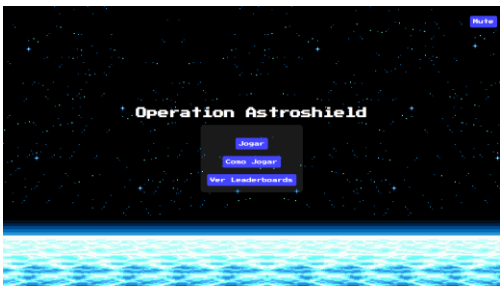
POLITÉCNICO DE SANTARÉM
ESCOLA SUPERIOR DE GESTÃO E TECNOLOGIA

Escola Superior de Gestão e Tecnologia

Licenciatura em Informática

Ano letivo de 2023/2024



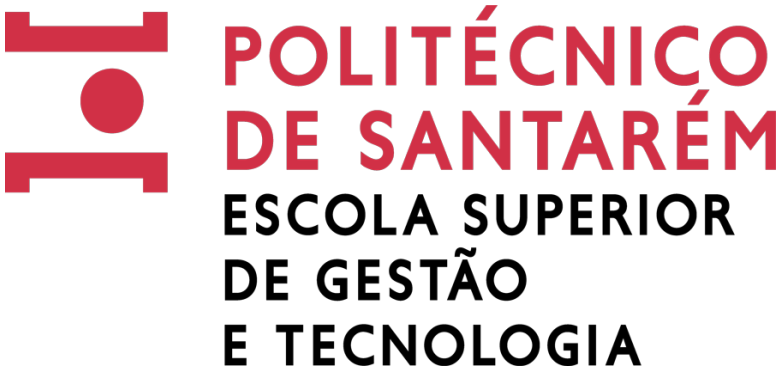
Documento - Operation Astroshield

João Nogueira, 230000299

Francisco Reis, 230001290

Antônio Elói, 230000932

ESGTS, 31 de maio de 2024



Escola Superior de Gestão e Tecnologia

Licenciatura em Informática

Ano letivo de 2023/2024

Documento - Operation Astroshield

Relatório do projeto Operation Astroshield, submetido à Escola Superior de Gestão e Tecnologia para a disciplina Sistemas Web.

Santarém, 31 de maio de 2024

Resumo

Operation Astroshield é um projeto de desenvolvimento de um jogo inspirado no clássico "Asteroids". Este jogo é implementado utilizando três tecnologias principais: HTML para a interface do utilizador, JavaScript para a lógica do jogo e Python Flask para a gestão dos dados.

A interface do jogo é construída em HTML, proporcionando uma estrutura visual onde o jogo é exibido. O layout inclui a tela de jogo principal, botões de controlo e áreas para pontuação e informações do jogador. Através desta interface, os jogadores podem interagir facilmente com o jogo, tornando a experiência mais intuitiva e envolvente.

O JavaScript é utilizado para implementar a lógica do jogo. Isto inclui a movimentação da nave espacial, a movimentação dos asteroides, a deteção de colisões e a resposta aos comandos do jogador. O canvas HTML5 é usado para renderizar os elementos gráficos do jogo, permitindo uma apresentação visual dinâmica e fluida. Os jogadores controlam uma nave espacial, que pode mover-se á volta do mundo e disparar projéteis para destruir os asteroides que surgem aleatoriamente e movem-se em direções variadas, aumentando o desafio conforme o jogo progride.

Python Flask serve como o backend do projeto, gerindo os dados do jogo. O Flask é responsável por armazenar as pontuações dos jogadores. Facilita a comunicação entre o frontend (HTML/JavaScript) e o servidor, permitindo a atualização e recuperação de dados em tempo real. Esta gestão eficiente dos dados garante que as pontuações são corretamente registadas e que os jogadores podem competir por posições no ranking, adicionando uma camada extra de competição ao jogo.

Funcionalidades Principais:

Movimentação da Nave:

O jogador controla uma nave espacial que pode mover-se em várias direções e disparar projéteis para destruir asteroides.

