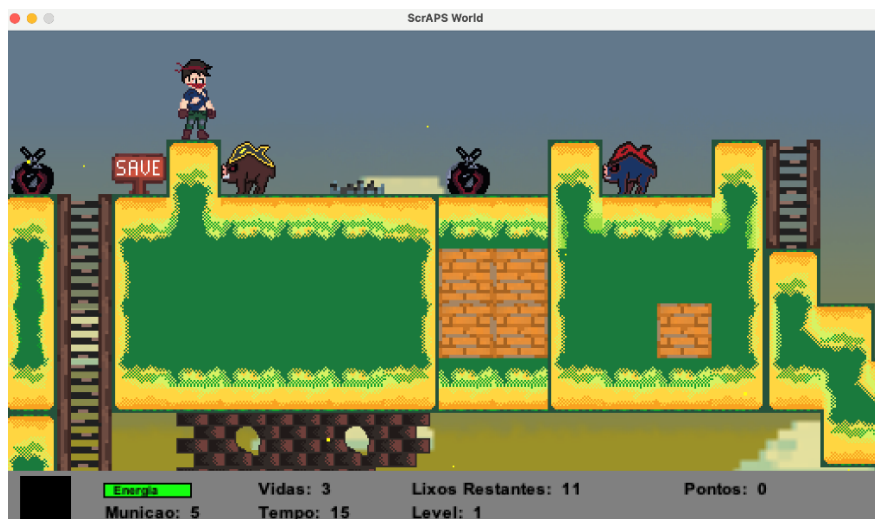


Figura 33: Renderização do mundo



Figura 34: Movimentação em escadas

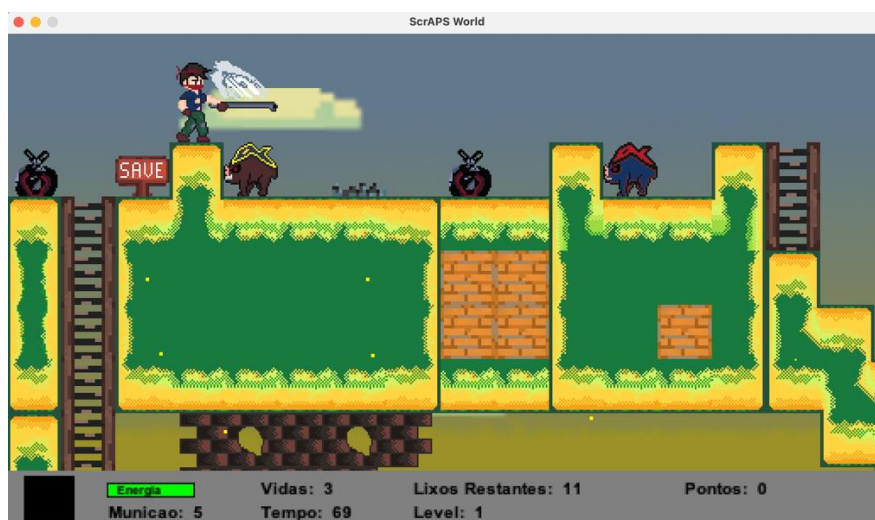


Figura 35: Ataque com cano



Figura 36: Inimigo destruído

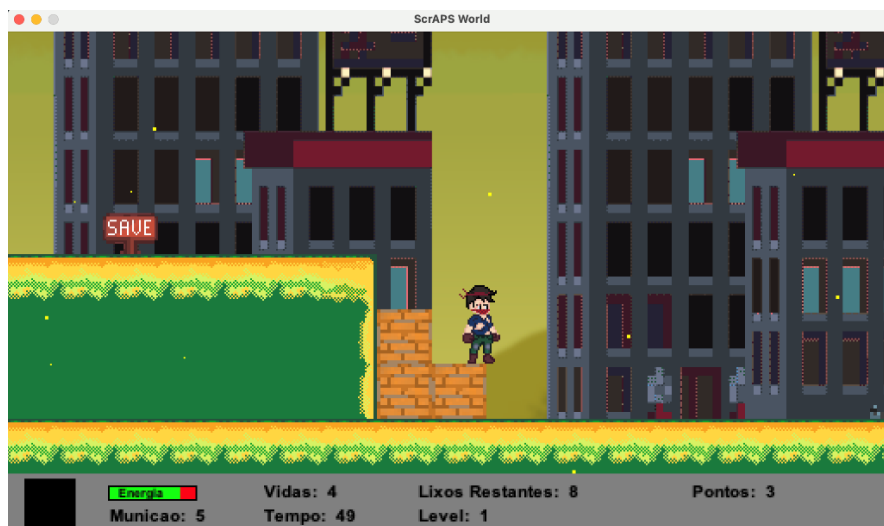


Figura 37: Elementos do cenário

6. CONCLUSÃO

O presente trabalho procurou demonstrar a criação de um Jogo em Java, utilizando apenas as bibliotecas nativas da linguagem, e com foco na educação ambiental, como proposto no arquivo de orientação da APS.

Em conclusão, a criação do jogo em Java foi um desafio significativo devido à falta de conhecimento e habilidades específicas por parte da maioria dos membros do grupo. Foi evidente que a faculdade não forneceu o treinamento e a instrução adequados para lidar com um projeto tão complexo. No entanto, mesmo diante dessa dificuldade, um membro do grupo mostrou-se capacitado e assumiu a responsabilidade de desenvolver o código do jogo. Sua dedicação e habilidades foram essenciais para superar os obstáculos e concluir o projeto com sucesso.

