Snake Evolution - API Reference v1.0

Document Header

Project: Snake Evolution Game
Document Type: API Reference Documentation
Version: 1.0
Date: November 2025
Status: Production Ready
Prepared by: Senior Software Architect
Standard: Based on OpenAPI 3.0 principles adapted for client-side modules

Executive Summary

This document provides **formal interface specifications** for all public-facing modules in Snake Evolution v2.1. Developers can use these specifications to understand module capabilities, integrate with the core engine, extend functionality, and write reliable tests.

Document Purpose

- **Who should read:** Developers extending the game, building tests, or integrating with external systems
- **What's covered:** Public APIs, method signatures, parameter types, return values, error handling
- **What's NOT covered:** Implementation details (see Technical Analysis for internals)

1. Core Module Interfaces

1.1 GameEngine Interface

Purpose: Orchestrates the entire game lifecycle and state management.

Methods

`initialize(config?: GameConfig): Promise<void>`

Initializes the game engine with optional configuration.

```
/**
 * Initialize game engine
 * @param {GameConfig} config - Optional configuration object
 * @param {number} config.gridWidth - Grid width (default: 20)
 * @param {number} config.gridHeight - Grid height (default: 20)
 * @param {number} config.initialSpeed - Speed in cells/sec (default: 5)
```

```
 * @param {boolean} config.soundEnabled - Enable audio (default: true)
 * @param {string} config.renderTargetId - Canvas element ID (default: 'gameCanvas')
 * @returns {Promise&lt;void&gt;} Resolves when engine is ready
 * @throws {GameInitializationError} If initialization fails
 */
```

Example Usage:

```
const engine = new GameEngine();
await engine.initialize({
  gridWidth: 20,
  gridHeight: 20,
  soundEnabled: true
});
```

`start(): Promise<void>`

Starts the game loop and begins gameplay.

```
/**
 * Start the game loop
 * @returns {Promise&lt;void&gt;} Resolves when game loop starts
 * @throws {GameStateError} If game is already running
 */
```

State Transition: MENU → INIT → PLAYING

`pause(): void`

Pauses the current game without losing state.

```
/**
 * Pause the game
 * @returns {void}
 * @throws {GameStateError} If game is not playing
 *
 * Note: Game state is preserved. Resume with resume()
 */
```

State Transition: PLAYING → PAUSED

`resume(): void`

Resumes a paused game.

```
/**
 * Resume a paused game
 * @returns {void}
```

```
 * @throws {GameStateError} If game is not paused
 */
```

State Transition: PAUSED → PLAYING

`gameOver(): Promise&lt;void&gt;`

Triggers game over condition and stops the game loop.

```
/**
 * End the current game
 * @returns {Promise&amp;lt;void&amp;gt;} Resolves when game over sequence completes
 */
```

State Transition: PLAYING → GAMEOVER

`getGameState(): GameState`

Returns the current game state snapshot.

```
/**
 * Get current game state
 * @returns {GameState} Immutable snapshot of current game state
 *
 * @typedef {Object} GameState
 * @property {number} score - Current score (length)
 * @property {number} length - Snake length in cells
 * @property {EvolutionStage} stage - Current evolution stage (1-5)
 * @property {Point} snakeHead - Head position {x, y}
 * @property {Point[]} snakeBody - Body segments
 * @property {Point} food - Food position {x, y}
 * @property {boolean} isPlaying - True if game active
 * @property {string} gameState - Current state: 'MENU'|'PLAYING'|'PAUSED'|'GAMEOVER'
 * @property {number} elapsedTime - Milliseconds since game start
 * @property {PerformanceMetrics} metrics - Current FPS and timing info
 */
```

Example Return Value:

```
{
  score: 127,
  length: 127,
  stage: 4,  // Empowered
  snakeHead: { x: 10, y: 10 },
  snakeBody: [{ x: 10, y: 11 }, { x: 10, y: 12 }, ...],
  food: { x: 15, y: 8 },
  isPlaying: true,
  gameState: 'PLAYING',
  elapsedTime: 12450,
```

```
  metrics: { avgFPS: 58.5, p95FrameTime: 16.2 }
}
```

`isOperational(): boolean`

Checks if engine is in a healthy state (not crashed).

```
/**
 * Check if game engine is operational
 * @returns {boolean} True if engine can accept commands
 *
 * Returns false if:
 * - Engine not initialized
 * - Fatal error occurred
 * - Too many errors detected
 */
```

1.2 StateManager Interface

Purpose: Manages finite state machine with atomic transitions and recovery.

