**CENTRO FEDERAL DE EDUCAÇÃO TECNOLÓGICA DE MINAS GERAIS**

**(CEFET-MG) - CAMPUS CONTAGEM**

**MARIA EDUARDA OLIVEIRA PEREIRA**

**OTTO EMANUEL MARTINS ABREU**

**VINICIUS DOMINGOS CANDIDO**

# RELATÓRIO TÉCNICO REFERENTE AO TRABALHO

# "TP - JOGOS 2D – TÉCNICAS DE PROGRAMAÇÃO"

**CONTAGEM**

**2023**

**MARIA EDUARDA OLIVEIRA PEREIRA**

**OTTO EMANUEL MARTINS ABREU**

**VINICIUS DOMINGOS CANDIDO**

# RELATÓRIO TÉCNICO REFERENTE AO TRABALHO

# "TP - JOGOS 2D – TÉCNICAS DE PROGRAMAÇÃO"

Relatório técnico apresentado à disciplina de Laboratório e Técnicas de Programação I do Centro Federal de Educação Tecnológico de Minas Gerais, ministrada pelo professor Alisson Rodrigo dos Santos.

Professores: Alisson Santos e Patrícia Resende

**CONTAGEM**

**2023**

**LISTA DE FIGURAS**

**SUMÁRIO**

# 1. INTRODUÇÃO

## 1.1. Mini Doom

O "Mini Doom", retratado na Figura 01, é uma criação amplamente reconhecida que surgiu em 2016, como uma homenagem criativa ao icônico jogo "Doom" desenvolvido pela id Software. Este notável projeto foi concebido por Felipe Porcel, um devoto admirador da série de jogos "Doom" e um talentoso desenvolvedor independente. O "Mini Doom" se destaca principalmente por sua abordagem em 2D, um contraponto ao jogo original em 3D. Essa distinção implica que os jogadores, ao invés de explorarem ambientes tridimensionais, percorrem níveis bidimensionais, permitindo movimentação tanto lateral quanto verticalmente.

**FIGURA 01: Mini Doom**


fonte: Calavera Studio

Além disso, o jogo preserva a essência da jogabilidade frenética do "Doom" original, desafiando os jogadores a enfrentar incansáveis hordas de inimigos demoníacos e a empunhar armas icônicas. O “Mini Doom” conseguiu, assim, consolidar seu lugar de destaque na rica história dos jogos eletrônicos. Essa audaciosa experiência inspirou uma série de recriações, todas elas em busca de capturar a essência do clássico “Doom”.

Ademais, o “Mini Doom” por sua vez acabou se destacando como uma recriação singular e continua a influenciar inúmeros desenvolvedores e jogadores. Seu sucesso perdura ao longo do tempo, e seu apelo transcende gerações, destacando como a paixão pela nostalgia e o amor

pelos desafios intensos permanecem vivos na comunidade de jogos. Essas recriações não apenas celebram a herança do “Doom” original, mas também demonstram a capacidade de uma experiência de jogo habilmente concebida para atravessar as eras, continuando a inspirar novas iterações e perpetuando o legado de um dos títulos mais emblemáticos da indústria de jogos eletrônicos.

## 1.2. Objetivos

Em vista disso, o propósito deste documento é apresentar o desenvolvimento proposto para a recriação do jogo, no contexto da disciplina de Laboratório e Técnicas de Programação 1, sob a orientação do professor Alisson Rodrigo dos Santos. Nesse sentido, o enfoque principal deste projeto consiste na exploração dos conhecimentos adquiridos em programação orientada a objetos, em conjunto com a utilização da biblioteca SFML. Para tal finalidade, foram estabelecidos os seguintes objetivos:

### 1.2.1 Objetivos do Projeto

- Modelagem e Estruturação
- Hierarquia de Classes
- Herança e Composição
- Nomenclatura Significativa
- Utilização da Biblioteca SFML

### 1.2.1 Objetivos Específicos para  a Recriação do “Mini Doom”

- Plataforma 2D
- Animação de Personagens
- Diversidade de Inimigos
- Mecânicas de Objetos e Armadilhas
- Técnica: *TileMaps*

Portanto, este relatório tem o propósito de informar sobre os objetivos alcançados durante o processo de recriação do jogo  "Mini Doom" e apresentar o resultado final obtido. Dois objetivos fundamentais foram estabelecidos para este projeto. O primeiro deles envolveu a reprodução dos aspectos essenciais presentes no jogo original, mantendo a sua identidade e familiaridade para os jogadores. O segundo objetivo visou à incorporação de novas

funcionalidades destinadas a enriquecer a experiência do usuário, proporcionando desafios e atrativos adicionais. O projeto "Mini Doom" representou um equilíbrio cuidadoso entre a preservação da essência do jogo original e a introdução de elementos criativos que contribuíram para tornar a experiência de jogo mais envolvente.

Além disso, é importante salientar que o ponto culminante deste projeto foi a realização de testes rigorosos, conduzidos de maneira iterativa ao longo de várias etapas do desenvolvimento. Esses testes desempenharam um papel vital na identificação e correção de problemas, assegurando, assim, a qualidade final do jogo recriado. Comprometidos com a excelência e a satisfação dos jogadores, focando especial atenção à garantia de que todas as funcionalidades implementadas funcionassem de maneira adequada, proporcionando uma experiência agradável.

## 2. IMPLEMENTAÇÃO

No processo de criação de um jogo, o sucesso muitas vezes depende da combinação harmoniosa de diversas ferramentas de desenvolvimento, da meticulosa organização do código e da criação de classes e funcionalidades que dão vida à experiência de jogo. Assim sendo, foram explorados esses elementos essenciais de criação de jogos, mergulhando nas ferramentas que impulsionam o desenvolvimento, examinando estratégias de organização do código e aprofundando-nos em detalhes específicos, como a função "Game" e a utilidade do "*TileMap*". Portanto, será analisado como esses componentes se interligam para criar o jogo, denominado de “Jão do Sertão”, e proporcionar experiências interativas inesquecíveis.

### 2. 1. Ferramentas de Desenvolvimento

Jão do Sertão é um emocionante jogo que oferece uma nova abordagem ao clássico “Mini Doom”, proporcionando uma experiência única para os jogadores. No desenvolvimento deste jogo, a equipe utilizou uma série de ferramentas e tecnologias. Primeiramente, escolheram o GCC ( *GNU Compiler Collection* ) na versão 7.3.0, uma escolha fundamental, visto que o GCC é uma coleção de compiladores de código aberto amplamente utilizados. Essa versão específica foi selecionada para compilar o código do jogo. Além disso, o Mingw 64, que inclui o GCC, foi empregado como ambiente de desenvolvimento para Windows, permitindo assim a criação de uma versão Windows do jogo.

Outro componente essencial foi a biblioteca SFML na versão 2.5.1, ou *Simple and Fast Multimedia Library*. Esta biblioteca multiplataforma oferece uma interface simples e intuitiva para o desenvolvimento de jogos. Com a SFML, a equipe conseguiu implementar eficientemente os elementos visuais e sonoros do Jão do Sertão. Além disso, o uso de XML foi essencial, permitindo a criação do mapa do jogo de forma estruturada e flexível. Isso possibilitou a definição detalhada do cenário, a disposição de objetos e a configuração de elementos interativos, contribuindo significativamente para a jogabilidade e a experiência dos jogadores.

Para garantir uma colaboração eficaz, o grupo utilizou o Discord como uma plataforma de comunicação fundamental. O Discord é uma aplicação de bate-papo por voz e texto amplamente adotada por desenvolvedores de jogos e equipes de projetos de software. No Discord, os membros da equipe puderam se comunicar em tempo real, realizar reuniões virtuais, compartilhar informações importantes e tirar dúvidas uns com os outros.

Além disso, o GitHub foi utilizado como uma plataforma para compartilhar e manter o código do jogo atualizado. O GitHub proporcionou um ambiente de trabalho colaborativo que permitiu aos desenvolvedores compartilharem o código-fonte, rastrearem as mudanças feitas, gerenciarem as versões do projeto e garantirem que todos estivessem trabalhando com a versão mais recente do jogo. Para acessar a comunidade de trabalho no GitHub, você pode seguir o link: GitHub do Projeto. Isso permitiu que qualquer membro da equipe ou interessado acompanhasse o progresso do projeto e contribuísse de forma eficaz.

Em resumo, o desenvolvimento do Jão do Sertão foi uma colaboração eficiente entre diversas ferramentas modernas, como GCC, SFML, Discord e GitHub. Graças a essa gama de recursos e à coordenação efetiva da equipe, foi possível trazer o jogo à vida e proporcionar aos jogadores uma experiência única no mundo dos jogos.

### 2. 2. Organização do Código

No processo de desenvolvimento do jogo "Jão do Sertão", foi adotada a estruturação do código em diferentes arquivos .cpp e .hpp com o objetivo de assegurar a organização e manutenção eficientes do projeto. Essa prática contribuiu para facilitar a separação das funcionalidades específicas de cada aspecto do jogo, permitindo que cada arquivo se concentrasse em áreas distintas, como a mecânica do jogo e renderização gráfica. Além disso,

promoveu a reutilização de códigos de trabalhos anteriores, otimizando o processo de desenvolvimento e simplificando a manutenção do projeto. Para equipes de desenvolvimento, essa estruturação modular também melhorou a colaboração interna, permitindo que cada membro se dedicasse a seu conjunto de arquivos, e ainda otimizou a compilação, acelerando o progresso do projeto em direção ao objetivo de criar uma experiência interativa e envolvente para os jogadores.

