Universidade Federal do Rio Grande do Norte
Escola Agrícola de Jundiaí
Graduação em Análise e Desenvolvimento de Sistemas

Autoria Web: Jogo com Javascript e HTML Canvas

Gilneide Fernanda Felix Bezerra
Luan Victor Ferreira Verissimo

Macaíba-RN
Novembro, 2023

Sumário

1 Introdução

Os avanços na tecnologia web têm proporcionado oportunidades significativas para a criação de experiências educativas inovadoras e acessíveis. Neste contexto, o presente relatório apresenta o processo de desenvolvimento de um simples jogo interativo e educativo utilizando JavaScript e HTML. O objetivo principal do projeto é explorar as capacidades de tal tecnologia para a criação de jogos que facilitam o aprendizado, desta forma, criando uma experiência envolvente e acessível.

Será abrangido no relatório as principais etapas do desenvolvimento, desde a concepção de ideias até a implementação de tais. Discutiremos os desafios encontrados no caminho, como também as soluções e decisões tomadas durante o processo.

Além disso, serão destacadas as principais funcionalidades do projeto, como a lógica do jogo, para uma melhor familiarização da linguagem de código utilizada no projeto. Acreditamos que ao compartilhar não apenas os sucessos, mas também os obstáculos e suas soluções, podemos contribuir para o entendimento mais amplo do processo de desenvolvimento. Pretendemos não apenas apresentar um produto final, mas também proporcionar uma compreensão aprofundada de nossas escolhas.

2 Tecnologias utilizadas

O ambiente de desenvolvimento para este projeto foi fundamentado em tecnologias web, sendo elas HTML, CSS e JavaScript. O JavaScript é uma linguagem de programação de alto nível devido à sua capacidade de criar interatividade dinâmica em páginas web. Juntamente com HTML (Hypertext Markup Language), que fornece a estrutura básica da página, e CSS (Cascading Style Sheets) para o design e apresentação visual, essas tecnologias formam a espinha dorsal do desenvolvimento web moderno.

Tanto o JavaScript, quanto o HTML possuem uma acessibilidade quase que globas, sendo suportados por um enorme número de navegadores e, além disso, são executados no lado do cliente, possibilitando uma melhor experiência interativa diretamente com a interface do usuário.

Além disso, tais tecnologias oferecem uma experiência completamente acessível a uma ampla gama de usuários, fazendo com que independente do dispositivo ou sistema operacional, sejam capazes de aproveitar do projeto e seus ensinamentos. A compatibilidade nativa dessas tecnologias com os padrões da web contribui para a disseminação efetiva do jogo, alinhando-se com os princípios de acessibilidade e usabilidade.

2.1 Aplicação das tecnologias

JavaScript foi implementado para gerenciar a lógica do jogo, desde a manipulação de eventos, como por exemplo o sistema de pontuação ou os eventos de teclas durante o jogo. O HTML canvas foi empregado para criar uma área de renderização dinâmica, permitindo a exibição de elementos gráficos em resposta às interações do jogador, sendo extremamente necessário para a manipulação de sprites durante o jogo, ou o uso de animações no mesmo.

3 Concepção e Design

A ideia inicial para o design do jogo foi inspirada por uma estética medieval, proporcionando um cenário envolvente, que combina elementos arcaicos e elementos naturais - apelando para a familiarização que esse tipo de elemento apresenta em produtos de entretenimento desenvolvidos para o público alvo.

Essa ambientação se estende por quatro cenários distintos, cada um mesclando a aparência medieval com toques naturais, como evidenciado no segundo e terceiro cenário, ambientados em um vasto templo dominado pela natureza, mantendo a grandiosidade e design arcaico. Os cenários variam desde ambientes sombrios, como o cemitério no primeiro cenário, onde lápides e uma igreja ao fundo definem a atmosfera, até paisagens mais claras, reforçando a riqueza visual do jogo. A estética medieval também se estende à personagem principal, que exibe detalhes de uma armadura nos braços e pernas, empunhando uma espada para enfrentar possíveis desafios.

