Relatório Técnico: Jogo "Sobrevivência Cósmica"

Autor: Italo Butinholi Mendes, Gustavo Correa e LLM (Gemini)

Data: 07 de outubro de 2025

Tecnologia: HTML5 Canvas, JavaScript (ES6+)

1. Introdução

Este relatório detalha a arquitetura técnica e as mecânicas implementadas no jogo 2D de nave "Sobrevivência Cósmica". O projeto foi desenvolvido inteiramente em JavaScript puro, utilizando a API do HTML5 Canvas para renderização, sem a necessidade de bibliotecas ou motores de jogo externos.

O objetivo do projeto foi criar uma experiência de jogo arcade, do gênero *shoot 'em up* vertical, que fosse leve, responsiva e demonstrasse conceitos importantes de desenvolvimento de jogos, como gestão de estados, dificuldade progressiva e geração de gráficos procedurais.

2. Arquitetura do Jogo

A estrutura do código foi organizada de forma modular e orientada a objetos para garantir clareza, manutenção e escalabilidade.

2.1. Motor de Jogo (Game Loop)

O coração do jogo é o gameLoop, controlado por requestAnimationFrame. Esta abordagem garante que a renderização seja sincronizada com os ciclos de atualização do monitor, resultando em animações fluidas e eficientes.

A lógica é desacoplada da taxa de quadros (FPS) através do uso de um parâmetro deltaTime, que representa o tempo decorrido desde o último quadro. Todas as atualizações de movimento, timers e animações são multiplicadas por deltaTime para garantir que a velocidade do jogo seja consistente em diferentes hardwares.

2.2. Máquina de Estados Finitos (FSM)

O jogo opera com uma máquina de estados simples para gerenciar os diferentes contextos de jogabilidade:

GameState.MENU: A tela inicial, onde o jogador aguarda para iniciar a partida.

GameState.PLAYING: O estado principal, onde toda a lógica de jogabilidade (movimento, disparos, colisões) é executada.

GameState.GAME_OVER: A tela final, exibida após o jogador ser derrotado, mostrando a pontuação.

A variável currentGameState controla o estado atual, e uma estrutura switch na função draw() determina quais elementos devem ser renderizados em cada estado, separando a lógica de apresentação da lógica de jogo.

2.3. Estrutura de Classes

O código é componentizado nas seguintes classes principais:

InputHandler: Captura e gerencia eventos de teclado (keydown, keyup), armazenando as teclas atualmente pressionadas em um array para fácil acesso.

Player: Representa a nave do jogador. Controla sua posição, movimento, lógica de disparo (incluindo cooldown) e renderização.

Enemy: Representa as naves inimigas. Gerencia seu movimento vertical, IA de disparo e ciclo de vida.

Projectile / EnemyProjectile: Classes para os projéteis do jogador e dos inimigos, respectivamente. Controlam sua velocidade, direção e condição de remoção.

Particle: Uma classe genérica para criar efeitos visuais, como explosões e rastros de propulsores.

Background / Layer: Sistema responsável pelo fundo de estrelas com efeito de paralaxe, criando uma sensação de profundidade e movimento.

3. Mecânicas Principais

3.1. Dificuldade Progressiva e Aleatoriedade

Para evitar uma jogabilidade repetitiva, a dificuldade aumenta dinamicamente com o tempo.

Cadência de Tiro Inimiga: A classe Enemy calcula o shootCooldown (intervalo entre disparos) com base na variável global gameTime. Quanto mais tempo o jogador sobrevive, menor se torna o intervalo de tempo para os disparos, aumentando a intensidade dos ataques.

Aleatoriedade: O shootCooldown de cada inimigo é definido dentro de um intervalo aleatório que se estreita com o tempo. Isso evita que todos os inimigos atirem em uníssono, criando padrões de ataque mais orgânicos e desafiadores.

3.2. Sistema de Colisão

A detecção de colisão é realizada na função checkCollisions e utiliza o método *Axis-Aligned Bounding Box* (AABB), que verifica a sobreposição de retângulos. As seguintes interações são monitoradas a cada quadro:

Projéteis do jogador vs. Naves inimigas.

Nave do jogador vs. Naves inimigas (colisão direta).

Projéteis inimigos vs. Nave do jogador.

Quando uma colisão é detectada, os objetos envolvidos são marcados para remoção (markedForDeletion), e os efeitos correspondentes (pontuação, explosão, fim de jogo) são acionados.

4. Componentes Visuais e Efeitos

4.1. Gráficos Procedurais

Todos os elementos visuais do jogo (naves, projéteis, partículas) são desenhados dinamicamente através da API do Canvas. Esta abordagem elimina a necessidade de carregar arquivos de imagem (sprites), tornando o jogo extremamente leve e rápido para carregar. Gradientes e sombras são utilizados para adicionar profundidade e estilo aos desenhos.

4.2. Fundo Dinâmico com Paralaxe

O cenário espacial é composto por múltiplas camadas (Layer) de estrelas, cada uma se movendo em uma velocidade diferente, criando um efeito de paralaxe convincente.

Progressão Visual: A classe Background atualiza continuamente uma variável de matiz (hue) com base no gameTime. Essa matiz é usada para colorir as estrelas e o fundo do canvas com o padrão HSL (Hue, Saturation, Lightness). Como resultado, o cenário transita suavemente por todo o espectro de cores (azul, roxo, vermelho etc.), dando ao jogador um feedback visual claro de sua progressão e da passagem do tempo no jogo.