Methods

`transitionState(newState: GameState, context?: any): Promise&lt;TransitionResult&gt;`

Performs atomic state transition with automatic rollback on error.

```
/**
 * Transition to new state
 * @param {GameState} newState - Target state to transition to
 * @param {*} context - Optional context data for state entry hooks
 * @returns {Promise&amp;lt;TransitionResult&amp;gt;} Transaction result
 *
 * @typedef {Object} TransitionResult
 * @property {boolean} success - True if transition succeeded
 * @property {GameState} previousState - State before transition
 * @property {string} error - Error message if failed
 *
 * Valid Transitions:
 * MENU        → INIT
 * INIT        → PLAYER_INPUT
 * PLAYER_INPUT → PLAYING
 * PLAYING     → PAUSED, GAMEOVER
 * PAUSED      → PLAYING, MENU
 * GAMEOVER    → MENU
 */
```

Example Usage:

```
const result = await stateManager.transitionState('PLAYING');
if (!result.success) {
  console.error('Transition failed:', result.error);
}
```

`getCurrentState(): GameState`

Returns current state (synchronous).

```
/**
 * Get current state
 * @returns {GameState} Current state string
 */
```

`getStateHistory(): GameState[]`

Returns history of state transitions.

```
/**
 * Get state history
 * @returns {GameState[]} Array of states in order
 * @note Limited to last 100 states to prevent memory bloat
 */
```

1.3 InputManager Interface

Purpose: Processes and validates player input through validation pipeline.

Methods

`processInput(input: Direction): Promise&lt;InputResult&gt;`

Processes directional input through validation pipeline.

```
/**
 * Process player input
 * @param {Direction} input - Direction: 'UP'|'DOWN'|'LEFT'|'RIGHT'
 * @returns {Promise&amp;lt;InputResult&amp;gt;} Processing result
 *
 * @typedef {Object} InputResult
 * @property {boolean} accepted - True if input was processed
 * @property {string} rejectionReason - Why rejected (if not accepted)
 * @property {number} queueLength - Current queue depth
 *
 * Pipeline stages:
 * 1. Rate limiting (50ms debounce)
 * 2. Direction validation (4-way valid)
```

```
 * 3. Duplicate filtering (no consecutive identical)
 * 4. 180° turn prevention (no reverse turns)
 * 5. Queue buffering (up to 3 ahead inputs)
 */
```

Example Usage:

```
const result = await inputManager.processInput('UP');
if (!result.accepted) {
  console.log('Input rejected:', result.rejectionReason);
  console.log('Queue depth:', result.queueLength);
}
```

`registerInputListener(callback: (direction: Direction) => void): () => void`

Registers listener for processed inputs.

```
/**
 * Register input listener
 * @param {Function} callback - Called when input is processed
 * @returns {Function} Unsubscribe function
 *
 * Callback receives direction: 'UP'|'DOWN'|'LEFT'|'RIGHT'
 */
```

Example Usage:

```
const unsubscribe = inputManager.registerInputListener((direction) =&gt; {
  console.log('Player moved:', direction);
});

// Later: unsubscribe()
```

1.4 CollisionDetector Interface

Purpose: Detects collisions using spatial hashing with fallback verification.

Methods

`detectCollision(context: CollisionContext): boolean`

Detects if snake head collides with walls or itself.

```
/**
 * Detect collision
 * @param {CollisionContext} context - Collision detection context
 *
```

```
 * @typedef {Object} CollisionContext
 * @property {Point} snakeHead - Head position {x, y}
 * @property {Point[]} snakeBody - Body segments array
 * @property {number} gridWidth - Game grid width
 * @property {number} gridHeight - Game grid height
 *
 * @returns {boolean} True if collision detected
 *
 * Detects:
 * - Wall collision (head x/y out of bounds)
 * - Self collision (head hit own body, skip first 4 segments)
 *
 * Uses spatial hash (O(1)) with fallback brute force (O(n))
 */
```

Example Usage:

```
const collision = detector.detectCollision({
  snakeHead: { x: 10, y: 10 },
  snakeBody: [{ x: 10, y: 11 }, { x: 10, y: 12 }],
  gridWidth: 20,
  gridHeight: 20
});

if (collision) {
  console.log('COLLISION DETECTED');
}
```

`checkFoodCollision(snakeHead: Point, food: Point): boolean`

Checks if snake head collected food (exact match).

```
/**
 * Check food collision
 * @param {Point} snakeHead - Snake head position
 * @param {Point} food - Food position
 * @returns {boolean} True if head matches food position exactly
 */
```

1.5 EvolutionSystem Interface

Purpose: Manages progression through 5 evolution stages.