Ao longo dos cenários, o jogador encontra itens coletáveis, como moedas e frutas, que servem como elementos de pontuação, enriquecendo a experiência e incentivando o jogador a explorar o ambiente. Esses elementos visuais contribuem com a diversão e a experiência educativa do usuário. Ao enfrentar desafios ao final de cada cenário, o jogo introduz uma caixa de diálogo na parte superior da tela, apresentando questões matemáticas. A resolução correta dessas perguntas não apenas permite a progressão do jogador para o próximo nível, mas também incrementa sua pontuação. Uma resposta incorreta não penaliza o jogador, mantendo uma abordagem positiva e encorajadora para a aprendizagem, reforçando o conceito de que errar faz parte do processo, sem desmotivar o jogador a continuar tentando. Ao atingir o último cenário do jogo, os jogadores são surpreendidos com uma caixa de diálogo que não apresenta a esperada pergunta matemática, mas sim uma mensagem de parabéns. Essa escolha visa provocar um sentimento genuíno de satisfação e felicidade no jogador, marcando o término da jornada de forma positiva.

Além dos conceitos educativos incorporados, o jogo busca narrar uma pequena e sig-

nificativa jornada da nossa carismática personagem. Com um prazo maior, planejávamos implementar também perigos em cenários sombrios, que contrastariam com as aventuras em ambientes mais abertos e receptivos, proporcionando uma experiência que vai além dos elementos educativos. Momentos desafiadores, como os pulos em plataformas e as perguntas matemáticas, representam obstáculos que contribuem para a construção da jornada da personagem.

De maneira intencional, o desfecho do jogo ocorre em uma pacata vila, criando uma atmosfera de tranquilidade. Nossa personagem, após enfrentar desafios e superar obstáculos, finalmente alcança seu objetivo. Este desfecho não apenas reforça a ideia de que a educação é uma jornada valiosa, mas também proporciona um contraste significativo, oferecendo ao jogador a sensação de que, após uma árdua jornada, é hora de descansar em um ambiente acolhedor.

4 Desenvolvimento do projeto

O progresso do projeto do jogo abrangeu diversas etapas, desde as concepções iniciais de conceitos e designs discutidos anteriormente, até a fase crucial de implementação do código. A transição dessas ideias para uma forma tangível começou com um modesto ponto de partida: implementar o código a um pequeno quadrado vermelho. Este quadrado não era apenas uma entidade visual, mas o embrião de uma série de funcionalidades que se desdobrariam ao longo do desenvolvimento do código.

A simplicidade do início foi estrategicamente escolhida, apresentando um quadrado que podia mover-se lateralmente e saltar. Este elemento básico permitiu uma compreensão fundamental das funções de pressionamento de teclas proporcionadas pelo JavaScript.

4.1 Sprites e animação da personagem

Após o modesto início com um simples quadrado em movimento, chegou o momento de elevar a experiência visual do jogo por meio da implementação de sprites. Ao estabelecermos o design para nossa personagem, empenhamo-nos em encontrar sprites disponíveis na internet que atendessem aos nossos critérios estéticos

A flexibilidade proporcionada pelo HTML canvas revelou-se essencial, permitindo-nos uma liberdade na criação de elementos gráficos, desde os mais simples até os mais elaborados. Foi através dessa ferramenta que transformamos nosso quadrado vermelho em uma encantadora protagonista, dotada de sprites animados para cada uma de suas ações, como caminhar lateralmente e saltar pelos cenários.

4.2 Implementação dos Backgrounds

Com uma personagem já animada, percebemos a necessidade de mais do que uma tela em branco para dar vida aos movimentos dela, precisávamos da presença de backgrounds

envolventes. Novamente, o HTML canvas assumiu o papel crucial de desenhar o cenário, utilizando uma imagem local. Além disso, o canvas era responsável pelo redimensionamento da imagem, garantindo que ela se integrasse perfeitamente ao nosso cenário.

Desde o início do projeto, nosso objetivo era a implementação de vários cenários, o que significava a necessidade de carregar diferentes imagens em momentos distintos do jogo. Foi nesse ponto que nos deparamos com um desafio significativo: a organização do código. A partir desse desafio, começamos a implementar um esquema de classes e orientação a objetos, utilizando múltiplos arquivos JavaScript dentro do mesmo projeto. Desta forma poderíamos trabalhar separadamente cada parte do código, utilizamos classes como Player, Background, NPC, etc.

Para facilitar a questão de troca de cenários, criamos um Array de backgrounds. Dessa forma os cenários serão trocados ao incrementarmos a variável IndexBG, que será usada como índice do Array.

```
const bg = [ // Vetor de backgrounds para facilitar a troca de cenário
    new Background('imgs/Backgrounds/Ruinas1.png'),
    new Background('imgs/Backgrounds/Temple1Complete.png'),
    new Background('imgs/Backgrounds/Temple2Complete.png'),
    new Background('imgs/Backgrounds/cenario1.png')
];
```

Figura 1: Array de Backgrounds presente no script.js.