Os arquivos .cpp e .hpp estão organizados da seguinte forma:

- **background.cpp e background.hpp:** Esses arquivos contêm a implementação e a definição da classe responsável por gerenciar o plano de fundo do jogo. Eles controlam a renderização e os efeitos visuais do cenário, criando a atmosfera desejada.
- **enemy.cpp e enemy.hpp:** Aqui, encontra-se a implementação e a definição da classe que representa os inimigos no jogo. Esses arquivos contêm a lógica que governa o comportamento dos inimigos, como movimento, ataques e interações com o jogador.
- **main.cpp:** Este arquivo é o ponto de entrada principal do jogo. Ele contém a função principal (main) que inicia o jogo e coordena as principais funcionalidades, como inicialização, loop de jogo e encerramento.
- **map.cpp e map.hpp:** Nesses arquivos, estão presentes a implementação e a definição da classe responsável pela construção e gerenciamento dos mapas e níveis do jogo. Eles lidam com a criação de cenários, obstáculos e a disposição geral do ambiente de jogo.
- **player.cpp e player.hpp:** Esses arquivos contêm a implementação e a definição da classe que representa o jogador. Eles incluem a lógica do personagem controlado pelo jogador, incluindo movimento, interações com objetos e eventos específicos do jogador.
- **bullet.hpp:** Este arquivo define a classe que representa projéteis ou balas no jogo. Ele inclui informações sobre o comportamento das balas, como direção, velocidade e danos causados.
- **game.hpp:** Neste arquivo, estão presentes as definições necessárias para a classe principal do jogo. Ele coordena os principais elementos do jogo, a interface do usuário e a interação entre os diferentes componentes.
- **entity.hpp:** Encarregado de tudo o que é partilhado pelas entidades no jogo.

## 2. 3. Método

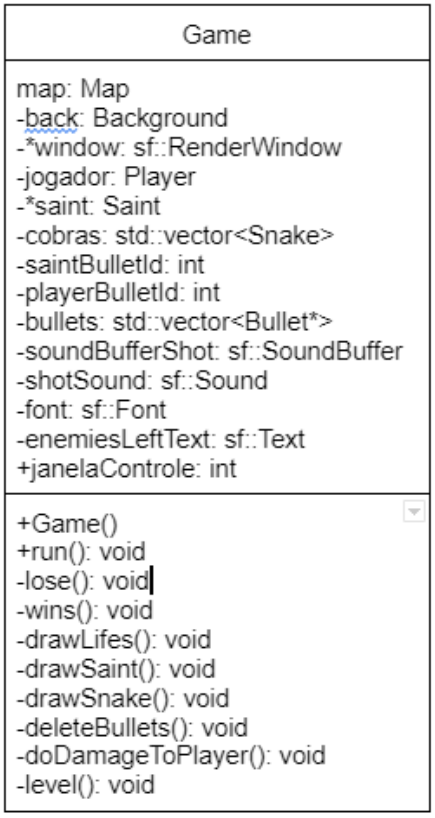
Neste código de recriação do Mini Doom, cada componente desempenha um papel específico e fundamental na materialização deste icônico jogo. Cada linha de código contribui para aspectos cruciais, como a mecânica de movimento do jogador, a renderização dos gráficos, a interação com oponentes e outros elementos essenciais. A seguir, será apresentado as principais características deste código e como cada parte contribui para uma envolvente experiência.

### 2.3.1. O Jogo

#### 2.3.1.1. Class Game

A classe Game, Figura 02, no jogo "Jão do Sertão" desempenha um papel central na gestão e execução do jogo.

**FIGURA 02: UML *GAME***




Fonte: Autoral

**Construtor *Game():***

- Este construtor é responsável por inicializar um objeto da classe `Game`.
- Ele cria a janela do jogo, carrega recursos como texturas e sons, define o mapa, o fundo, o jogador, os inimigos e configura o texto de exibição. Além disso, ele define as condições iniciais do jogo.

**Função *void run():***

- Esta função é o loop principal do jogo e controla a execução do jogo.
- Dentro do loop, ela trata eventos de fechamento da janela e atualiza o estado do jogo chamando a função `level()` quando o jogo está em execução. Além disso, ela gerencia as transições entre diferentes estados do jogo, como vitória e derrota.

**Função *void level():***

- Esta função é responsável por gerenciar o nível atual do jogo.
- No nível, ela controla a renderização de elementos, como o fundo, balas, inimigos, o jogador, o mapa e as vidas restantes. Ela também lida com a lógica do jogo, como colisões entre objetos e verificações de vitória ou derrota.

**Função *void lose():***

- Esta função é chamada quando o jogador perde o jogo.
- Ela exibe uma tela de derrota com uma imagem de fundo e um botão que permite ao jogador fechar o jogo. Ela também detecta a interação do jogador com o botão de fechar o jogo.

**Função *void wins():***

- Esta função é chamada quando o jogador vence o jogo.
- Ela exibe uma tela de vitória com uma imagem de fundo e um botão que permite ao jogador fechar o jogo. Ela também detecta a interação do jogador com o botão de fechar o jogo.

**Função *void drawLifes():***

- Esta função desenha as vidas do jogador na tela.
- Ela posiciona os sprites das vidas com base na posição do jogador e no número de vidas restantes.

**Função *void drawSaint():***

- Esta função desenha o personagem "Saint" na tela.
- Ela verifica se o personagem está vivo, atualiza sua posição e animação, e o renderiza na janela.

**Função void *drawSnake():***

- Esta função desenha as cobras inimigas na tela.
- Ela verifica se as cobras estão vivas, atualiza a animação delas, dimensiona e posiciona as sprites das cobras, e as renderiza na janela.

**Função *void deleteBullets():***

- Esta função remove balas do jogo após um determinado período de tempo.
- Ela verifica o tempo de vida das balas do jogador e do personagem "Saint" e as remove do vetor de balas se o tempo limite for atingido.

**Função *void doDamageToPlayer():***

- Esta função verifica se o jogador sofreu dano.
- Ela verifica colisões entre as balas do "Saint", das cobras inimigas e o jogador, e aplica dano ao jogador quando necessário.

No geral, a classe `Game` gerencia a lógica de execução do jogo "Jão do Sertão". Ela controla a renderização de elementos, as interações do jogador, verificações de vitória e derrota, e é responsável por manter o loop principal do jogo em funcionamento. É o componente central para coordenar todos os elementos do jogo e oferecer uma experiência de jogo completa aos jogadores.

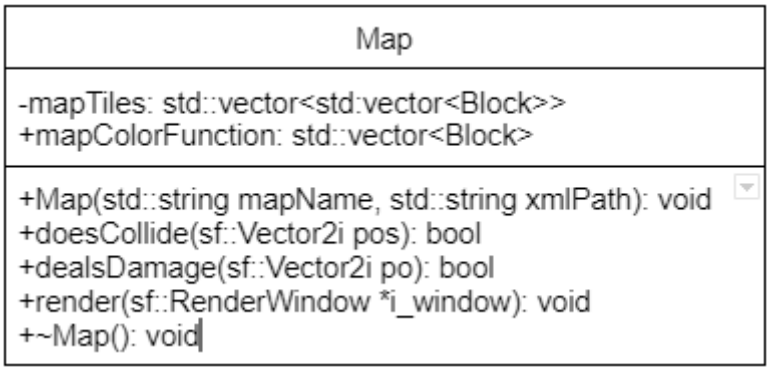
### 2.3.2 Criação do Mapa

Na recriação do jogo, foram desenvolvidas funções específicas para o mapa interativo, abordando desde a detecção de colisões até a identificação de áreas que causam dano ao jogador. Essas funções desempenham um papel crítico na jogabilidade, garantindo que o personagem do jogador interaja de forma coerente com o ambiente. Além disso, essas funcionalidades contribuem para a autenticidade da experiência, recriando assim os desafios e a dinâmica de jogabilidade do Mini Doom original.

#### 2.3.2.1. *Class map*

A classe ***"Map"*** , representada na Figura 03, neste projeto de recriação do Mini Doom tem as seguintes funcionalidades:

**FIGURA 03: UML *MAP***




Fonte: Autoral

**Construtor *Map(std::string mapName, std::string xmlPath):***

- Este construtor é responsável por inicializar um objeto da classe `Map`.
- Ele recebe dois parâmetros: o nome do mapa e o caminho para um arquivo XML que contém informações sobre o layout do mapa.
- A função lê o arquivo XML especificado para criar a representação do mapa no jogo, preenchendo a estrutura de dados `mapTiles` com informações sobre os blocos no mapa. Também pode realizar outras configurações iniciais necessárias para o mapa.