Methods

`checkEvolution(length: number): EvolutionStage | null`

Checks if snake should evolve based on length.

```
/**
 * Check and return evolution stage for length
 * @param {number} length - Current snake length
 * @returns {EvolutionStage|null} Evolution stage or null if not evolved
 *
 * Evolution Stages:
 * Stage 1 (Base):       length 0-49
 * Stage 2 (Growing):    length 50-99
 * Stage 3 (Aware):      length 100-149
 * Stage 4 (Empowered):  length 150-199
 * Stage 5 (Legendary):  length 200+
 *
 * Returns null if no change needed
 */
```

Example Usage:

```
const newStage = evolutionSystem.checkEvolution(150);
if (newStage &amp;amp;&amp;amp; newStage !== currentStage) {
  console.log('Evolved to:', newStage.name);
  playEvolutionSound();
}
```

`getStageProperties(stage: number): StageProperties`

Returns visual and gameplay properties for a stage.

```
/**
 * Get stage properties
 * @param {number} stage - Stage number (1-5)
 * @returns {StageProperties} Stage configuration
 *
 * @typedef {Object} StageProperties
 * @property {string} name - Stage name
 * @property {string} color - Hex color code
 * @property {number} speed - Speed in cells/second
 * @property {string} particle - Particle effect (or 'none')
 * @property {string} soundCue - Sound effect filename
 * @property {number} minLength - Minimum length for this stage
 */
```

Example Return:

```
{
  name: 'Empowered',
  color: '#FFD700',
  speed: 8,
```

```
  particle: 'trail',
  soundCue: 'whoosh.wav',
  minLength: 150
}
```

1.6 AudioManager Interface

Purpose: Manages sound effects and background music with error recovery.

Methods

`playSound(soundId: string, volume?: number): Promise<void>`

Plays a sound effect.

```
/**
 * Play sound effect
 * @param {string} soundId - Sound identifier
 * @param {number} volume - Volume 0.0-1.0 (default: 0.5)
 * @returns {Promise&lt;void&gt;} Resolves when sound queued
 *
 * Available sounds:
 * - 'food_collect' - When food eaten
 * - 'evolution_*' - Evolution stage sounds
 * - 'gameover' - Game over sound
 * - 'menu_click' - Button click
 *
 * @throws {AudioError} If audio context not initialized
 */
```

Example Usage:

```
await audioManager.playSound('food_collect', 0.7);
```

`playBGM(musicId: string, loop?: boolean): Promise<void>`

Plays background music.

```
/**
 * Play background music
 * @param {string} musicId - Music identifier
 * @param {boolean} loop - Loop music (default: true)
 * @returns {Promise&lt;void&gt;} Resolves when music started
 */
```

`setMasterVolume(volume: number): void`

Sets global volume 0.0-1.0.

```
/**
 * Set master volume
 * @param {number} volume - Volume level 0.0 (silent) to 1.0 (max)
 * @throws {RangeError} If volume outside 0.0-1.0
 */
```

`isMuted(): boolean`

Returns mute status.

```
/**
 * Check if audio is muted
 * @returns {boolean} True if muted
 */
```

1.7 StorageManager Interface

Purpose: Persists game data with checksum validation and recovery.

Methods

`saveHighScore(score: number, stage: number): Promise&lt;boolean&gt;`

Saves a high score to persistent storage.

```
/**
 * Save high score
 * @param {number} score - Score to save
 * @param {number} stage - Final stage reached
 * @returns {Promise&amp;lt;boolean&amp;gt;} True if saved successfully
 *
 * Storage format (localStorage):
 * Key: 'snakeEvolution_leaderboard'
 * Value: JSON array of scores (max 10)
 *
 * Each entry has:
 * - score (number)
 * - stage (1-5)
 * - timestamp (ISO string)
 * - checksum (CRC32)
 */
```

Example Usage:

```
const saved = await storageManager.saveHighScore(250, 5);
if (saved) {
  console.log('Score saved to leaderboard!');
}
```

