Wake Word Functionality Preservation Guide

Overview

The wake word system (activation phrase: "computer") is a core feature of RENT A HAL, allowing voice interaction through different modes including chat, vision, imagine, weather, and Gmail. This functionality is particularly complex because it involves:

1. Speech recognition with state management
2. Multi-platform compatibility (especially iOS/Safari challenges)
3. Error handling and recovery
4. Audio visualization
5. Transitions between different command modes

Original Implementation Analysis

In script.js, the wake word functionality is implemented with these key components:

Core State Management

javascript

```
// State variables
let wakeWordState = 'inactive'; // 'inactive', 'listening', 'menu', 'prompt', 'processing', 'gm
let wakeWordRecognition;
let isListening = false;
let isSystemSpeaking = false;
let isWakeWordModeActive = false;
```

Initialization

javascript

```javascript
function initializeWakeWordRecognition() {
    const SpeechRecognition = window.SpeechRecognition || window.webkitSpeechRecognition;
    if (SpeechRecognition) {
        wakeWordRecognition = new SpeechRecognition();
        wakeWordRecognition.lang = 'en-US';
        wakeWordRecognition.interimResults = false;
        wakeWordRecognition.maxAlternatives = 1;
        wakeWordRecognition.continuous = false;

        // Event handlers...
    }
}
```

Command Processing

javascript

```javascript
function handleTopLevelCommand(command) {
    clearTimeout(inactivityTimer);
    if (command.includes("computer")) {
        wakeWordState = 'menu';
        inactivityCount = 0;
        speakAndListen("What would you like to do? Say the MODE.", handleMenuCommand);
    } else if (command.includes("goodbye")) {
        deactivateWakeWordMode();
    } else {
        inactivityCount++;
        if (inactivityCount >= 2) {
            speakFeedback(" ");
            deactivateWakeWordMode();
        } else {
            if (isWakeWordModeActive) {
                speakAndListen(" ", handleTopLevelCommand);
            } else {
                handleTopLevelCommand("");
            }
        }
    }
    startInactivityTimer();
}
```

Mode Handling

```javascript
function handleMenuCommand(command) {
    // Handle different modes: chat, vision, imagine, weather, gmail
}

function handlePromptInput(command) {
    // Handle input during prompt mode
}
```

Speech Output

```javascript
function speakFeedback(message, callback) {
    // Text-to-speech functionality
}

function speakAndListen(message, callback) {
    // Speak then listen for response
}
```

SpeechManager Implementation

The refactored SpeechManager should maintain all this functionality while improving organization and maintainability. Here's how to ensure full feature preservation:

1. State Management

Implement a complete state tracking system:

javascript

```javascript
export class SpeechManager {
    constructor(websocketManager) {
        // Manager reference
        this.websocket = websocketManager;

        // Feature manager references (set by App.js)
        this.vision = null;
        this.weather = null;
        this.gmail = null;

        // State management
        this.wakeWordState = 'inactive'; // 'inactive', 'listening', 'menu', 'prompt', 'process
        this.isListening = false;
        this.isSystemSpeaking = false;
        this.recognitionPaused = false;
        this.inactivityCount = 0;
        this.promptInactivityCount = 0;

        // Timeout tracking
        this.inactivityTimer = null;
        this.promptInactivityTimer = null;
        this.recognitionTimeout = null;

        // Error tracking for recovery
        this.errorCounts = {
            noSpeech: 0,
            audioCapture: 0,
            network: 0,
            aborted: 0
        };
        this.lastError = null;
        this.lastErrorTime = null;

        // Platform detection
        this.isIOS = /iPad|iPhone|iPod/.test(navigator.userAgent);
        this.isSafari = /^((?!chrome|android).)*safari/i.test(navigator.userAgent);
        this.isMobile = /Mobi|Android/i.test(navigator.userAgent);
    }

    // Additional properties and methods...
}
```

2. Recognition Initialization with Error Handling

Implement robust speech recognition initialization:

```
javascript
```

```
async initializeRecognition() {
    console.log("[DEBUG] Initializing speech recognition");

    // Stop existing recognition if any
    if (this.recognition) {
        try {
            this.recognition.stop();
        } catch (e) {
            console.log("[DEBUG] Error stopping existing recognition:", e);
        }
    }

    const SpeechRecognition = window.SpeechRecognition || window.webkitSpeechRecognition;
    if (!SpeechRecognition) {
        console.error("[ERROR] Speech recognition not supported in this browser");
        return false;
    }

    // Create new recognition instance
    this.recognition = new SpeechRecognition();

    // Platform-specific configurations
    if (this.isIOS || this.isSafari) {
        this.recognition.continuous = false;
        this.recognition.interimResults = false;
    } else {
        this.recognition.continuous = true;
        this.recognition.interimResults = true;
    }

    this.recognition.lang = 'en-US';
    this.recognition.maxAlternatives = 1;

    // Set up event handlers
    this.setupRecognitionHandlers();

    return true;
}

setupRecognitionHandlers() {
    this.recognition.onstart = () => {
        console.log("[DEBUG] Recognition started");
        this.isListening = true;
```

```
            this.showWaveform();
            clearTimeout(this.recognitionTimeout);
        };

        this.recognition.onend = () => {
            console.log("[DEBUG] Recognition ended");
            this.isListening = false;

            if (this.wakeWordState !== 'inactive' && !this.recognitionPaused) {
                console.log("[DEBUG] Restarting recognition");
                setTimeout(() => {
                    if (!this.isSystemSpeaking && !this.isListening) {
                        try {
                            this.startListening();
                        } catch (error) {
                            console.error("Error restarting recognition:", error);
                        }
                    }
                }, 250);
            } else {
                this.hideWaveform();
            }
        };

        this.recognition.onerror = (event) => {
            this.handleRecognitionError(event);
        };

        this.recognition.onresult = (event) => {
            this.handleRecognitionResult(event);
        };
    }
```

3. Error Handling and Recovery

Implement comprehensive error handling and recovery mechanisms:

javascript

```javascript
async handleRecognitionError(event) {
    console.error("[ERROR] Recognition error:", event.error);
    this.isListening = false;
    this.lastError = event.error;
    this.lastErrorTime = Date.now();

    if (this.isIOS && event.error === 'not-allowed') {
        await this.speakFeedback("Please enable microphone access in your iOS settings.");
        return;
    }

    switch(event.error) {
        case 'no-speech':
            this.errorCounts.noSpeech++;
            if (this.errorCounts.noSpeech < 3) {
                setTimeout(() => {
                    if (!this.isSystemSpeaking && !this.recognitionPaused) {
                        this.startListening();
                    }
                }, 100);
            } else {
                await this.handleRecovery('no-speech');
            }
            break;

        case 'audio-capture':
            this.errorCounts.audioCapture++;
            await this.handleRecovery('audio-capture');
            break;

        case 'network':
            this.errorCounts.network++;
            await this.
```