Geração de Asteroides:

Asteroides aparecem aleatoriamente na tela e movem-se em direções variadas, aumentando o desafio conforme o jogo progride.

Sistema de Pontuação:

Os jogadores ganham pontos ao destruir asteroides, e as suas pontuações são armazenadas na base de dados gerida pelo Flask.

Deteção de Colisões:

A lógica de colisão deteta quando a nave colide com um asteroide, resultando em perda de vida ou término do jogo.

Armazenamento e Recuperação de Dados:

Flask gere o armazenamento das pontuações e possivelmente o histórico de jogos, permitindo que os jogadores vejam as suas pontuações mais altas e histórico de desempenho.

Tecnologias Utilizadas:

HTML: Estrutura da interface do utilizador.

JavaScript: Implementação da lógica de jogo e interatividade.

Python Flask: Backend para gestão de dados, comunicação com o frontend e funcionalidades de armazenamento e recuperação de pontuações.

Operation Astroshield combina estes componentes para criar uma experiência de jogo envolvente, onde os jogadores podem testar as suas habilidades de pilotagem e competir por altas pontuações num ambiente interativo e dinâmico. A combinação de uma interface intuitiva, lógica de jogo complexa e gestão eficiente de dados garante que cada sessão de jogo seja única e desafiadora, proporcionando horas de entretenimento.

Índice

Lista de figuras

Capítulo 1 - Introdução

Operation Astroshield é um projeto inovador que visa recriar a magia do clássico jogo "Asteroids" com uma abordagem moderna. Desenvolvido com tecnologias web atuais, este jogo oferece uma experiência imersiva e interativa para os jogadores. Utilizando HTML para a interface do utilizador, JavaScript para a lógica do jogo e Python Flask para a gestão de dados. Este projeto não só desafia os jogadores a testar as suas habilidades de pilotagem, como também lhes permite competir por pontuações mais altas, criando um ambiente de jogo dinâmico e competitivo.

1.1 - Objetivos:

Ter um jogo funcional que recrie a experiência clássica de "Asteroids" com gráficos e mecânicas atualizados.

Implementar um sistema de dificuldade que se ajuste dinamicamente, proporcionando desafios crescentes à medida que o jogador avança.

Guardar as pontuações dos jogadores, permitindo a criação de rankings e incentivando a competição saudável entre os jogadores.

Com estes objetivos, Operation Astroshield não só reviverá um clássico amado, mas também trará novas funcionalidades e desafios que manterão os jogadores envolvidos e entretidos.

Capítulo 2 - Desenvolvimento

2.1 - Criação de classes em JavaScript

Primeiramente, foram criadas classes em JavaScript para cada objeto do trabalho. Essas classes são a classe nave, inimigos, balas, colisão, mundo, música, nave, ui e a main onde se irá importar cada classe e instanciar os objetos.

2.1.1 - Classe mundo

A classe Mundo é responsável por gerenciar a representação visual e o estado do mundo do jogo. Ela é responsável por desenhar o mundo na tela, atualizar sua posição e detetar colisões com outros objetos.

Atributos

- mundoImage: Uma imagem que representa o mundo do jogo.
- raio: O raio do mundo do jogo.
- posicaoX: A posição X do mundo do jogo na tela.
- posicaoY: A posição Y do mundo do jogo na tela.
- raioColisao: O raio da área de colisão do mundo do jogo.
- debug: Uma flag que indica se o modo de debug está ativado.
- tela: Uma referência para a tela do jogo.
- contexto: Uma referência para o contexto de desenho da tela.
- vidaInicial: A quantidade inicial de vidas do mundo do jogo.
- vida: A quantidade atual de vidas do mundo do jogo.
- colisao: Uma instância da classe Colisao que é responsável por detetar colisões entre o mundo e outros objetos.

Construtor

O construtor da classe Mundo é responsável por inicializar os atributos da classe. Ele define a imagem do mundo, o raio, a posição inicial e o raio de colisão. Ele também cria uma nova instância da classe Colisao e define a quantidade inicial de vidas do mundo.