**Destrutor *~Map():***

- O destrutor é usado para liberar recursos ou realizar ações de limpeza quando um objeto da classe `Map` não é mais necessário.
- Neste código, o destrutor pode não realizar ações específicas, mas em projetos mais complexos, pode ser usado para liberar recursos alocados dinamicamente ou executar outras tarefas de limpeza.

**Função *void render(sf::RenderWindow *i_window)*:**

- Esta função é responsável por renderizar o mapa na janela gráfica.
- Ela recebe um ponteiro para a janela (`sf::RenderWindow`) onde o mapa deve ser renderizado.
- No geral, essa função realiza a renderização dos blocos do mapa na janela, criando a representação visual do ambiente de jogo.

**Função *bool doesCollide(sf::Vector2i pos)*:**

- Esta função verifica se há uma colisão em uma posição específica do mapa.
- Ela recebe um vetor 2D de posição ***(sf::Vector2i)*** como parâmetro.
- A função verifica se o bloco na posição especificada ***(pos)*** possui uma colisão (propriedade ***collision*** definida como verdadeira). Isso é útil para detectar se um jogador ou objeto colidiu com um obstáculo no mapa.

**Função *bool dealsDamage(sf::Vector2i pos)*:**

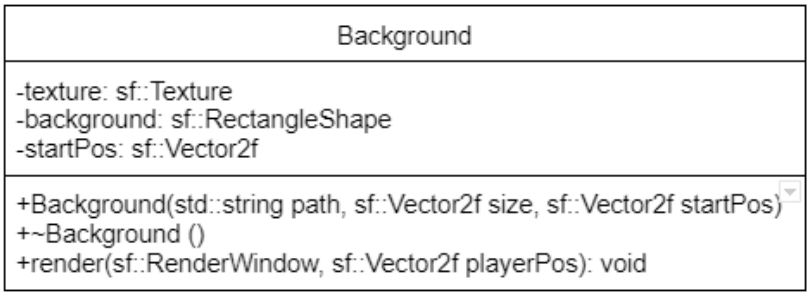
- Esta função verifica se uma posição específica do mapa causar dano.
- Ela recebe um vetor 2D de posição ***(sf::Vector2i)*** como parâmetro.
- A função verifica se o bloco na posição especificada ***(pos)*** causa dano (propriedade ***damage*** definida como verdadeira). Isso é útil para determinar se um jogador ou objeto sofre dano ao entrar em contato com uma área perigosa no mapa.

No geral, a *classe Map* é projetada para representar e gerenciar o mapa do jogo. Ela carrega as informações do mapa a partir de um arquivo XML, permite a renderização do mapa na janela gráfica e fornece funções para verificar colisões e danos no mapa. Essa classe desempenha um papel fundamental na jogabilidade e na interação do jogador com o ambiente do jogo em "Jão do Sertão".

#### 2.3.2.2. *Class Background*

Na recriação do "Mini Doom", a classe Background, conforme apresentada na Figura 04, desempenha um papel crucial na construção do ambiente visual do jogo. Ela possui as seguintes funcionalidades:

**FIGURA 04: UML DO *BACKGROUND***




Fonte: Autoral

**Construtor *Background(std::string path, sf::Vector2f size, sf::Vector2f startPos):***

- O construtor é responsável por criar uma instância da classe *Background* e configurar suas propriedades.
- Ele recebe três parâmetros: o caminho para a imagem de textura do fundo, o tamanho desse fundo e sua posição inicial.
- No contexto do jogo "Jão do Sertão", essa função permite a criação de diferentes cenários de fundo, cada um com uma textura específica, tamanho e posição inicial. Isso contribui para a ambientação e imersão do jogador no mundo do jogo.

**Destrutor *~Background():***

- O destrutor é uma função especial que pode ser usada para realizar tarefas de limpeza ou liberar recursos quando um objeto da classe *Background* não é mais necessário.
- No presente código, o destrutor pode não realizar ações específicas, mas em projetos mais complexos, ele pode ser útil para a gestão de recursos, como a liberação de memória de texturas ou outros recursos gráficos.

**Função *void render(sf::RenderWindow *i_window, sf::Vector2f playerPos):***

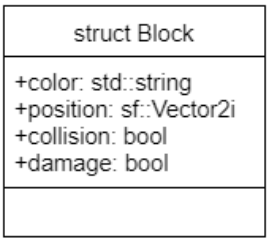
- Esta função desempenha um papel importante na renderização do cenário do jogo na janela gráfica.
- Ela recebe dois parâmetros: um ponteiro para a janela onde o cenário deve ser renderizado e a posição atual do jogador no jogo.
- Em "Jão do Sertão", a função ***render*** permite atualizar e exibir o cenário de fundo na janela do jogo, levando em consideração a posição do jogador. Isso cria a ilusão de movimento do cenário e contribui para a experiência visual do jogador no jogo.

No contexto do jogo "Jão do Sertão", a classe *Background* desempenha um papel fundamental na criação dos ambientes visuais que o jogador irá explorar. Ela permite a personalização dos cenários de fundo, sua renderização na janela do jogo e a sincronização com a posição do jogador para criar uma experiência imersiva e envolvente. E assim, o jogo é capaz de criar diferentes cenários e dar vida ao mundo do "Jão do Sertão".

#### 2.3.2.3. *Class Struct Block*

A estrutura Block, conforme representada na Figura 05, desempenha um papel fundamental na construção do ambiente do jogo. Ela é responsável por definir as características e comportamentos dos blocos que compõem o cenário do jogo, contribuindo significativamente para a jogabilidade e estética geral da experiência. Em vista disso, segue o detalhamento das funcionalidades e o papel desempenhado pela estrutura Block neste contexto.

**FIGURA 05 : UML *STRUCT BLOCK***




Fonte: Autoral

**Membro *color:***

- Este membro armazena uma string que representa a cor do bloco.

- No contexto do jogo "Mini Doom", a cor é usada para determinar a aparência visual do bloco, o que contribui para a estética geral do jogo.

**Membro *position:***

- Este membro é uma variável do tipo *sf::Vector2i* que armazena as coordenadas X e Y da posição do bloco no ambiente 2D do jogo.
- A posição do bloco é crucial para definir sua localização exata no mapa do jogo, permitindo que os jogadores interajam com ele ou evitem colisões.

**Membro *collision:***

- Este membro é uma variável booleana que indica se o bloco está sujeito a colisões no jogo.
- Quando *collision* é verdadeiro, significa que o bloco é um objeto sólido que pode ser colidido por outros elementos do jogo, como personagens ou projéteis.

**Membro *damage:***

- Este membro é uma variável booleana que indica se o bloco pode causar dano a outros elementos do jogo.
- No contexto de "Mini Doom", blocos com *damage* verdadeiro podem representar armadilhas ou elementos perigosos que causam dano ao jogador se ele interagir com eles.

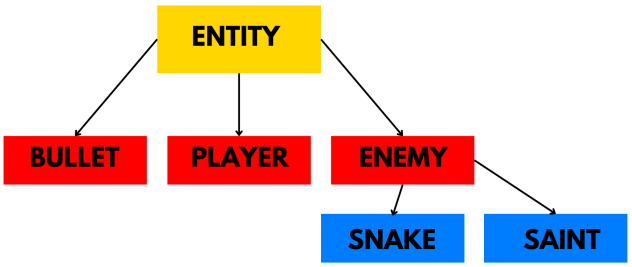
No geral, a estrutura Block é uma parte essencial da recriação do jogo "Mini Doom" e é usada para definir as características e comportamentos dos blocos que compõem o ambiente do jogo. Os blocos desempenham um papel importante na construção do mapa, na criação de obstáculos, na definição da estética do jogo e na interação do jogador com o ambiente. Cada membro da estrutura Block contribui para a personalização e funcionalidade desses elementos no jogo "Mini Doom".

### 2.3.3 Personagens

Na recriação do jogo, uma parte fundamental do desenvolvimento envolveu a criação dos personagens, que incluem o jogador (Player) e os inimigos (Enemy). Esses personagens são representados através de uma hierarquia de herança, como representado na Figura 06, na qual

a classe base Entity atua como o ancestral comum para as classes Player, Enemy e Bullet. Essa abordagem hierárquica permite a definição de comportamentos e atributos compartilhados, mantendo a estrutura organizada e eficiente.

**FIGURA 06 : Herança**

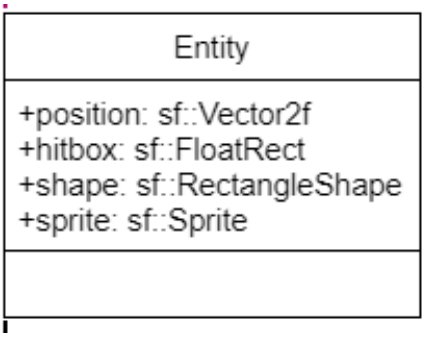



Fonte: Autoral

### 2.3.3.1. *Class Entity*

A classe `Entity`, Figura 07, foi utilizada na criação do jogo "Jão do Sertão" como parte da estrutura de representação dos personagens no jogo. Abaixo segue a análise dos propósitos gerais dessa classe e o significado de cada membro no contexto do projeto:

**FIGURA 07 : UML *ENTITY***




Fonte: Autoral

- ***position (sf::Vector2f):*** Este membro descreve a posição de um personagem no ambiente do jogo, usando coordenadas x e y em um espaço 2D. Ele permite controlar a localização precisa de cada personagem no mundo do jogo.

