Documentação do Sistema: Hear-Oh!

1. Visão Geral do Projeto

O Hear-Oh! é uma Aplicação Web de Página Única (SPA - Single Page Application) desenvolvida com foco em acessibilidade para deficientes visuais. O sistema utiliza a arquitetura baseada em componentes do React para gerenciar os estados do jogo e APIs nativas dos navegadores modernos para renderização de áudio espacial 3D e síntese de voz.

2. Stack Tecnológica (Ferramentas Utilizadas)

- Frontend Framework: React.js (com uso intenso de React Hooks: useState, useEffect, useRef).
- Estilização: CSS embutido e Tailwind CSS (para prototipagem rápida e design de alto contraste focado na supervisão visual).
- Ícones: Lucide-React (renderização de SVGs leves para a interface de suporte).
- Motor Sonoro (Audio Engine): Web Audio API (API nativa do JavaScript para processamento e reprodução de arquivos sonoros com espacialização em tempo real).
- Motor de Voz (TTS Engine): Web Speech API (window.speechSynthesis) para leitura de tela dinâmica sem necessidade de bibliotecas externas.

3. Arquitetura e Gerenciamento de Estado

A aplicação concentra sua lógica principal no componente <App />, operando como uma máquina de estados finitos (FSM).

3.1. Estados Globais (useState)

- gameState: Controla o ciclo de vida do jogo ('unlock', 'menu', 'tutorial', 'playing', 'win', 'gameover').
- gameMode: Define as regras de persistência ('story' para fases sequenciais ou 'infinite' para geração procedural).
- coords: Armazena a posição atual do jogador em um objeto { x, y }.
- lives e score: Gerenciam a saúde do jogador e a pontuação atual.
- entities: Objeto que armazena os arrays de entidades ativas na matriz (itens, cristais, armadilhas).
- hasShield: Booleano que define se o jogador possui proteção temporária (hit block).

3.2. Referências Mutáveis (useRef)

- audioCtx: Mantém a instância global do AudioContext, garantindo que o áudio seja instanciado apenas uma vez, após a interação do usuário.
- masterGain: Nó de ganho global, usado para controlar o volume mestre da aplicação e evitar distorções de áudio.
- audioBuffers: Objeto em cache (memória) usado para armazenar os arquivos de áudio já decodificados, garantindo reprodução imediata.

4. Estruturas de Dados e Matriz (Grid)

O ambiente do jogo é uma grade virtual de 10x10 coordenadas cartesianas (indo de X:0, Y:0 até X:9, Y:9). Os níveis do Modo História são armazenados em uma constante estruturada estaticamente (DATABASE_LEVELS), que contém:

- id e name: Identificadores do nível.
- items, crystals, traps: Arrays contendo as coordenadas {x, y} fixas onde cada entidade será instanciada.

5. Motores Principais (Core Engines)

5.1. Motor de Áudio Espacial (playSonicCue)

O sistema utiliza arquivos de áudio externos selecionados especificamente para cada acontecimento do jogo (formatos de .mp3., para armadilhas, tesouros, passos, vitórias, fim de jogo). Estes arquivos são pré-carregados, decodificados via audioCtx.decodeAudioData e armazenados em memória (buffers) durante a inicialização para garantir reprodução com latência zero.

Sempre que uma ação ocorre ou o "radar" examina o ambiente, a função playSonicCue é disparada. Ela reproduz o arquivo sonoro mapeado usando um nó createBufferSource(), aplicando as seguintes manipulações de áudio 3D (espacialização):

- Balanço (Pan): O nó createStereoPanner recebe um valor de -1 (esquerda total) a 1 (direita total). Esse valor é calculado baseando-se na diferença entre a coordenada X do jogador e a do objeto (objeto.x - jogador.x). Isso permite que o jogador identifique rapidamente em qual lado do fone de ouvido o objeto está localizado.
- Atenuação de Volume: A distância entre as coordenadas do jogador e da entidade no mapa (calculada via Teorema de Pitágoras) determina o volume ajustado pelo nó de ganho (createGain). Assim, o áudio do .mp3 soa cada vez mais alto e nítido à medida que o jogador se aproxima de sua fonte de origem na grade.

5.2. Motor de Lógica de Colisão (handleMove)

O input do teclado intercepta as *Arrow Keys* (Setas Direcionais) via evento keydown. A função handleMove(dx, dy) realiza as seguintes verificações:

1. Verificação de Limites: Avalia se nextX ou nextY estão fora do intervalo 0-9.
2. Scanner de Proximidade: Calcula a distância euclidiana para todos os objetos ativos e dispara o Motor de Áudio para cada um, criando um "radar".
3. Resolução de Colisão: Verifica interseção exata de coordenadas entre o jogador e elementos da matriz, atualizando o estado de lives ou coletando items.

5.3. Geração Procedural (Modo Infinito)

A função generateInfiniteLevel cria níveis dinâmicos utilizando a função getSafePosition(minDistance). O algoritmo gera coordenadas randômicas e calcula a distância para todas as entidades previamente geradas no nível, garantindo que recompensas e armadilhas nunca dividam a mesma célula e mantenham um espaçamento acústico mínimo para evitar sobreposição (clipping) de áudio.

6. Segurança e Inicialização (Autoplay Policy)

Devido às restrições modernas dos navegadores, a Web Audio API inicia em estado suspended. O estado inicial da aplicação (gameState === 'unlock') exige uma ação explícita (evento global de tecla) que chama a função audioCtx.current.resume(), desbloqueando o hardware de som do dispositivo do usuário de forma segura.