Métodos

- ❖ atualizarPosicao(tela): Este método atualiza a posição do mundo do jogo para o centro da tela.
- ❖ desenhar(): Este método desenha o mundo do jogo na tela. Ele calcula as coordenadas X e Y para desenhar a imagem do mundo no centro da tela, define o tamanho desejado para a imagem e a desenha usando o contexto de desenho. Ele também desenha a área de colisão do mundo se o modo de debug estiver ativado.
- ❖ perderVida(): Este método diminui a quantidade de vidas do mundo do jogo em 1.
- ❖ detectarColisoes(): Este método deteta colisões entre o mundo do jogo e outros objetos. (Não implementado nesta classe)

Exportação

A classe Mundo é exportada como padrão, o que significa que pode ser importada na classe main usando a seguinte sintaxe:

```
import Mundo from './mundo.js';
```

Figura 1 - Importar classe Mundo

2.1.2 - Classe Nave

A classe Nave é responsável por gerenciar a nave espacial do jogador no jogo. Ela controla o movimento da nave, a rotação, o disparo de balas e a deteção de colisões.

Atributos

- ❖ angulo: O ângulo de rotação da nave em radianos.
- ❖ velocidadeAngular: A velocidade angular da nave em radianos por segundo.
- ❖ maximaVelocidadeAngular: A velocidade angular máxima da nave em radianos por segundo.
- ❖ aceleracao: A aceleração angular da nave em radianos por segundo ao quadrado.
- ❖ atrito: O coeficiente de atrito que afeta a velocidade angular da nave.
- ❖ aVirarEsquerda: Uma flag que indica se a nave está virando para a esquerda.
- ❖ aVirarDireita: Uma flag que indica se a nave está virando para a direita.
- ❖ tamanho: O tamanho da nave em pixels.
- ❖ raioOrbita: O raio da órbita em que a nave se move em pixels.
- ❖ posicaoX: A posição X da nave na tela em pixels.
- ❖ posicaoY: A posição Y da nave na tela em pixels.

- tela: Uma referência para a tela do jogo.
- debug: Uma flag que indica se o modo de debug está ativado.
- contexto: Uma referência para o contexto de desenho da tela.
- podeDisparar: Uma flag que indica se a nave pode disparar uma bala.
- balas: Um array que armazena as balas disparadas pela nave.
- balasNoPente: A quantidade de balas no pente da nave.
- maxBalasNoPente: A quantidade máxima de balas no pente da nave.
- tempoRegeneracao: O tempo em milissegundos que leva para regenerar uma bala.
- iniciarRegeneracaoBalas(): Um método que inicia a regeneração das balas.
- audioTiro: Uma referência para o objeto de áudio que toca o som do tiro.
- volume: O volume do som do tiro.
- naveImage: Uma imagem que representa a nave espacial.
- colisao: Uma instância da classe Colisao que é responsável por detectar colisões entre a nave e outros objetos.
- iniciarEventListeners(): Um método que inicia os eventos de teclado para controlar a nave.

Construtor

O construtor da classe Nave inicializa os atributos da classe, define a imagem da nave, cria uma nova instância da classe Colisao e inicia os eventos de teclado.

Métodos

- desenhar(): Este método desenha a nave na tela, incluindo a imagem da nave, a área de colisão e as balas.
- atualizar(): Este método atualiza a posição da nave, a velocidade angular, a posição das balas e verifica se a nave saiu da tela.
- iniciarEventListeners(): Este método adiciona eventos de teclado para controlar a rotação da nave e o disparo de balas.
- iniciarRegeneracaoBalas(): Este método inicia um intervalo que aumenta a quantidade de balas no pente da nave a cada tempoRegeneracao milissegundos.
- dispararBala(): Este método cria uma nova instância da classe Bala, adiciona-a ao array de balas, diminui a quantidade de balas no pente e toca o som do tiro.
- resetar(): Este método redefine a posição da nave para o centro da tela.