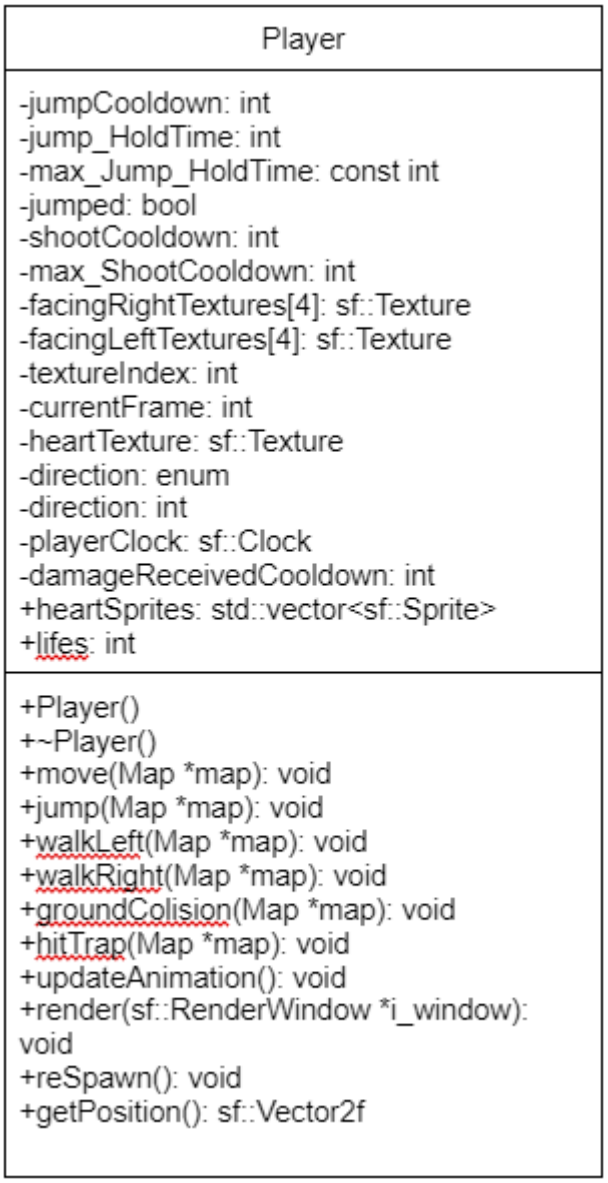
- ***hitbox (sf::FloatRect):*** Este membro define a área retangular que representa a zona de colisão de um personagem. No jogo, é essencial para detectar colisões entre personagens ou entre personagens e objetos, afetando interações e comportamentos no ambiente.
- ***shape (sf::RectangleShape):*** Este membro especifica a forma visual de um personagem, que geralmente é um retângulo. Ele influencia a aparência gráfica do personagem no jogo, permitindo sua renderização.
- ***sprite (sf::Sprite):*** Este membro representa a imagem ou sprite associado a um personagem. O *sf::Sprite* é usado para exibir a representação visual do personagem na tela do jogo, incluindo animações e texturas.

A classe *Entity* desempenha um papel fundamental na representação dos personagens que interagem no mundo do jogo. Cada personagem é definido por sua posição, área de colisão, forma gráfica e representação visual. Esses elementos permitem que o jogo posicione, colida, renderize e anime os personagens de acordo com as regras e mecânicas do jogo. A classe *Entity* serve como uma base para a criação de diversos tipos de personagens no jogo, como o próprio "Jão do Sertão" e inimigos. Cada personagem específico pode herdar dessa classe base e incorporar comportamentos e características adicionais conforme necessário para o jogo.

#### 2.3.3.2. *Class Player*

A classe *Player,* Figura 08, representa o jogador, ou seja, o personagem principal do jogo. Sendo assim, é de extrema importância descrever o propósito e a função de cada elemento do UML.

**FIGURA 08 : UML *PLAYER***




Fonte: Autoral

**Variáveis de membro privado:**

- ***jumpCooldown, jump_HoldTime, max_Jump_HoldTime, jumped, shootCooldown, max_ShootCooldown, facingRightTextures[4], facingLeftTextures[4], textureIndex, currentFrame, heartTexture, direction, playerClock, damageReceivedCooldown:*** Essas são variáveis de membro privado que armazenam informações relacionadas ao estado e comportamento do jogador. Elas controlam o tempo de espera entre pulos, o

tempo de segurar o botão de pulo, o tempo de espera entre disparos, as texturas para a animação do jogador, a direção em que o jogador está enfrentando, um relógio para controlar o tempo, e o tempo de resfriamento após receber dano, entre outras informações.

***heartSprites: std::vector<sf::Sprite>:***

- Esta variável de membro privado é um vetor de sprites que representam as vidas do jogador. Esses sprites são usados para mostrar as vidas restantes do jogador na interface do jogo.

***lifes: int:***

- Esta variável de membro privado mantém o número de vidas que o jogador possui.

**Construtor *Player():***

- O construtor é responsável por inicializar um objeto da classe Player.
- No contexto do jogo, ele inicializa as variáveis de membro e configura o jogador com valores.

**Destrutor *~Player():***

- O destrutor é usado para liberar recursos ou realizar tarefas de limpeza quando um objeto da classe Player não é mais necessário. No código fornecido, pode não conter ações específicas.

**Funções públicas:**

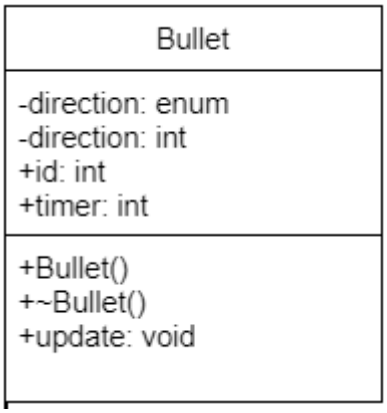
- As funções públicas desempenham várias ações no jogador e são essenciais para seu funcionamento no jogo.
- Elas incluem ***move, shoot, receiveDamage, jump, walkLeft, walkRight, groundColision, hitTrap, update, updateAnimation, render, reSpawn, e getPosition.*** Cada uma dessas funções executa uma ação específica, como mover o jogador, atirar, receber dano, realizar pulos, detecção de colisões, atualização da animação, renderização, respawn, e obter a posição atual do jogador.

No geral, a *classe Player* representa o personagem controlado pelo jogador no jogo "Jão do Sertão". Ela gerencia todas as interações e ações do jogador, como movimento, disparo, pulo, colisões, animação, e renderização. Além disso, mantém informações sobre a vida do jogador e o controle de várias temporizações para evitar ações abusivas. A classe é uma parte fundamental do jogo, proporcionando a experiência de jogo do jogador dentro do contexto do jogo.

#### 2.3.3.3. *Class Bullet*

Na recriação do "Mini Doom", a classe Bullet, conforme apresentada na Figura 09, desempenha um papel crucial na construção do jogo. Ela possui as seguintes funcionalidades:

**FIGURA 09 : UML *BULLET***




Fonte: Autoral

***enum direction:***

- Este é um tipo de enumeração que representa a direção da bala. Pode assumir dois valores: *left* (esquerda) ou *right* (direita).

***int direction:***

- Esta é uma variável membro que armazena a direção atual da bala como um valor inteiro (baseado no enum).

***int id:***

- Esta é uma variável membro que armazena um identificador único para a bala.

***int timer:***

- Esta é uma variável membro que controla o tempo de vida da bala.

***Bullet(sf::Vector2f startPosition, const int direction, int &id):***

- Este é o construtor da classe *Bullet.*
- Ele recebe três parâmetros: a posição inicial da bala, a direção (como um valor inteiro) e uma referência a um identificador único.
- O construtor inicializa os membros da classe, incluindo a posição, a forma da bala, a cor, o identificador, a direção e a caixa de colisão da bala.

**Destrutor *~Bullet():***

- Este é o destrutor da classe *Bullet*.
- O destrutor pode não conter ações específicas neste código, mas em projetos mais complexos, pode ser usado para liberar recursos ou fazer tarefas de limpeza.

***void update()***

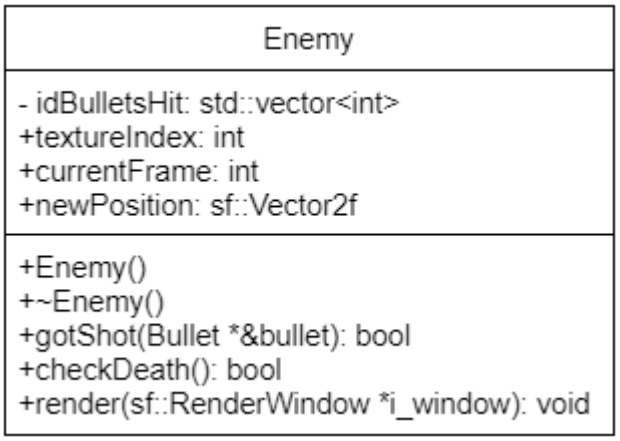
- Esta função é responsável por atualizar a posição e o estado da bala em cada quadro do jogo.
- Dentro dessa função, a posição da bala é ajustada com base na direção (esquerda ou direita), a posição da forma é atualizada, o temporizador é incrementado e a caixa de colisão é atualizada.