Na função changeBackground, é verificado se estamos ou não no último cenário do Array. No caso de não estarmos, a várivel indexBG é incrementada.

```
function changeBackground() { // Função para trocar o background
    if (indexBG < bg.length-1) {
        indexBG++;
        dialog = false;
        indexQuestion++;
    }

    currentPlayer = player[indexPlayer];

    if(indexPlayer < bg.length-1){
        indexPlayer++;
    }
}
```

Figura 2: Função presente no script.js.

4.2.1 Colisões dos cenários

Chegamos a possivelmente mais desafiadora etapa do projeto: a implementação das colisões no cenário, onde nossa personagem poderia se mover, interagir, pular e mais. O primeiro passo para a criação das colisões envolveu a obtenção/criação manual de uma matriz do cenário, indicando cada bloco de colisão que ele possuiria. Atualmente, existem tecnologias que podem gerar essas matrizes de maneira semi-automática, simplificando esse processo. Utilizamos o Tiled. No entanto, a parte mais desafiadora ainda estava por vir: desenvolver toda a lógica das colisões e transformar essa lógica em linhas de código.

Para desenhar propriamente os blocos, criamos a classe CollisionBlock, presente no arquivo collisionBlock.js, que também contém a função que analisa se as colisões estão ocorrendo.

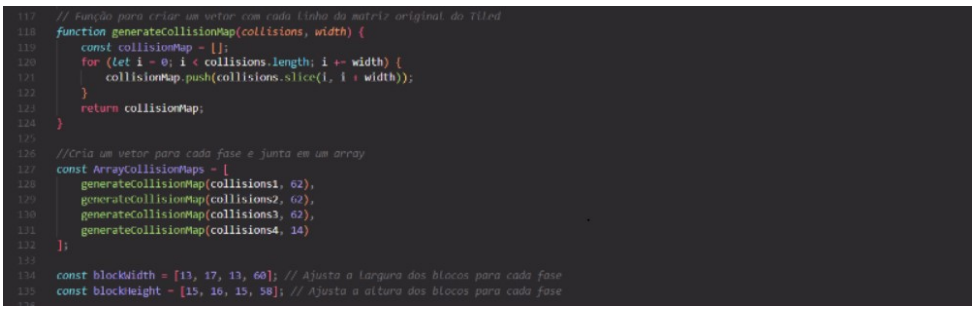
```
class CollisionBlock {
    constructor(options) {
        this.position = options.position || { x: 0, y: 0 };
        this.width = 62;
        this.height = 34;
    }

    draw() {
        ctx.fillStyle = 'rgba(255,0,0,0)'
        ctx.fillRect(this.position.x, this.position.y, this.width, this.height)
    }
    update(){
        this.draw();
    }
}

function collision(object1, object2){
    return (object1.position.y + object1.height >= object2.position.y &&
            object1.position.y <= object2.position.y + object2.height &&
            object1.position.x <= object2.position.x + object2.width &&
            object1.position.x + object1.width >= object2.position.x
    )
};
```

Figura 3: Arquivo collisionBlock.js

1. **Class CollisionBlock:** Essa classe representa um bloco de colisão retangular. Cada instância dessa classe é criada com uma posição (um objeto com as propriedades x e y) e dimensões padrão (largura = 62, altura = 34). A função draw é responsável por desenhar o bloco de colisão em um contexto de desenho (ctx). A função update simplesmente chama a função draw, indicando que o bloco deve ser redesenhado.

2. **collision(object1, object2):** Essa função cria um mapa de colisões a partir de um vetor collisions e uma largura especificada (width). Ela divide o vetor em linhas de acordo com a largura especificada, criando um array bidimensional que representa as colisões em um formato de matriz.