Ao enfrentar essa situação, pudemos compreender a importância de adquirir conhecimentos além do currículo acadêmico. A experiência evidenciou a necessidade de buscar recursos externos, como tutoriais, cursos online ou projetos pessoais, para aprimorar nossas habilidades técnicas.

Além disso, a criação do jogo em Java foi uma oportunidade valiosa para aprendermos sobre trabalho em equipe, comunicação efetiva e gerenciamento de projetos. A necessidade de distribuir as tarefas de forma adequada e de cooperar uns com os outros ficou clara. Embora o desenvolvimento do código tenha sido liderado por apenas um membro, todos contribuíram com ideias, arte, design e testes, desempenhando um papel fundamental para o sucesso do projeto.

Apesar dos desafios encontrados, a criação do jogo em Java foi uma experiência enriquecedora que nos permitiu adquirir habilidades valiosas e compreender melhor a importância do conhecimento prático no campo da programação e entender os conceitos de orientação a objetos. Com base nessa experiência, estamos motivados a continuar aprimorando nossas habilidades técnicas e buscando oportunidades de aprendizado fora da sala de aula.

7. REFERÊNCIAS

WIKIPEDIA- JOGO, 2023. Disponível em: <https://pt.wikipedia.org/wiki/Jogo>, Acesso em 15/05/2023

UGO ROVEDA, 2022. Disponível em: <https://kenzie.com.br/blog/programacao-orientada-a-objetos/>, Acesso em 15/05/2023.

"Object-Oriented Programming: Concepts and Principles" - GeeksforGeeks. Disponível em: <https://www.geeksforgeeks.org/object-oriented-programming-oops-concept-in-java/>, Acesso em 15/05/2023.

"The Importance of Object-Oriented Programming" - Udacity. Disponível em: <https://www.udacity.com/blog/2021/02/the-importance-of-object-oriented-programming.html/>, Acesso em 15/05/2023.

"Why Is Object-Oriented Programming Important?" - Guru99. Disponível em: <https://www.guru99.com/oops-concept-java.html/>, Acesso em 15/05/2023.

"The Importance of Object-Oriented Programming" - Simplilearn. Disponível em: <https://www.simplilearn.com/object-oriented-programming-basics-explained-article/>, Acesso em 15/05/2023.

Oracle - The Java™ Tutorials - Disponível em: <https://docs.oracle.com/javase/tutorial/java/concepts/class.html>
Acesso em 15/05/2023.

MDN Web Docs - JavaScript - Disponível em:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_with_Objects
Acesso em 15/05/2023

Tutorials Point - Encapsulation in Java - Disponível em:
https://www.tutorialspoint.com/java/java_encapsulation.htm

Acesso em 15/05/2023

W3Schools - Java Inheritance: Disponível em:
https://www.w3schools.com/java/java_inheritance.php

Acesso em 15/05/2023

Baeldung - Polymorphism in Java: Disponível em: <https://www.baeldung.com/java-polymorphism>

Acesso em 15/05/2023

Tutorialspoint - Object-Oriented Analysis & Design, Disponível em:
https://www.tutorialspoint.com/object_oriented_analysis_design/ooad_object_oriented_abstraction.htm, Acesso em 15/05/2023

Oracle - Java Gaming, Disponível em: <https://www.oracle.com/java/technologies/java-gaming.html>

Acesso em 15/05/2023

DZone - Top Java Game Development Libraries , Disponível em:
<https://dzone.com/articles/top-java-game-development-libraries>

Acesso em 15/05/2023

Oracle - Java Community, Disponível em:
<https://community.oracle.com/community/developer> , Acesso em 15/05/2023

Joe Matar. "Pixel Art: The Golden Age of Video Game Art" - Artsy, disponível em:
<https://www.artsy.net/article/artsy-editorial-pixel-art-golden-age-video-game-art>

Acesso em 15/05/2023

Dan Whitehead. "The history of pixel art, from the bitmap to the present" - PC Gamer, disponível em: <https://www.pcgamer.com/the-history-of-pixel-art-from-the-bitmap-to-the-present/> , Acesso em 15/05/2023

Apple. GarageBand. Disponível em: <https://www.apple.com/mac/garageband/>, Acesso em 15/05/2023.

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, iniciação científica, trabalhos individuais e em grupo, práticas de ensino e outras)

NAME: Guilherme Pinheiro Guelle

RA: N015985 CURSO: Ciência da Computação

CAMPUS: Manaus SEMESTRE: 2º TURNO: Noturno

[illegible]

TOTAL DE HORAS:

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, Iniciação Científica, trabalhos Individuais e em grupo, práticas de ensino e outras)

RA: T48110-4 CURSO: CIÊNCIA DA COMPUTAÇÃO

TURNNO: NOT VUENO

[illegible]

TOTAL DE HORAS: 143 hs

Atividades Práticas Supervisionadas (laboratórios, atividades em biblioteca, iniciação científica, trabalhos individuais e em grupo, práticas de ensino e outras)

RA: F3447F8

TURNO: Noturno

SEMESTRE: 3º período

TOTAL DE HORAS: 60h