4.3. Sistema de Partículas

A classe Particle é utilizada para adicionar "juice" (polimento visual) ao jogo.

Explosões: Quando um inimigo é destruído ou o jogador é derrotado, uma rajada de partículas é gerada no local da colisão.

Propulsores: Partículas são emitidas continuamente pela traseira da nave do jogador enquanto ela se move, criando um efeito de rastro.

Cada partícula tem um ciclo de vida limitado, e sua opacidade diminui com o tempo, fazendo-a desaparecer suavemente.

5. Regras e Boas Práticas Adotadas

O desenvolvimento do projeto seguiu um conjunto de diretrizes para garantir a qualidade, performance e organização do código.

Linguagem: O jogo foi construído utilizando exclusivamente JavaScript puro (ES6+) e a API do HTML5 Canvas, sem dependência de frameworks ou bibliotecas externas, conforme o requisito.

Assets: A implementação atual utiliza gráficos 100% procedurais, eliminando a necessidade de carregar assets externos. No entanto, a boa prática de creditar artistas de sprites e sons gratuitos (de plataformas como itch.io e OpenGameArt) é reconhecida e seria aplicada em futuras versões que utilizassem tais recursos.

Performance: Para otimizar o desempenho, a criação de novos objetos (new) dentro do gameLoop foi minimizada. Objetos como player e background são instanciados apenas uma vez. Arrays que contêm entidades dinâmicas (projéteis, inimigos, partículas) são gerenciados de forma eficiente, filtrando e removendo objetos marcados para exclusão em vez de recriar os arrays a cada quadro.

Organização: O código mantém uma clara separação entre a lógica de jogo (update()) e a renderização (draw()). A função update() é responsável por todos os cálculos de estado, física e IA, enquanto a função draw() se encarrega apenas de desenhar o estado atual no canvas.

Boas Práticas de IA: O desenvolvimento deste projeto foi assistido pela LLM Gemini. Os prompts utilizados foram iterativos, evoluindo o jogo passo a passo. A seguir, os principais prompts que levaram à versão final do código:

Planejamento e Concepção Inicial: "Atue como desenvolvedor de jogos 2D sênior, especialista em HTML5 Canvas. Estamos criando um jogo com a temática 'Nave no espaço'. Sugira 3 ideias de jogo, detalhando para cada uma: 1. Mecânicas centrais, 2. Entidades do jogo, 3. Estados do jogo, 4. Eventos de teclado, 5. Uso de paralaxe, 6. Colisão, 7. Spritesheet/Clipping, 8. Disparo. Para a ideia mais promissora, crie uma lista de tarefas."

Estrutura inicial: "Crie um projeto base de jogo 2D em Canvas com os seguintes arquivos: index.html, style.css, js/main.js."

Implementação da versão com gráficos procedurais: "Melhore visualmente a minha nave, os obstáculos, etc." (solicitando a substituição das formas geométricas simples por desenhos mais detalhados).

Implementação da IA inimiga: "Os inimigos poderiam ser naves que atiram também, e se pegar o projeteis em mim eu perco também morrendo/explodindo".

Implementação da dificuldade progressiva (disparos): "Podia aumentar a cadência de tiro dos inimigos, mas que seja aleatório".

Implementação da dificuldade progressiva (visual): "Poderia modificar o fundo também com o passar do tempo para mostrar uma progressão".

Nave no espaçoLinguagem: O jogo foi construído utilizando exclusivamente JavaScript puro (ES6+) e a API do HTML5 Canvas, sem dependência de frameworks ou bibliotecas externas, conforme o requisito.

Assets: A implementação atual utiliza gráficos 100% procedurais, eliminando a necessidade de carregar assets externos. No entanto, a boa prática de creditar artistas de sprites e sons gratuitos (de plataformas como itch.io e OpenGameArt) é reconhecida e seria aplicada em futuras versões que utilizassem tais recursos.

Performance: Para otimizar o desempenho, a criação de novos objetos (new) dentro do gameLoop foi minimizada. Objetos como player e background são instanciados apenas uma vez. Arrays que contêm entidades dinâmicas (projéteis, inimigos, partículas) são gerenciados de forma eficiente, filtrando e removendo objetos marcados para exclusão em vez de recriar os arrays a cada quadro.

Organização: O código mantém uma clara separação entre a lógica de jogo (update()) e a renderização (draw()). A função update() é responsável por todos os cálculos de estado, física e IA, enquanto a função draw() se encarrega apenas de desenhar o estado atual no canvas.

Boas Práticas de IA: O desenvolvimento deste projeto foi assistido pela LLM Gemini. Os prompts utilizados focaram na criação de estruturas de jogo, implementação de mecânicas específicas (como dificuldade progressiva e efeitos de partículas) e na refatoração e depuração de código.

6. Conclusão

"Sobrevivência Cósmica" é um protótipo de jogo funcional e bem estruturado que demonstra a viabilidade de criar experiências de jogo complexas usando apenas tecnologias web padrão. A arquitetura modular, a dificuldade dinâmica e os gráficos procedurais, desenvolvidos seguindo boas práticas de performance e organização, formam uma base sólida que pode ser facilmente expandida com novos tipos de inimigos, power-ups e funcionalidades.