Dada a variedade de cenários, tornou-se imperativo que as colisões se adaptassem e utilizassem diferentes matrizes conforme o jogador avançasse pelos diferentes ambientes



```
// Função para criar um vetor com cada linha da matriz original do Tiled
function generateCollisionMap(collisions, width) {
    const collisionMap = [];
    for (let i = 0; i < collisions.length; i += width) {
        collisionMap.push(collisions.slice(i, i + width));
    }
    return collisionMap;
}

//Cria um vetor para cada fase e junta em um array
const ArrayCollisionMaps = [
    generateCollisionMap(collisions1, 62),
    generateCollisionMap(collisions2, 62),
    generateCollisionMap(collisions3, 62),
    generateCollisionMap(collisions4, 14)
];

const blockWidth = [13, 17, 13, 60]; // Ajusta a largura dos blocos para cada fase
const blockHeight = [15, 16, 15, 58]; // Ajusta a altura dos blocos para cada fase
```

Figura 4: Funções do arquivo collision.js

do jogo. Criamos então o arquivo collisions.js para guardar as matrizes citadas e as funções para implementá-las, como mostrado na figura a seguir.

1. **generateCollisionMap(collisions, width)**: Essa função cria um mapa de colisões a partir de um vetor collisions e uma largura especificada (width). Ela divide o vetor em linhas de acordo com a largura especificada, criando um array bidimensional que representa as colisões em um formato de matriz.

2. **ArrayCollisionMaps:** Esse trecho de código cria um array que contém mapas de colisões para diferentes fases. Cada fase tem seu próprio vetor de colisões (collisions1, collisions2, etc.), e esses vetores são transformados em mapas de colisões usando a função generateCollisionMap.

```
//Localiza onde devem ser desenhados os blocos de colisões de cada fase
function generateCollisionBlocks(collisionMap, blockWidth, blockHeight) {
    const collisionBlocks = [];
    collisionMap.forEach((row, y) => {
        row.forEach((symbol, x) => {
            if (symbol === 2109 || symbol === 127) {
                collisionBlocks.push(new CollisionBlock({
                    position: { x: x * blockWidth, y: y * blockHeight }
                }));
            }
        });
    });
    return collisionBlocks;
}

//Gera collisionBlocks de acordo com a fase
const ArrayCollisionBlocks = ArrayCollisionMaps.map((collisionMap, index) => {
    return generateCollisionBlocks(collisionMap, blockWidth[index], blockHeight[index]);
});
```

Figura 5: Funções do arquivo collision.js

1. **generateCollisionBlocks:** Essa função cria instâncias da classe CollisionBlock com base no mapa de colisões e nas dimensões dos blocos especificadas pelos arrays blockWidth e blockHeight. Os blocos são criados apenas nas posições onde o valor do símbolo no mapa de colisões é igual a 2109 ou 127.

2. **ArrayCollisionBlocks:** Esse trecho de código gera um array que contém arrays de instâncias da classe CollisionBlock para cada fase. Cada array de blocos é gerado com base no mapa de colisões correspondente, usando a função generateCollisionBlocks.

Após enfrentar desafios consideráveis nessa fase do desenvolvimento, conseguimos compreender melhor o que estávamos fazendo, permitindo-nos criar e implementar funções para lidar com diferentes tipos de colisões em cenários diversos do jogo.

Na classe Player, criamos os métodos mostrados nas imagens a seguir para verificar as colisões. Esses métodos são chamados no updatePlayer.