Exportação

A classe Nave é exportada como padrão, o que significa que pode ser importada na classe main usando a seguinte sintaxe:

```
import Nave from './nave.js';
```

Figura 2 - Importar classe Nave

2.1.3 - Classe Inimigos

A classe Inimigo é responsável por gerenciar os inimigos no jogo. Ela controla a criação, o movimento, a colisão e a dificuldade dos inimigos.

Atributos Estáticos

- vidaBase: A vida inicial de cada inimigo.
- velocidadeBase: A velocidade inicial de cada inimigo.
- Atributos da Instância
- tela: Uma referência para a tela do jogo.
- raioMundo: O raio do mundo do jogo.
- raio: O raio do inimigo em pixels.
- pixeis: O tamanho da imagem do inimigo em pixels.
- raioColisao: O raio da área de colisão do inimigo em pixels.
- velocidade: A velocidade atual do inimigo em pixels por segundo.
- posicaoX: A posição X do inimigo na tela em pixels.
- posicaoY: A posição Y do inimigo na tela em pixels.
- direcaoX: A direção horizontal do movimento do inimigo (1 para a direita, -1 para a esquerda).
- direcaoY: A direção vertical do movimento do inimigo (1 para cima, -1 para baixo).
- inimigosImage: Uma imagem que representa o inimigo.

- vida: A vida atual do inimigo.
- colisao: Uma instância da classe Colisao que é responsável por detectar colisões entre o inimigo e outros objetos.

Construtor

O construtor da classe Inimigo inicializa os atributos da instância, define a imagem do inimigo, cria uma nova instância da classe Colisao e chama o método iniciar() para definir a posição inicial e a direção do inimigo.

Métodos Estáticos

aumentarDificuldade(): Este método aumenta a vida base e a velocidade base dos inimigos.

resetarDificuldade(): Este método redefine a vida base e a velocidade base dos inimigos para os valores iniciais.

Métodos da Instância

- iniciar(): Este método define a posição inicial do inimigo fora da tela (à esquerda ou à direita) e a direção para mover em direção ao mundo.
- definirImagemAleatoria(): Este método define a imagem do inimigo aleatoriamente entre quatro opções.
- atualizar(): Este método atualiza a posição do inimigo de acordo com sua direção e velocidade, e atualiza a posição da colisão.
- desenhar(contexto, debug): Este método desenha a imagem do inimigo na tela, incluindo a área de colisão.
- evitarColisoes(outroInimigo): Este método evita que dois inimigos colidam um com o outro.
- resetar(): Este método redefine a posição do inimigo para fora da tela.

Exportação

A classe Inimigo é exportada como padrão, o que significa que pode ser importada na classe main usando a seguinte sintaxe:

```
import Inimigo from './inimigos.js';
```

Figura 3 - Importar a classe Inimigos

2.1.4 - Classe Bala

A classe Bala é responsável por gerenciar as balas disparadas pela nave espacial do jogador no jogo. Ela controla o movimento, a colisão e a renderização das balas.

Atributos

- posicaoX: A posição X da bala na tela em pixels.
- posicaoY: A posição Y da bala na tela em pixels.
- angulo: O ângulo de rotação da bala em radianos.
- velocidade: A velocidade da bala em pixels por segundo.
- debug: Uma flag que indica se o modo de debug está ativado.
- tela: Uma referência para a tela do jogo.
- raio: O raio da bala em pixels.
- contexto: Uma referência para o contexto de desenho da tela.
- dano: A quantidade de dano causada pela bala ao colidir com um inimigo.
- raioColisao: O raio da área de colisão da bala em pixels.
- colisao: Uma instância da classe Colisao que é responsável por detectar colisões entre a bala e outros objetos.
- spriteSheet: Uma imagem que representa os sprites da bala.
- frameWidth: A largura de cada quadro no sprite sheet em pixels.
- frameHeight: A altura de cada quadro no sprite sheet em pixels.
- currentFrame: O quadro atual exibido no sprite sheet.
- totalFrames: O número total de quadros no sprite sheet.
- frameRate: A taxa de atualização dos quadros (em milissegundos).
- frameCounter: O contador para controle da taxa de atualização dos quadros.