No geral, a classe *Bullet* representa uma bala em do jogo "Jão do Sertão". Ela armazena informações como posição, direção, identificador e temporizador. O construtor permite criar balas com configurações iniciais específicas, e a função *update* é usada para atualizar o movimento e o estado da bala no jogo. Essa classe é essencial para implementar a mecânica de tiros no jogo.

#### 2.3.3.4. *Class Enemy*

A classe ***"Enemy"*** , representada na Figura 10, neste projeto de recriação do Mini Doom tem as seguintes funcionalidades:

**FIGURA 10 : UML ENEMY**




Fonte: Autoral

**Construtor *Enemy():***

- Este é o construtor da classe Enemy.
- No contexto do jogo, esse construtor pode ser responsável por inicializar um inimigo genérico, mas o código apresentado não contém a lógica específica do construtor.

**Destrutor *~Enemy():***

- Este é o destrutor da classe Enemy.
- Da mesma forma que o construtor, ele pode ser usado para realizar tarefas de limpeza específicas relacionadas aos inimigos, mas o código atual não contém ações específicas no destrutor.

**Função *bool gotShot(Bullet *&bullet):***

- Essa função verifica se o inimigo foi atingido por uma bala (Bullet) específica.
- Ela recebe um ponteiro para um objeto Bullet.
- A função verifica se a bala colidiu com o inimigo e retorna verdadeiro se isso aconteceu. Isso é útil para determinar se um inimigo foi atingido por uma bala do jogador.

**Função bool *checkDeath():***

- Esta função verifica se o inimigo está morto.
- No contexto do jogo, é implementado uma lógica para determinar se a vida do inimigo chegou a zero ou se ele foi derrotado de alguma outra maneira. Retorna verdadeiro se o inimigo estiver morto.

**Função *void render(sf::RenderWindow *i_window):***

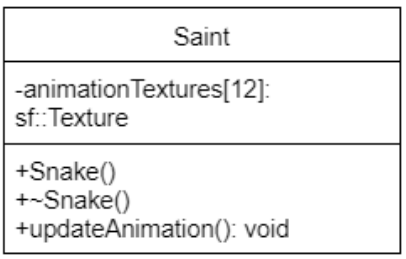
- Esta função é responsável por renderizar o inimigo na janela gráfica.
- Ela recebe um ponteiro para uma janela gráfica (sf::RenderWindow).
- No contexto do jogo, essa função desenha o inimigo na janela para que ele seja visível durante o jogo.

No geral, a classe Enemy serve como uma base genérica para representar inimigos no jogo "Jão do Sertão". Ela contém funcionalidades comuns a todos os inimigos, como a capacidade de ser atingido por balas, verificar a morte e ser renderizado na tela. Outras classes derivadas, como Snake e Saint, podem herdar de Enemy e adicionar funcionalidades específicas para diferentes tipos de inimigos no jogo, como animações, movimentos e comportamentos únicos. Portanto, a classe Enemy desempenha um papel fundamental na modelagem dos inimigos do jogo e na implementação de funcionalidades compartilhadas.

### 2.3.3.5. *Class Snake*

A classe *Snake*, Figura 11 define uma classe chamada `Snake`, que é uma classe derivada (subclasse) de *Enemy*. Ademais, este projeto de recriação tem as seguintes funcionalidades:

**FIGURA 11: UML *SNAKE***




Fonte: Autoral

**Construtor *Snake(int posx, int posy):***

- Este é o construtor da classe *Snake*.
- Ele recebe duas coordenadas (posx e posy) que representam a posição inicial da cobra.
- No geral, essa função é usada para criar instâncias da classe *Snake*, iniciando sua posição e outras propriedades específicas desse inimigo.

**Destrutor ~*Snake():***

- O destrutor é responsável por realizar tarefas de limpeza quando um objeto da classe `Snake` não é mais necessário.
- No código fornecido, não há ações específicas no destrutor, mas em projetos mais complexos, ele pode ser usado para liberar recursos ou realizar outras tarefas de limpeza.

**Função *void updateAnimation():***

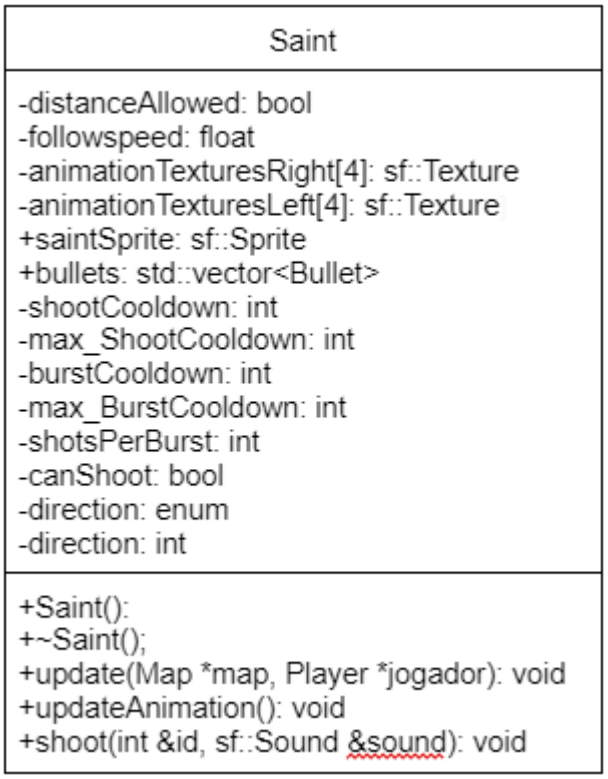
- Esta função é responsável por atualizar a animação da cobra.

No geral, a classe *Snake* representa um tipo específico de inimigo no jogo. Ela herda características e comportamentos da classe base *Enemy*. O construtor permite criar instâncias da cobra com uma posição inicial, e a função *updateAnimation* é utilizada para controlar a animação da cobra durante o jogo. O objetivo principal da classe é definir o comportamento e a representação de uma cobra estática no contexto do jogo, como parte da lógica do inimigo do jogo.

### 2.3.3.6. *Class Saint*

Na recriação do "Mini Doom", a classe Saint, conforme apresentada na Figura 12, desempenha um papel crucial na criação do ***Boss*** do Jogo:

**FIGURA 12: UML *SAINT***




Fonte: Autoral

**Variável *distanceAllowed:***

Esta variável booleana indica se o inimigo está autorizado a seguir o jogador com base na distância. Pode ser usada para controlar o comportamento do inimigo em relação à proximidade do jogador.

**Variável *followspeed:***

- Essa variável de ponto flutuante representa a velocidade com que o inimigo segue o jogador. Ela controla a taxa de movimento do inimigo em direção ao jogador.

**Vetor *animationTexturesRight[4] e animationTexturesLeft[4]:***

- Esses vetores de texturas representam as texturas utilizadas para a animação do inimigo quando ele se move para a direita e para a esquerda. Cada vetor contém 4 texturas para as diferentes etapas da animação.

**Variável *saintSprite:***

- [Esta variável do tipo *sf::Sprite* representa o sprite do inimigo, que é usado para renderizar o inimigo na tela.

**Vetor *bullets:***

- Este vetor de `Bullet` representa os projéteis disparados pelo inimigo. Os projéteis são armazenados neste vetor.

**Variáveis *shootCooldown, max_ShootCooldown, burstCooldown, max_BurstCooldown, shotsPerBurst e canShoot:***

- Essas variáveis controlam o comportamento de disparo do inimigo. `shootCooldown` controla o tempo de recarga entre tiros, *max_ShootCooldown* define o máximo de tempo de recarga permitido, *burstCooldown* controla o tempo de recarga entre rajadas de tiros, *max_BurstCooldown* define o máximo de tempo de recarga entre rajadas, *shotsPerBurst* especifica quantos tiros o inimigo dispara em cada rajada, e *canShoot* indica se o inimigo pode disparar neste momento.

**Enumeração *direction:***

- Esta enumeração define duas direções possíveis: `left` (esquerda) e `right` (direita). É usada para determinar a direção em que o inimigo está se movendo ou mirando.

**Funçãovoid update(Map *map, Player *jogador):**

- Esta função é responsável por atualizar o comportamento do inimigo com base no mapa e na posição do jogador. No geral, ela controla o movimento do inimigo em direção ao jogador e a lógica de ataque.

**Função *void updateAnimation():***

Esta função atualiza a animação do inimigo com base na sua direção de movimento (esquerda ou direita). Ela é responsável por alternar as texturas conforme o inimigo se move.

**Função *void shoot(int &id, sf::Sound &sound):***

- Esta função é responsável por disparar projéteis (balas) em direção ao jogador. Ela pode receber um identificador `id` para identificar o inimigo que está atirando e um som `sound` para reproduzir um efeito sonoro de tiro.

**Construtor *Saint():***

- Este é o construtor da classe `Saint` e é usado para inicializar os membros da classe quando um objeto `Saint` é criado.