`getLeaderboard(): HighScoreEntry[]`

Retrieves top 10 scores with validation.

```
/**
 * Get leaderboard
 * @returns {HighScoreEntry[]} Array of top scores (max 10)
 *
 * @typedef {Object} HighScoreEntry
 * @property {number} rank - Position 1-10
 * @property {number} score - Score value
 * @property {number} stage - Final stage
 * @property {string} timestamp - Date achieved
 *
 * Validates checksums and recovers from corruption
 */
```

Example Return:

```
[
  { rank: 1, score: 500, stage: 5, timestamp: '2025-11-06T14:30:00Z' },
  { rank: 2, score: 425, stage: 4, timestamp: '2025-11-06T12:15:00Z' }
]
```

`clearLeaderboard(): Promise&lt;boolean&gt;`

Completely clears stored leaderboard.

```
/**
 * Clear all saved scores
 * @returns {Promise&lt;boolean&gt;} True if cleared
 *
 * ⚠ WARNING: This is irreversible
 */
```

1.8 Logger Interface

Purpose: Records game events with multiple output handlers.

Methods

`log(level: LogLevel, message: string, data?: object): void`

Logs a message at specified level.

```
/**
 * Log message
 * @param {LogLevel} level - Log level: 'DEBUG'|'INFO'|'WARN'|'ERROR'|'CRITICAL'
 * @param {string} message - Log message
 * @param {object} data - Optional structured data
 *
 * Outputs to:
 * - Browser console (colorized)
 * - localStorage (rotating buffer)
 * - Remote handler (if configured, v2)
 */
```

Example Usage:

```
Logger.log('INFO', 'Game started', { score: 0, stage: 1 });
Logger.log('ERROR', 'Save failed', { reason: 'quota exceeded' });
```

`debug(message: string, data?: object): void`

`info(message: string, data?: object): void`

`warn(message: string, data?: object): void`

`error(message: string, data?: object): void`

`critical(message: string, data?: object): void`

Convenience methods for each log level.

```
Logger.debug('Collision detected');
Logger.info('Player evolved');
Logger.warn('FPS dropped below 30');
Logger.error('Audio context failed', { error: err });
Logger.critical('Game crash detected');
```

`getLogs(filter?: LogFilter): LogEntry[]`

Retrieves logs matching optional filter.

```
/**
 * Get logs
 * @param {LogFilter} filter - Optional filter criteria
 * @param {string} filter.level - Min log level
 * @param {number} filter.since - Milliseconds from now
 * @returns {LogEntry[]} Array of log entries
 */
```

2. Type Definitions

2.1 Common Types

```
/**
 * 2D Point/Vector
 * @typedef {Object} Point
 * @property {number} x - X coordinate
 * @property {number} y - Y coordinate
 */

/**
 * Game State Enum
 * @typedef {'MENU'|'INIT'|'PLAYER_INPUT'|'PLAYING'|'PAUSED'|'GAMEOVER'} GameState
 */

/**
 * Direction Enum
 * @typedef {'UP'|'DOWN'|'LEFT'|'RIGHT'} Direction
 */

/**
 * Evolution Stage
 * @typedef {'Base'|'Growing'|'Aware'|'Empowered'|'Legendary'} EvolutionStage
 */

/**
 * Log Level
 * @typedef {'DEBUG'|'INFO'|'WARN'|'ERROR'|'CRITICAL'} LogLevel
 */

/**
 * Game Configuration
 * @typedef {Object} GameConfig
 * @property {number} gridWidth - Width in cells (default: 20)
 * @property {number} gridHeight - Height in cells (default: 20)
 * @property {number} initialSpeed - Speed cells/sec (default: 5)
 * @property {boolean} soundEnabled - Enable audio (default: true)
 * @property {string} renderTargetId - Canvas element ID
 * @property {number} FPSTarget - Target FPS (default: 60)
```

```
 */

/**
 * Performance Metrics
 * @typedef {Object} PerformanceMetrics
 * @property {number} avgFPS - Average FPS
 * @property {number} p95FrameTime - 95th percentile frame time (ms)
 * @property {number} p99FrameTime - 99th percentile frame time (ms)
 * @property {number} collisionTimeAvg - Average collision detection time (ms)
 * @property {number} renderTimeAvg - Average render time (ms)
 * @property {number} memoryUsage - Memory in MB
 */
```

3. Event Bus Interface

3.1 EventBus

Purpose: Pub/Sub event system for loose coupling between modules.