Construtor

O construtor da classe Bala inicializa os atributos da instância, define a imagem do sprite sheet, configura as configurações do sprite sheet e cria uma nova instância da classe Colisao.

Métodos

- atualizar(): Este método atualiza a posição da bala de acordo com sua velocidade e ângulo, e atualiza a posição da colisão. Ele também atualiza o contador de quadros para controlar a animação do sprite sheet.
- desenhar(): Este método salva o estado atual do contexto, move o contexto para a posição da bala, rotaciona o contexto para a direção da bala, desenha o quadro atual do sprite sheet e restaura o estado do contexto.
- saiuDaTela(): Este método verifica se a bala saiu da tela e retorna true se sim.

Exportação

A classe Bala é exportada como padrão, o que significa que pode ser importada na classe Nave usando a seguinte sintaxe:

```
import bala from "./balas.js";
```

Figura 4 - Importar classe Bala

2.1.5 - Classe Colisão

A classe Colisao é responsável por detetar colisões entre objetos. Vamos analisar os detalhes dessa classe:

Construtor

- raio: O raio da colisão em pixels.
- cor: A cor da colisão.
- posicaoX: A posição X da colisão.
- posicaoY: A posição Y da colisão.

Métodos

- desenharColisao(debug, contexto): Este método desenha a colisão na tela, mas apenas se o modo de debug estiver ativado.
- atualizarColisao(posicaoX, posicaoY): Este método atualiza a posição da colisão.
- detetarColisoes(ColisaoA, ColisaoB): Este método verifica se há colisão entre duas instâncias de Colisao e retorna true se houver colisão.

Exportação

A classe Colisao é exportada como padrão, o que significa que pode ser importada na classe Nave, Inimigos, Mundo e Balas usando a seguinte sintaxe:

```
import Colisao from "./colisao.js";
```

Figura 5 - Importar classe Colisao

2.1.6 - Classe UI

A classe Ui é responsável por gerenciar a interface do usuário (UI) no seu jogo. Ela lida com elementos visuais e interativos que os jogadores veem e interagem durante o jogo. Vamos analisar cada parte dessa classe:

Atributos

- tela: Representa o elemento de tela (canvas) onde a UI será desenhada.
- contexto: Representa o contexto de desenho no canvas.
- balaImage: Uma imagem que provavelmente representa o sprite das balas.
- frameWidth e frameHeight: As dimensões dos quadros do sprite de bala.
- backgroundImages: Uma lista de objetos que contém informações sobre as imagens de fundo e suas velocidades de rolagem.
- backgroundPositions: Uma lista que armazena as posições atuais das imagens de fundo.

Métodos

- desenharBackground(): Este método desenha os backgrounds na tela. Ele itera sobre as imagens de fundo, calcula suas posições e as desenha repetidamente para criar um efeito de rolagem. A velocidade de rolagem é controlada pelo atributo speed.
- desenharLabelVidaMundo(vida): Este método desenha o texto da "Vida do Mundo" na tela. Ele usa a fonte "Press Start 2P" e posiciona o texto no centro da tela.
- clearEcra(): Este método limpa o canvas, removendo qualquer desenho anterior.
- desenharLabelPontuacao(pontuacao): Este método desenha o texto da "Pontuação" na tela. Assim como o método anterior, ele também usa a fonte "Press Start 2P" e posiciona o texto no centro inferior da tela.
- desenharBalasNoPente(balasNoPente, maxBalasNoPente): Este método desenha as balas disponíveis no pente na tela. Ele calcula a posição correta para cada bala com base no índice e no deslocamento horizontal, e desenha a bala do sprite sheet na posição calculada.

A classe Ui é essencial para proporcionar uma experiência visual agradável e funcional para os jogadores. Ela combina elementos gráficos, texto e interatividade para criar uma interface intuitiva e envolvente. Esta classe é apenas importada na Classe main.