**Destrutor *~Saint():***

- O destrutor da classe *Saint* é responsável por liberar recursos ou realizar tarefas de limpeza quando um objeto *Saint* é destruído.

No geral, a classe *Saint* representa  a cobra no jogo "Jão do Sertão". Ela controla o comportamento do inimigo, incluindo seu movimento em relação ao jogador, animação, disparo de projéteis e outras características relacionadas ao inimigo. Essa classe desempenha um papel fundamental na jogabilidade e na interação do jogador com os inimigos no jogo.

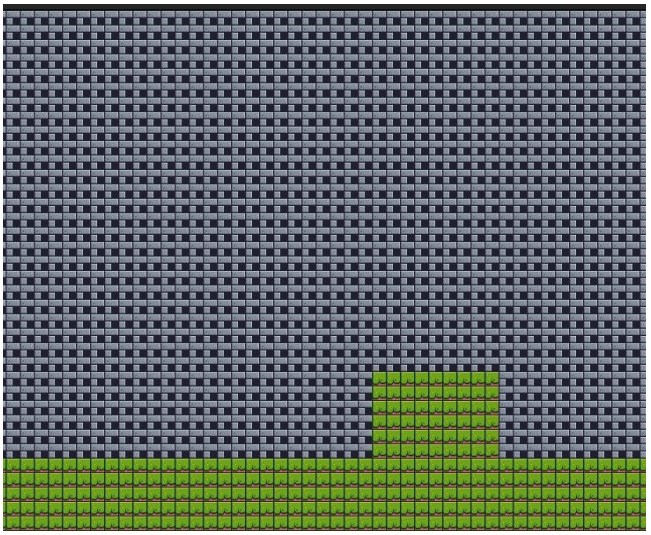
## 3. TESTE

Durante o processo de desenvolvimento do jogo, foram realizados testes abrangentes para garantir o bom funcionamento e a jogabilidade dos elementos principais, como mapa*, player* e inimigos.

### 3.1 Mapa

A fase inicial do desenvolvimento foi dedicada à criação do mapa do jogo, e essa escolha se deu por razões específicas. A técnica *TileMap*, necessária para a recriação do jogo, era um território desconhecido para o grupo de desenvolvedores do jogo. Portanto, o primeiro teste com essa técnica foi uma experiência desafiadora, uma vez que foi deparado com questões técnicas e desafios de implementação que não haviam sido enfrentados anteriormente pela equipe. Apesar de algumas tentativas iniciais não terem produzido resultados conforme o esperado, essa fase crítica de aprendizado foi essencial para a equipe. Como resultado desse processo, foi possível desenvolver uma estrutura de imagens predefinida que posteriormente se tornou uma base sólida para a construção do ambiente do jogo, como pode ser observado na figura 13.

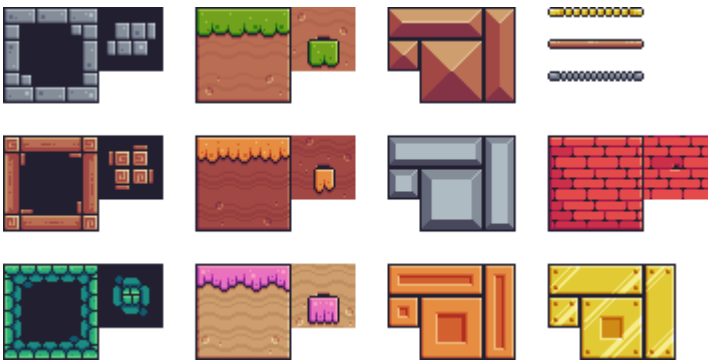
**FIGURA 13 : O Início dos Fins**



Fonte: Autoral

No estágio inicial dos testes, foi optado por utilizar um *tileset* que encontrou durante as primeiras pesquisas de *tilemap*, apresentando um formato de 16x16 pixels. Essas escolhas foram feitas com base na premissa de que 1º atenderia às dimensões apropriadas e seria suficiente para os testes iniciais, até que o jogo adquirisse uma aparência mais personalizada, incorporando seu próprio tema.

FIGURA 14: *Tileset* Inicial

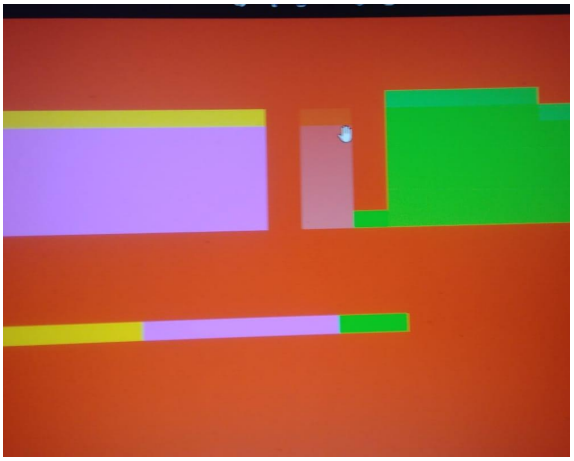


Fonte: itch.io

Uma vez compreendido o sistema de *TileMap* e superados os desafios iniciais, a equipe avançou para a próxima fase do desenvolvimento, na qual foi criado um mapa que incorporava a funcionalidade de detecção de colisão e armadilhas, conforme representado nas

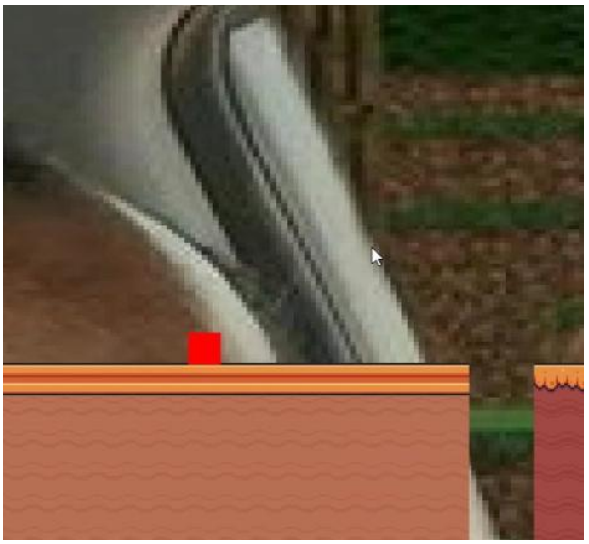
figuras 15 e Figura 16. A adição das colisões e armadilhas ao mapa era fundamental para garantir que o jogador pudesse interagir com o ambiente de maneira realista, permitindo evitar obstáculos e objetos e enriquecendo a jogabilidade do jogo, proporcionando uma experiência mais imersiva.

**FIGURA 15: 1º Imagem de Referência para a Criação do Mapa com Colisão & Armadilhas**



Fonte: Autora

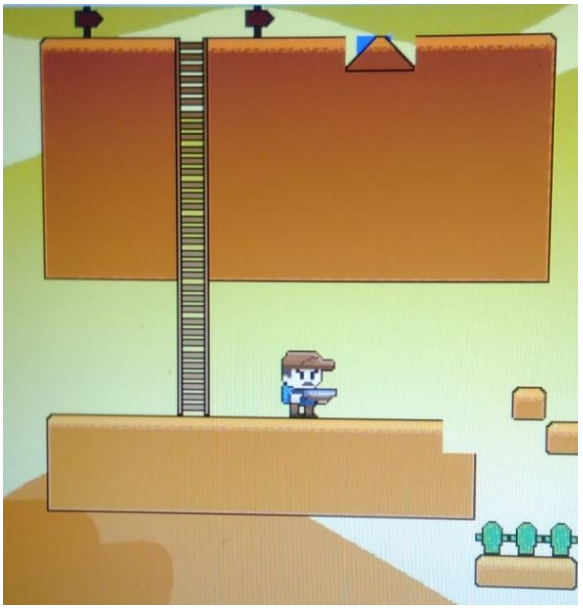
**FIGURA 16: 1º Mapa com Colisão & Armadilhas**



Fonte: Autora

Em etapa subsequente, a equipe concentrou seus esforços em dar vida e estilo ao jogo, cujo tema central era o "Sertão", o local de moradia do protagonista do jogo. Nesse processo, foram desenvolvidos elementos estéticos, incluindo cenários e elementos visuais que não apenas estabeleceram a atmosfera do jogo, mas também se alinharam de forma coesa com o tema do Sertão, como pode ser observado na Figura 17. Essa estética única contribuiu significativamente para a identidade visual do jogo, enriquecendo a experiência global do jogador.