Methods

`subscribe(eventType: string, handler: (data: any) => void): () => void`

Subscribes to an event.

```
/**
 * Subscribe to event
 * @param {string} eventType - Event type identifier
 * @param {Function} handler - Callback function
 * @returns {Function} Unsubscribe function
 *
 * Standard events:
 * - 'game:start' - Game started
 * - 'game:pause' - Game paused
 * - 'game:resume' - Game resumed
 * - 'game:over' - Game ended
 * - 'food:collected' - Food eaten
 * - 'snake:evolved' - Snake evolved
 * - 'collision:detected' - Collision occurred
 * - 'error:fatal' - Fatal error occurred
 */
```

Example Usage:

```
const unsub = eventBus.subscribe('snake:evolved', (data) =&gt; {
  console.log('Evolved to:', data.stage.name);
});

// Unsubscribe later
unsub();
```

`publish(eventType: string, data?: any): void`

Publishes an event to all subscribers.

```
/**
 * Publish event
 * @param {string} eventType - Event type identifier
 * @param {*} data - Optional event data
 */
```

4. Error Handling

4.1 Error Types

All errors inherit from `GameError` base class.

```
/**
 * Base error class
 */
class GameError extends Error {
  constructor(message, code, context) {
    this.message = message;
    this.code = code;     // Machine-readable code
    this.context = context;  // Debug context
    this.timestamp = Date.now();
  }
}

/**
 * Game state invalid for operation
 */
class GameStateError extends GameError {
  code = 'ERR_INVALID_STATE';
}

/**
 * Initialization failed
 */
class GameInitializationError extends GameError {
  code = 'ERR_INIT_FAILED';
}

/**
 * Collision detection error
 */
class CollisionError extends GameError {
  code = 'ERR_COLLISION';
}

/**
 * Storage/persistence error
```

```
 */
class StorageError extends GameError {
  code = 'ERR_STORAGE';
}

/**
 * Audio system error
 */
class AudioError extends GameError {
  code = 'ERR_AUDIO';
}
```

4.2 Error Handling Pattern

```
try {
  await engine.start();
} catch (error) {
  if (error instanceof GameStateError) {
    console.log('Game state invalid:', error.message);
  } else if (error instanceof GameInitializationError) {
    console.log('Init failed:', error.context);
  } else {
    Logger.error('Unexpected error', { error });
  }
}
```

5. Integration Examples

5.1 Complete Game Initialization

```
// 1. Create engine
const engine = new GameEngine();
const stateManager = engine.getStateManager();
const inputManager = engine.getInputManager();

// 2. Initialize
await engine.initialize({
  gridWidth: 20,
  gridHeight: 20,
  soundEnabled: true,
  renderTargetId: 'gameCanvas'
});

// 3. Subscribe to events
eventBus.subscribe('game:over', (data) =&gt; {
  console.log('Final score:', data.score);
  showGameOverScreen();
});

// 4. Setup input handling
document.addEventListener('keydown', (e) =&gt; {
```

```
  const direction = mapKeyToDirection(e.key);
  if (direction) {
    inputManager.processInput(direction);
  }
});

// 5. Start game
await engine.start();
```

5.2 Extending with Custom Logic

```
// Subscribe to evolution events
eventBus.subscribe('snake:evolved', async (data) =&gt; {
  const properties = evolutionSystem.getStageProperties(data.newStage);

  // Play animation
  await renderEvolutionAnimation(properties);

  // Play sound
  await audioManager.playSound(properties.soundCue);

  // Update HUD
  updateHUDDisplay(data.newStage);
});
```

6. Performance Considerations

6.1 API Call Frequency

API	Recommended Frequency
`getGameState()`	Every frame (60Hz)
`detectCollision()`	Every frame (60Hz)
`processInput()`	On key/touch event
`saveHighScore()`	On game over
`getLeaderboard()`	On menu open
`log()`	Trace/debug only

6.2 Memory Allocation

- ✅ Reuse Point objects (no allocation per frame)
- ✅ Use object pooling for collision contexts
- ❌ Avoid large data copies in hot paths
- ❌ Don't store full game state snapshots frequently

7. Document Information

Item	Details
Document Type	API Reference
Version	1.0
Date Published	November 2025
Standard	OpenAPI 3.0 adapted for client-side
Language	English

Document Version History

Version	Date	Changes
1.0	2025-11-06	Initial complete API reference documentation

End of API Reference

For implementation details, see Technical Analysis v2.1.
For user documentation, see User Guide v1.0.
For requirements, see PRD v2.1.
[1] [2]

⁂

1. PRD-v2-1-EN.pdf
2. Functional-Analysis-EN.pdf