2.1.7 - Classe Main

A classe main desempenha um papel crucial na estrutura do jogo, coordenando a interação entre os diferentes componentes e garantindo que o jogo funcione de forma fluida e responsiva. A organização e implementação cuidadosas das funções e variáveis globais permitem um gerenciamento eficiente do estado do jogo, proporcionando uma experiência de usuário envolvente e dinâmica.

Importações

No início do código, são importados diversos módulos necessários para o funcionamento do jogo, tais como Mundo, Nave, Inimigo, ui e bala. Estas importações garantem que as diferentes partes do jogo (mundo, nave, inimigos, interface de usuário, e balas) estejam disponíveis e integradas na classe principal.

Inicialização de Elementos Gráficos e Áudio

Em seguida, são definidos os elementos gráficos e de áudio que serão utilizados durante o jogo. O canvas HTML é configurado para ocupar toda a janela do navegador, e são carregados os arquivos de áudio necessários para diferentes eventos do jogo, como música de fundo e efeitos sonoros.

Instanciação das Classes e Variáveis

As instâncias das classes Mundo, Nave, e ui são criadas, juntamente com variáveis globais que controlam o estado do jogo, como a pontuação, o nome do jogador e o controle de áudio.

Funções Principais

Ajuste do Canvas

A função ajustarTamanhoCanvas é responsável por garantir que o canvas do jogo se ajuste dinamicamente ao redimensionamento da janela do navegador.

Atualização do Jogo

A função atualizar é o núcleo do loop do jogo. Ela limpa a tela, desenha os elementos do jogo (mundo, nave, inimigos, interface de usuário), verifica colisões e atualiza o estado do jogo.

Criação e Reinicialização do Jogo

As funções criarInimigos e reiniciarJogo controlam a criação contínua de inimigos e o reinício do jogo após o término de uma partida, respetivamente.

Interação com o Utilizador

Diversas funções são implementadas para permitir a interação do jogador com o jogo, como showUsernamePopup para capturar o nome do jogador, enviarDadosParaFlask para enviar dados ao servidor, e funções para alternar o estado de mute do jogo.

2.2 - HTML

2.2.1 - Corpo da página

- overflow: hidden: Esconde as barras de rolagem.
- margin: 0; e padding: 0: Remove margens e preenchimentos padrões.
- background: url('background.jpg') no-repeat center center fixed: Define uma imagem de fundo centralizada e fixada.
- background-size: cover: Faz a imagem de fundo cobrir toda a tela.
- display: flex: Usa o modelo de layout flexível.
- flex-direction: column: Organiza os filhos em uma coluna.
- justify-content: center: Centraliza o conteúdo verticalmente.
- align-items: center: Centraliza o conteúdo horizontalmente.
- height: 100vh: Define a altura como 100% da altura da viewport.
- font-family: 'Press Start 2P', sans-serif: Define a fonte principal.

2.2.2 - Menu

- display: flex: Usa o layout flexível.
- flex-direction: column: Organiza os itens em uma coluna.
- align-items: center: Centraliza os itens.
- background-color: rgba(255, 255, 255, 0.1): Fundo branco semitransparente.
- padding: 20px: Adiciona espaço interno.
- border-radius: 10px: Bordas arredondadas.
- font-family: 'Press Start 2P', cursive: Fonte específica.

2.2.3 - Botões

- margin: 10px: Espaço externo ao redor do botão.
- padding: 15px 30px: Espaço interno (15px vertical e 30px horizontal).
- font-size: 16px: Tamanho da fonte.
- cursor: pointer: Mostra um cursor de ponteiro ao passar o mouse.
- background-color: #6200ea: Cor de fundo roxa.
- color: white: Texto branco.
- border: none: Remove bordas.

- border-radius: 5px: Bordas arredondadas.
- transition: background-color 0.3s, transform 0.3s: Transições suaves ao mudar a cor de fundo e transformar.
- font-family: 'Press Start 2P', cursive: Fonte específica.
- :hover e :active: Estilos ao passar o mouse e clicar, respectivamente.

2.2.4 - Título

- color: white: Cor do texto branca.
- margin-bottom: 20px: Espaço inferior.
- font-size: 34px: Tamanho da fonte.
- font-family: 'Press Start 2P', cursive: Fonte específica.