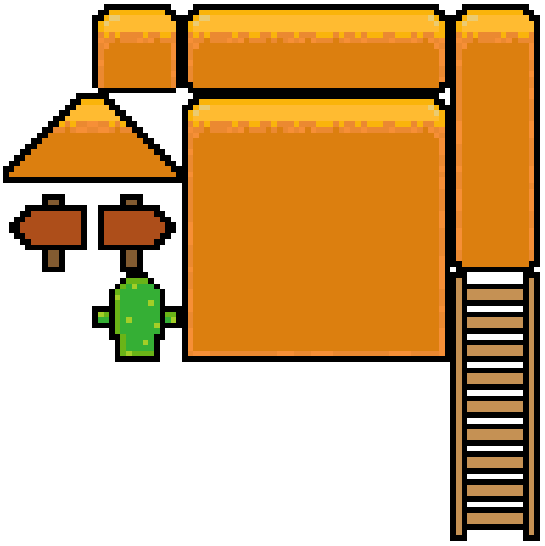
**FIGURA 17: Mapa Final**



Fonte: Autora

Para concretizar essa etapa, foi essencial realizar uma significativa modificação no *tileset*. O *tileset* em questão demandava uma reformulação completa a fim de se alinhar adequadamente à temática escolhida para o projeto, que foi definida como sendo o Sertão. Essa decisão não apenas exigiu uma repaginação visual, mas também a incorporação de elementos que pudessem capturar a essência e a atmosfera singular do Sertão, como pode ser reparado na Figura 18, o novo *tileset.*

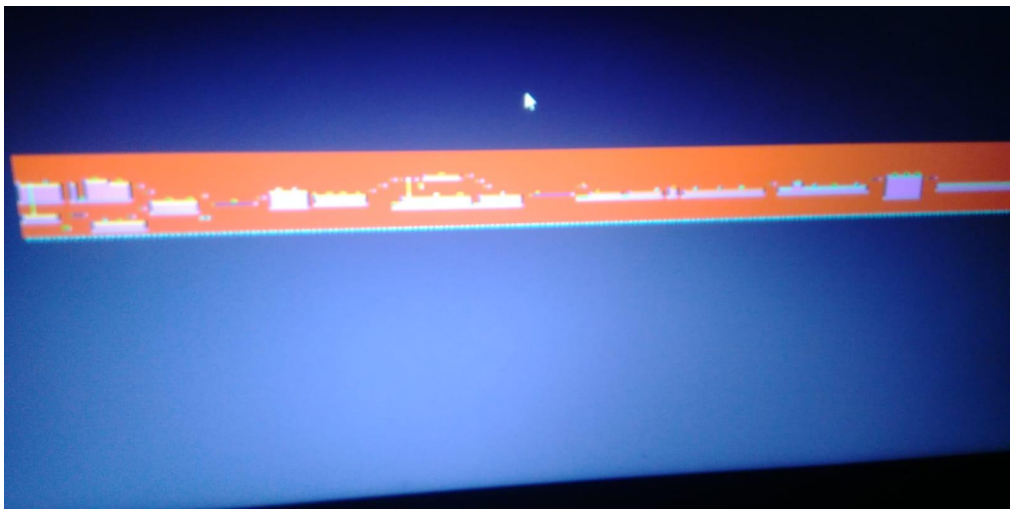
**FIGURA 18: *Tileset* Final**



**Fonte: itch.io**

Além disso, durante essa fase, houve a expansão do tamanho do mapa para acomodar os elementos adicionados, garantindo que o ambiente fosse coeso com a narrativa do jogo e oferecesse uma experiência mais rica e envolvente aos jogadores, como apresentado na Figura 18. Essa expansão contribuiu para uma exploração mais profunda do Sertão, tornando-o um cenário expansivo e desafiador para os jogadores explorarem.

**FIGURA 19: Imagem de Referência para a Criação do Mapa Final**



Fonte: Autora

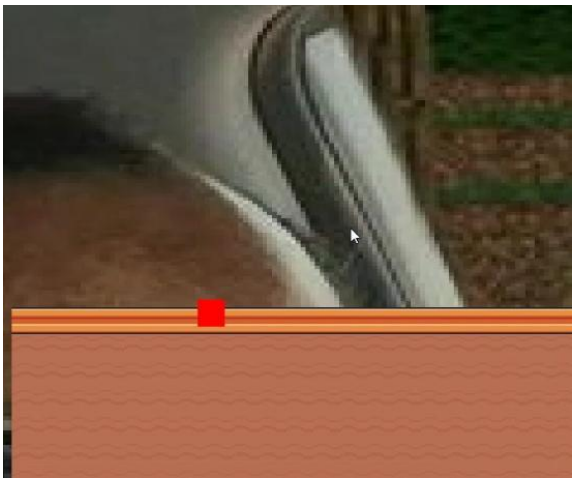
## 3.2 Personagens

No contexto dos personagens, foram conduzidos testes abrangentes de movimento, incluindo testes de pulo, com o objetivo de assegurar que os personagens se deslocassem de maneira fluida e em consonância com as ações dos jogadores. Além disso, ao final do processo, foi possível realizar os testes dos sprites nos personagens, garantindo que os modelos visuais dos personagens estivessem implementados corretamente e proporcionam uma experiência visual agradável aos jogadores. Esses testes desempenharam um papel fundamental na verificação da qualidade do jogo.

### 3.2. 1. Player

Inicialmente, na primeira aparição do jogador (o "Player"), este ainda não possuía um nome, sendo representado apenas por uma pequena figura vermelha, como pode ser observado na Figura 19. Ele foi criado simultaneamente com o mapa para verificar se a detecção de colisões estava funcionando corretamente.

**FIGURA 20: A Colisão**



Fonte: Autoral

No próximo passo, apresentado na Figura 20, foi abordado a implementação da mecânica de pulo para o personagem. A etapa de detecção de colisões bem-sucedida proporcionou uma base sólida para a introdução dessa funcionalidade. O jogador agora tinha a capacidade de

saltar, o que não apenas enriqueceu as possibilidades de movimento, mas também adicionou dinamismo ao jogo, permitindo a superação de obstáculos e desafios no ambiente.

**FIGURA 21 : Player, Edinaldo Pereira e o Poze do Rodo**

Fonte: Autoral

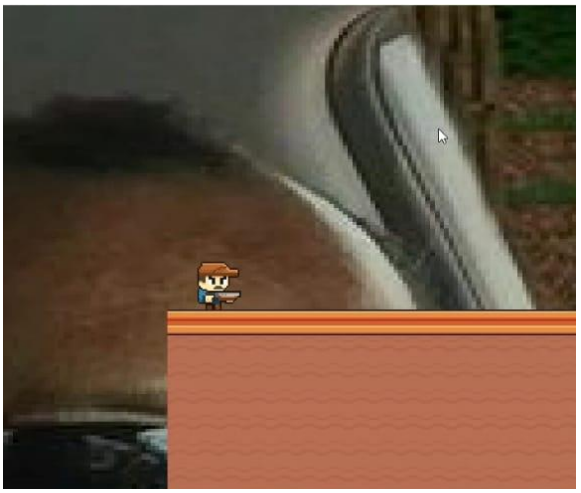
Após a implementação do salto, foi focado na adição da mecânica de tiros ao jogo, retratado na Figura 21. Esta edição trouxe uma nova camada de estratégia e ação, permitindo que o jogador interagisse com o ambiente de maneira mais complexa e desafiadora. Com tiros à disposição, o jogador podia enfrentar inimigos e superar obstáculos de uma forma totalmente nova, enriquecendo a experiência de jogo.

**FIGURA 22: Tiros**

Fonte: Autoral

Por fim, na Figura 22, a etapa final deste processo envolveu a adição do personagem. Com a mecânica de pulo e tiros já implementada e funcionando, foi possível focar na criação visual do jogador. Os sprites foram incorporados ao personagem, denominado como Jão, juntamente com a animação, dando-lhe uma identidade visual completa e tornando-o facilmente reconhecível. Esse toque final não apenas melhorou a estética do jogo, mas também contribuiu para a imersão dos jogadores, criando uma experiência mais envolvente e agradável.

**FIGURA 22: Jão,  Edinaldo Pereira e Poze do Rodo**



Fonte: Autoral

### 3.2.2. Inimigos

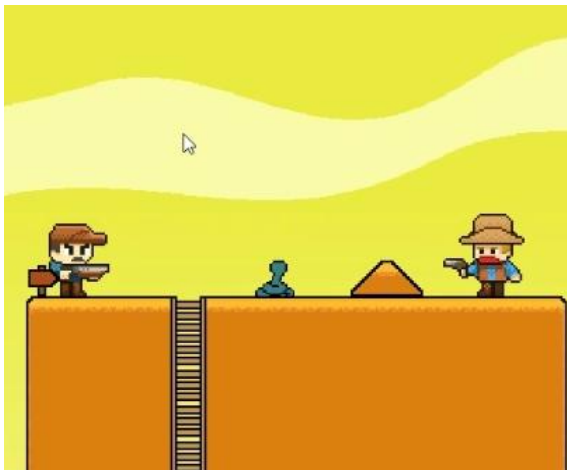
Na próxima fase de desenvolvimento e criação dos inimigos. Essa etapa levou algum tempo para ser concluída, uma vez que sua criação foi motivada pela necessidade de testar a mecânica de tiros. Inicialmente, foram criadas representações simples em forma de quadradinhos que podiam receber dano , apresentada na Figura 23, . Isso permitiu verificar o funcionamento da funcionalidade de tiro no jogo.

**FIGURA 23: Dano Inimigo**

Fonte: Autoral

Posteriormente, conforme a Figura 24, foram adicionados sprites e animações aos inimigos. Esse processo envolveu a criação das representações visuais das cobras e do Santo Cristo - ***boss*** do jogo - , elementos inimigos do jogo. Essas adições visuais contribuíram significativamente para a identificação e imersão dos jogadores no ambiente do jogo.