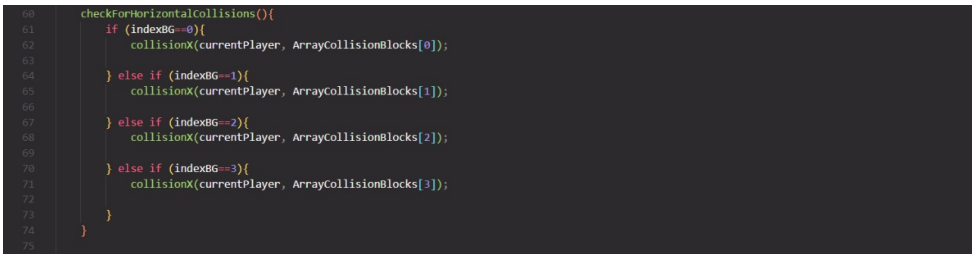


Figura 6: Métodos presentes no arquivo player.js

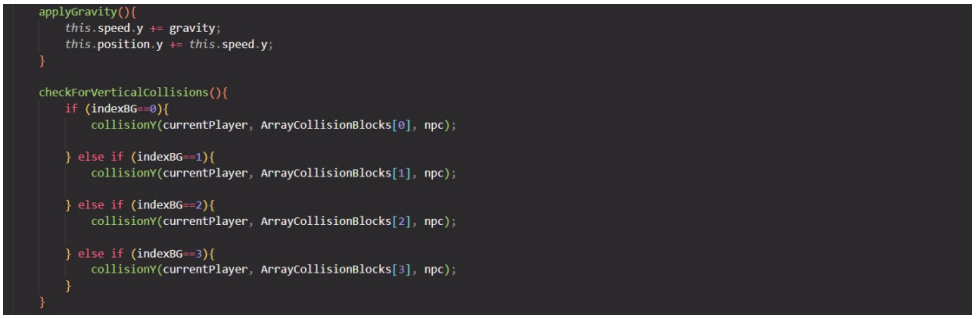


Figura 7: Métodos presentes no arquivo player.js

As funções chamadas, collisionX e collision Y, apresentam a mesma lógica, se diferenciando apenas na estrutura para cada eixo.

1. O loop for itera sobre cada bloco de colisão no array ArrayCollisionBlocks.
2. A função collision(currentPlayer, collisionBlock) verifica se há colisão entre o currentPlayer e o bloco de colisão atual. Se houver colisão, o código dentro do bloco if é executado. Dependendo da direção do movimento (currentPlayer.speed.x ou currentPlayer.speed.y), o código ajusta a posição e a velocidade do currentPlayer para evitar a sobreposição com o bloco de colisão.

4.3 Parte educativa

A lógica estabelecida era simples: quando o jogador alcançasse uma posição específica no canvas, caixas de diálogo com perguntas matemáticas seriam ativadas. Foi relativa-

mente fácil implementar a interação do usuário, permitindo que respondesse às perguntas por meio do teclado, desbloqueando assim novos cenários para exploração.

Utilizando, mais uma vez, o HTML canvas para criar as caixas de diálogo, o resultado foi um jogo que não apenas entreteve, mas também reforçou conhecimentos matemáticos de maneira recreativa. É válido ressaltar que, a fim de causar a sensação de aumento de nível de aprendizado ao jogador, a cada cenário concluído, as perguntas dificultavam.

A função showPrompt é responsável por fazer o prompt de resposta aparecer. A resposta que o jogador digitar será guardada na variável answer. Atribuir false à variável flag faz com que o prompt não apareça mais.

```
function showPrompt(){
    if(incorrect){
        answer = prompt('Resposta Incorreta! Tente novamente: ');
    } else {
        answer = prompt('Digite sua resposta: ');
        flag = false;
    }
}
```

Figura 8: Lógica presente no arquivo script.js

Em cada cenário, o dialogo mostrará uma pergunta diferente, por isso a criação de um vetor de perguntas. Juntamente com um vetor de respostas, para verificar se o jogador acertou a questão. A função showDialog é responsável por fazer a caixa de diálogo aparecer a tela.

4.4 NPC, coletáveis e assets

Era o momento de infundir mais vida e elementos ao nosso jogo, e uma ideia simples para atingir isso foi a implementação de NPCs, que surgiriam pelo mapa (um exemplo é visível no primeiro cenário). Além disso, decidimos adicionar a presença de itens coletáveis, escolhendo frutas e moedas douradas como elementos que pontuariam o mapa. Esses itens não apenas proporcionavam um incentivo para os jogadores explorarem novamente os mapas, mas também desempenhavam um papel fundamental no aumento exponencial de suas pontuações.

Para introduzi-los no jogo, criamos classes para cada um dos assets (decidimos chamá-los assim por serem elementos que enriquecem o ambiente visual do jogo, mas não têm um impacto significativo na jogabilidade). Utilizamos orientação a objetos. Para anima-

los adequadamente, no método update de cada classe, criamos os atributos currentFrame (frame atual), maxFrames (máximo de frames) e framesDrawn (frames já "desenhados").

1. this.currentFrame = this.currentFrame % this.maxFrames: Garante que currentFrame esteja dentro dos limites permitidos pelo número máximo de frames (maxFrames). Isso é feito utilizando a operação de módulo (%). Se currentFrame ultrapassar maxFrames, ele voltará ao início.

2. this.framesDrawn++: Incrementa o contador framesDawn. Este contador controla a taxa de atualização dos frames.

3. if (this.framesDrawn >= x): Verifica se o número de frames desde a última atualização ultrapassou um determinado limite. Isso é usado para controlar a velocidade de animação.

4. Dentro do bloco if:

   this.currentFrame++;: Incrementa o índice do frame atual.

   this.framesDrawn = 0;: Reinicia o contador framesDawn para começar a contar novamente.