2.2.5 - Mensagens de "Game Over" e pontuação final

- display: none: Inicialmente oculto.
- font-family: 'Press Start 2P', cursive: Fonte específica.
- #gameOverMessage: Estilo de "Game Over":
- color: red: Texto vermelho.
- font-size: 34px: Tamanho da fonte.
- text-align: center: Texto centralizado.
- margin-bottom: 20px: Espaço inferior.
- #finalScore: Estilo de pontuação final:
- color: white: Texto branco.
- font-size: 22px: Tamanho da fonte.
- text-align: center: Texto centralizado.
- margin-bottom: 20px: Espaço inferior.

2.2.6 - Botão de mute

- position: absolute: Posicionamento absoluto.
- top: 10px: 10 pixels do topo.
- right: 10px: 10 pixels da direita.
- z-index: 9999: Alta prioridade na pilha de camadas.

2.2.7 - Modal

- display: none: Inicialmente oculto.
- position: fixed: Posicionamento fixo.
- z-index: 1000: Prioridade na pilha de camadas.
- left: 0; e top: 0: Posicionado no canto superior esquerdo.
- width: 100%; e height: 100%: Ocupa toda a tela.
- overflow: auto: Rolagem automática se necessário.
- backdrop-filter: blur(5px): Desfoque do fundo.
- background-color: rgba(0, 0, 0, 0.5): Fundo preto semitransparente.
- justify-content: center; e align-items: center: Centraliza o conteúdo.
- opacity: 0: Transparência inicial.
- animation: fadeIn 1s forwards: Animação de aparecimento.

2.2.8 - Conteúdo do modal

- background-color: rgba(18, 18, 64, 0.9): Fundo escuro semitransparente.
- color: #fff: Texto branco.
- padding: 20px: Espaço interno.
- border: 1px solid #121240: Borda sólida.
- width: 80%; e max-width: 400px: Largura ajustável.
- text-align: center: Texto centralizado.
- display: flex; e flex-direction: column: Layout flexível.
- align-items: center: Centraliza os itens.
- font-family: 'Press Start 2P', cursive: Fonte específica.
- border-radius: 10px: Bordas arredondadas.
- margin-bottom: 40px: Espaço inferior.
- opacity: 1: Transparência.

2.2.9 - Texto e inputs dentro do modal

- modal p: Parágrafo no modal:
- margin-bottom: 20px: Espaço inferior.
- font-size: 14px: Tamanho da fonte.
- .modal input: Inputs no modal:

- padding: 10px: Espaço interno.
- width: 100%; e max-width: 300px: Largura ajustável.
- margin-bottom: 10px: Espaço inferior.
- border: 1px solid #121240: Borda sólida.
- border-radius: 5px: Bordas arredondadas.
- text-align: center: Texto centralizado.
- font-family: 'Press Start 2P', cursive: Fonte específica.
- font-size: 14px: Tamanho da fonte.
- background-color: #08081a: Fundo escuro.
- color: #fff: Texto branco.

2.3 - Python Flask

O objetivo principal do servidor é gerenciar uma "scoreboard" (tabela de pontuações), recebendo dados via POST e servindo arquivos estáticos.

O código começa importando as funções necessárias do Flask: Flask, send_from_directory e request. Em seguida, cria uma instância da aplicação Flask chamada app.

2.3.1 - Função organizar_scoreboard

Esta função lê o arquivo scoreboard.txt para extrair os nomes e as pontuações dos jogadores. Os dados são organizados em uma lista de tuplas e, em seguida, ordenados em ordem decrescente de pontuação. O arquivo é reescrito com os dados organizados.

2.3.2 - Rota Principal /

Esta rota serve o arquivo index.html diretamente da pasta raiz do servidor quando a URL raiz (/) é acessada.

2.3.3 - Rota para Arquivos Estáticos

Esta rota serve qualquer arquivo estático presente na pasta raiz. Isso permite que o servidor entregue arquivos como CSS, JavaScript, imagens, etc.