**FIGURA 24: Jão VS Santo Cristo & Cobrinha**



Fonte: Autoral

Na próxima etapa, representada na Figura 25, foi aprimorado as mecânicas de danos dos inimigos. Por exemplo, as cobras foram programadas para causar dano ao jogador quando se

aproximavam, enquanto o Santo Cristo atacava por meio de tiros à distância. Esses sistema de danos tornaram os inimigos desafiadores e interativos, enriquecendo a jogabilidade do jogo.

**FIGURA 25: Agora o Santo Cristo era bandido, destemido e temido no Distrito Federal**

Fonte: Autoral

Com o desenvolvimento progressivo do mapa, do jogador e dos inimigos, essas etapas iterativas desempenharam um papel crucial na construção sólida e funcional do jogo, como observado na Figura 25. A medida que foi avançando cada uma das fases, foi refletidos e otimizado continuamente as mecânicas e elementos visuais. Isso resultou em uma experiência de jogo completa e envolvente para os jogadores, onde a interação com o ambiente, a jogabilidade e a estética se combinaram de forma harmoniosa.

O mapa, por exemplo, foi meticulosamente projetado para oferecer desafios e explorar as habilidades do jogador, criando um ambiente imersivo. O jogador, por sua vez, passou por várias iterações de desenvolvimento para garantir movimentos suaves, respostas precisas e uma experiência controlada e envolvente. Os inimigos, apresentados de maneira gradual e iterativa, proporcionam desafios significativos aos jogadores, complementando a experiência de jogo e tornando-a mais dinâmica.

Em resumo, a abordagem gradual e cuidadosa adotada ao longo do processo de desenvolvimento, com foco no mapa, no jogador e nos inimigos, culminou em um jogo sólido

e funcional que oferece uma experiência completa e envolvente. Cada componente do jogo foi refinado e ajustado para garantir que todos os elementos funcionassem em conjunto de forma harmoniosa, proporcionando aos jogadores um ambiente desafiador e imersivo.

## 4. CONSIDERAÇÕES FINAIS

A recriação do jogo "Mini Doom", utilizando a linguagem de programação C + + em conjunto com a biblioteca SFML, representou um projeto desafiador e de grande sucesso. Durante a sua execução, foram enfrentados obstáculos relacionados à criação do mapa, implementação de animações e criação de diferentes inimigos.

### 4.1. Objetivos Concluídos

Ademais, os desenvolvedores investiram seus esforços na recriação minuciosa dos elementos primordiais que definiam o jogo original, incorporando uma abordagem estética completamente nova que conferiu ao jogo uma personalidade única e revitalizada, inspirada pela riqueza do Sertão. Esta abordagem foi fundamental para garantir que a essência e a nostalgia do "Mini Doom" original fossem preservadas, enquanto simultaneamente se introduziu uma atmosfera fresca, inspirada nas paisagens e na cultura sertaneja. Como resultado desses esforços, apresentam-se a seguir os objetivos que foram meticulosamente cumpridos:

#### 4.1.1 Objetivos do Projeto

- **Modelagem e Estruturação:** Foi realizada uma modelagem detalhada para identificar os elementos-chave necessários, como personagens, inimigos e lógica de jogo.
- **Hierarquia de Classes:** Uma hierarquia de classes sólida foi estabelecida para organizar o código-fonte de maneira lógica e permitir uma expansão eficiente.
- **Herança e Composição:** A herança e composição foram utilizadas para reutilização eficiente do código e flexibilidade no desenvolvimento de novos elementos do jogo.
- **Nomenclatura Significativa:** Foram atribuídos nomes significativos a variáveis, funções e classes para facilitar a compreensão do código.
- **Utilização da Biblioteca SFML:** A biblioteca SFML foi usada para lidar com gráficos, som e entrada de jogador, permitindo que a equipe se concentrasse na lógica do jogo.

### 4.1.2 Objetivos Específicos para a Recriação do "Mini Doom

- **Plataforma 2D:** O jogo foi mantido como uma plataforma 2D, mantendo a autenticidade do movimento e combate do "Mini Doom" original.
- **Animação de Personagens:** Animações detalhadas foram implementadas para dar vida aos personagens, tornando as ações mais fluidas e envolventes.
- **Diversidade de Inimigos:** Diferentes tipos de inimigos foram introduzidos, cada um com comportamentos únicos, aumentando a complexidade do jogo gradativamente.
- **Mecânicas de Objetos e Armadilhas:** Mecânicas envolventes para objetos interativos e armadilhas foram desenvolvidas, adicionando elementos estratégicos ao jogo.
- **Técnica: *TileMaps*:** A técnica de *TileMaps* foi usada para criar o mapa, permitindo a expansão do mundo do jogo e a personalização dos ambientes, bem como a implementação das mecânicas de objetos e armadilhas.

Essas realizações resultaram em uma recriação do "Mini Doom" que preservou a essência do jogo original, mas com uma nova estética e personalidade, proporcionando uma experiência refrescante para os jogadores. O projeto permanece aberto a melhorias contínuas e expansões de conteúdo para manter os jogadores envolvidos e entretidos. Esta recriação do "Mini Doom" marca o início de uma emocionante jornada no mundo dos jogos.

## 4.2. Melhorias Futuras

Visando futuros aprimoramentos, seria vantajoso realizar um estudo mais aprofundado em relação ao uso de Tiled Maps para a construção de mapas mais robustos e livres de erros. Isso possibilitaria um design de níveis mais refinado, com maior flexibilidade na criação de cenários complexos e interativos.

Além disso, aprimorar o design geral do jogo é crucial. Isso inclui melhorar a aparência visual, adicionar animações mais detalhadas e trabalhar na interface do usuário para proporcionar uma experiência mais imersiva e agradável aos jogadores.

No que diz respeito à estrutura do jogo, seria benéfico implementar recursos como telas de menu, opções de configuração (incluindo ajustes de volume de som) e uma maior diversidade de níveis e desafios para atender a diferentes estilos de jogadores. Isso tornaria o jogo mais acessível e cativante para um público mais amplo.

Em resumo, investir em aprimoramentos futuros, como o estudo aprofundado de Tiled Maps, melhorias no design e na estrutura do jogo, e a inclusão de recursos para personalização, contribuirá para a criação de uma experiência de jogo mais completa e envolvente que atenda às necessidades e preferências de diversos tipos de jogadores.

## REFERÊNCIAS


[1] DOMESTIKA. O que é tileset e tilemap no desenvolvimento de games. Domestika.org. Disponível em: https://www.domestika.org/pt/blog/6985-o-que-e-tileset-e-tilemap-no-desenvolvimento-de-games . Acesso em: 1 ago. 2023.

[2] SFML DEVELOPMENT TEAM. SFML Documentation 2.5.1. Nov 2018. Disponível em: <https://www.sfml-dev.org/documentation/2.5.1/ . Acesso constante.

[3] MELO, Diego. O que é XML: Guia para iniciantes. Tecnoblog: Mobilion Mídia, 2021. Disponível em: https://tecnoblog.net/responde/o-que-e-xml-guia-para-iniciantes . Acesso em: 11 ago. 2023.

[4] OLIVEIRA, Marcos. Como fazer parser de XML com TinyXML2 C + +: Uma ferramenta simples e funcional para ler os tilemaps dos seus games.. Terminal Root, 6 mar. 2022. Disponível em: <https://terminalroot.com.br/2022/03/como-fazer-parser-de-xml-com-tinyxml2-cpp.html> Acesso em: 11 ago. 2023.

[5] Microsoft. Ler dados XML de um arquivo. Disponível em: https://learn.microsoft.com/pt-br/troubleshoot/developer/visualstudio/cpp/language-compilers/read-xml-data-from-file . Acesso em: 11 ago. 2023

[6] C + + PROGRESSIVO. Comando switch case - Instrução C + +. C + + Progressivo. 10 out. 2019. Disponível em: https://www.cmmprogressivo.net/2019/10/Comando-switch-case-instrucao-Cpp.html . Acesso em: 22 de ago. de 2023.

[7] CHAT GPT.Classe Map com strings. Chat Gpt, 2023. Disponível em: https://chat.openai.com/share/bcabcead-71d1-4347-bc2f-34e90a06d3d5. Acesso em: 03 de set. de 2023.

[8] CHAT GPT.Classe IDs Únicos para Objetos. Chat Gpt, 2023. Disponível em: https://chat.openai.com/share/c35b1630-2e10-41af-a3d6-b0efe20d9b4c. . Acesso em: 03 de set. de 2023.

[9] BUENO, André Duarte. Apostila de Programação Orientada a Objeto em C + +. Versão 0.4. UFSC-LMPT-NPC, 22 de agosto de 2002. Disponível em: http://www.cesarkallas.net/arquivos/apostilas/programacao/c_c%2B%2B/ApostilaProgramacaoCppv045.pdf . Acesso constante

[10] Cplusplus.com. Disponível em: https://cplusplus.com/ . Acesso constante.

[11] Projecefet. Race. Disponível em: https://github.com/projecefet/Race . Acesso em: constante.