A introdução de moedas e frutas exigiu uma lógica elaborada para lidar com a colisão e detecção desses objetos. Após o jogador entrar em contato com esses itens, ele deveriam sumir do mapa, e sua pontuação seria incrementada ao longo do jogo.

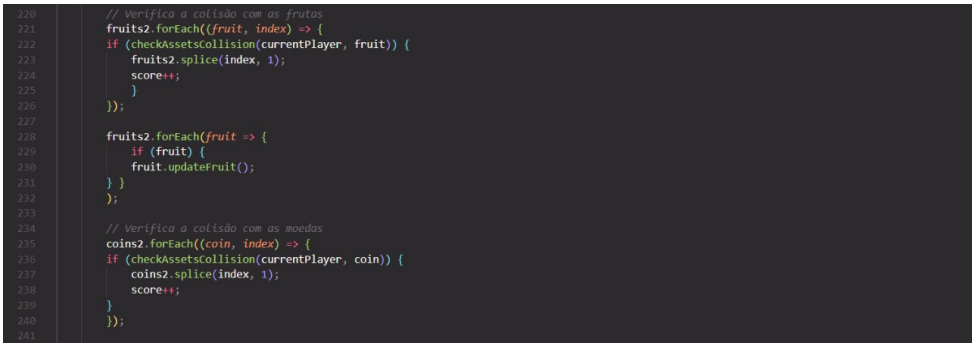
Para isso, criamos a função checkAssetsCollision. Esta função verifica se há colisão entre o jogador (player) e o objeto. Retorna true se houver colisão e false caso contrário. A verificação é feita comparando as coordenadas do retângulo do jogador com as coordenadas do retângulo do objeto em questão.

Decidimos por chamar essa função dentro do updateGameArea, já que é necessário que as verificações sejam feitas a cada atualização. Um exemplo pode ser visto na imagem a seguir.

Iremos exemplificar como funciona para os arrays presentes na imagem mostrada, mas a lógica pode ser utilizada para todos os assets presentes no código.

1. O código itera sobre todas as frutas no array fruits2 usando forEach.

2. Para cada fruta, verifica-se se há colisão com o jogador usando a função checkAssetsCollision.



```
// Verifica a colisão com as frutas
fruits2.forEach((fruit, index) => {
if (checkAssetsCollision(currentPlayer, fruit)) {
    fruits2.splice(index, 1);
    score++;
    }
});

fruits2.forEach(fruit => {
    if (fruit) {
    fruit.updateFruit();
} }
);

// Verifica a colisão com as moedas
coins2.forEach((coin, index) => {
if (checkAssetsCollision(currentPlayer, coin)) {
    coins2.splice(index, 1);
    score++;
}
});
```

Figura 9: Lógica presente no arquivo script.js

3. Se houver colisão, a fruta é removida do array (fruits2.splice(index, 1)) e o escore é incrementado.

4. Em seguida, itera-se novamente sobre as frutas remanescentes no array fruits2 para atualizar o estado de cada fruta chamando o método updateFruit para cada uma.

Na classe background, utilizamos o

```
if (fruits2.length > 0) { \\
    fruits2.forEach(fruit => {
        fruit.updateFruit();
    })
```

para garantir que o código só tente desenhá-las se ainda não tiverem sido coletadas.

5 Conclusão e considerações finais

O desenvolvimento deste jogo representou uma jornada desafiadora e recompensadora. Ao enfrentarmos obstáculos, como a implementação de colisões e a criação de um sistema dinâmico para diferentes cenários, desenvolvemos não apenas um jogo funcional, mas uma experiência que busca equilibrar diversão e aprendizado.

O HTML canvas foi uma ferramenta essencial, permitindo-nos criar animações envolventes, diálogos interativos e uma variedade de cenários visuais. É fundamental destacar que a concepção e desenvolvimento deste jogo não emergiram apenas de nossas ideias, mas foram forjados ao longo de extensas horas de estudo dedicado. A criação deste projeto exigiu uma imersão profunda em recursos online, incluindo sites e fóruns especializados em programação. Além disso, a consulta a vídeos tutoriais, a análise de projetos de inteligências artificiais foram recursos valiosos que enriqueceram significativamente o processo de aprendizado.

Em última análise, esperamos que este jogo ofereça aos jogadores não apenas entretenimento, mas também a oportunidade de explorar conceitos matemáticos de maneira criativa.