2.3.4 - Rota para Receber Dados via POST /enviar_dados

Esta rota recebe dados JSON via um método POST. Ela extrai o nome e a pontuação do jogador dos dados recebidos e os adiciona ao arquivo scoreboard.txt. Depois de adicionar os novos dados, a função organizar_scoreboard é chamada para reordenar o arquivo. Finalmente, retorna uma mensagem de sucesso.

2.3.5 - Execução da Aplicação

Esta linha faz com que a aplicação Flask seja executada quando o script é executado diretamente. O parâmetro debug=True ativa o modo de depuração, que é útil durante o desenvolvimento, mas deve ser desativado em produção.

Capítulo 3 - Problemas e Bugs

Durante o desenvolvimento de Operation Astroshield, encontrámos vários problemas e bugs que foram resolvidos ao longo do processo. Estes incluem:

- Alinhamento de Colisões: As colisões não estavam corretamente alinhadas com os objetos, causando problemas na detecção de impactos.
- Exibição do Menu ao Perder: O menu não aparecia quando o jogador perdia o jogo, dificultando a navegação pós-jogo. Este problema foi corrigido garantindo que o menu fosse acionado após a perda.
- Exibição do Menu ao Iniciar: O menu não aparecia ao iniciar o jogo, o que criava confusão para o jogador ao começar uma nova partida. Esta questão foi resolvida assegurando que o menu fosse exibido corretamente ao início.
- Funcionamento do Botão "Jogar": O jogo não iniciava ao clicar no botão "Jogar", impedindo o início da experiência de jogo. Este bug foi resolvido corrigindo a ligação entre o botão e a função de início do jogo.
- Funcionalidade do Botão "Mute": O botão "Mute" apenas silenciava a música do jogo e não a do menu. Além disso, ao morrer, o modal para inserir o nome do jogador não aparecia. Estas questões foram resolvidas ajustando a função de mute para abranger todo o áudio e garantindo a exibição do modal.
- Eficácia das Balas: As balas disparadas pelos jogadores não estavam a eliminar os inimigos como esperado. Este problema foi solucionado ajustando a lógica de colisão das balas.
- Regeneração das Balas: As balas não regeneravam, limitando a capacidade do jogador de continuar a disparar. Este bug foi corrigido implementando um sistema de regeneração adequado.
- Sprites das Balas: As balas não exibiam o sprite correto, afetando a representação visual durante o jogo. Este problema foi resolvido assegurando que o sprite correto fosse carregado.
- Contagem de Vidas: O sistema de contagem de vidas não estava a funcionar corretamente, resultando em valores negativos e não realistas. Este bug foi corrigido garantindo uma contagem de vidas precisa.

A resolução destes problemas foi fundamental para melhorar a experiência geral do jogo e assegurar que todos os elementos funcionem conforme esperado.

Capítulo 4 - Conclusões

Operation Astroshield representa uma fusão harmoniosa entre a nostalgia dos clássicos jogos arcade e a inovação das tecnologias web modernas. Ao longo deste projeto, foram superados diversos desafios técnicos e criativos para desenvolver um jogo que não só honra a essência de "Asteroids", mas também introduz novas funcionalidades que melhoram a experiência do jogador.

A implementação de HTML, JavaScript e Python Flask permitiu a criação de uma interface intuitiva, uma lógica de jogo robusta e uma gestão eficiente dos dados dos jogadores. O sistema de dificuldade dinâmico e o armazenamento das pontuações proporcionam um ambiente competitivo e envolvente, incentivando os jogadores a melhorar constantemente as suas habilidades.

Com este projeto, não apenas recriámos um clássico, mas também estabelecemos uma base sólida para futuras expansões e melhorias. Operation Astroshield é mais do que um simples jogo; é uma plataforma que pode evoluir continuamente, adaptando-se às necessidades e preferências dos jogadores. Esperamos que esta experiência tenha sido tão gratificante para os jogadores como foi para nós desenvolvê-la, e estamos entusiasmados com as possibilidades futuras deste projeto.