Documento di Analisi Tecnica - Snake Evolution

Tecnologie, Strutture Dati e Implementazione Dettagliata

Redatto da: Senior Software Architect
Data: Novembre 2025
Versione: 1.0
Status: Approvato per lo Sviluppo

Executive Summary

Il presente documento fornisce un'analisi tecnica completa del progetto **Snake Evolution**, dettagliando le tecnologie adottate, le strutture dati necessarie, i pattern di implementazione, e le specifiche di integrazione tra componenti. Questo documento funge da blueprint tecnico per il team di sviluppo, fornendo una definizione precisa di ogni struttura dati, algoritmo critico e tecnologia selezionata.

1. Stack Tecnologico Dettagliato

1.1 Selezione Tecnologie e Giustificazione

Frontend Framework & Rendering

Tecnologia	Versione	Utilizzo	Giustificazione
HTML5	ES2020+	Struttura documento	Standard web consolidato, nativamente supportato
CSS3	Latest	Styling UI	Flexbox/Grid per layout responsivo
JavaScript (Vanilla)	ES6+	Logica applicativa	Pieno controllo, nessuna dipendenza esterna
Canvas 2D API	HTML5	Rendering grafica	Performance elevate, supporto cross-browser
Web Audio API	W3C Standard	Audio playback	Gestione audio nativa, bassa latenza

Motivazione Architetturale: L'uso di **Vanilla JavaScript** invece di framework come React o Vue garantisce:

- Riduzione delle dipendenze (file size < 15MB)
- Controllo totale sulla performance
- Curva di apprendimento ridotta per team piccoli
- Maggiore flessibilità per ottimizzazioni game-specific

Build Tools & Dev Environment

Tool	Scopo	Alternativa Scartata
Webpack/Parcel	Bundling e ottimizzazione	Rollup (meno versatile)
Babel	Transpiling ES6 a ES5 (compatibilità)	SWC (non maturo)
npm/yarn	Package management	pnpm (curva apprendimento)
Jest	Unit testing	Mocha (setup più complesso)
Cypress	Integration testing	Selenium (lento e verboso)

Performance & Optimization Tools

- **Lighthouse**: Audit performance, accessibility, SEO
- **Chrome DevTools**: Profiling, debugging, FPS measurement
- **WebPageTest**: Load time analysis cross-device
- **ImageOptim/TinyPNG**: Asset compression

1.2 Dipendenze Esterne Minime

```
{
  "dependencies": {
    "crypto-js": "^4.1.1"  // Per checksum MD5 salvataggio
  },
  "devDependencies": {
    "webpack": "^5.88.0",
    "babel-loader": "^9.1.3",
    "jest": "^29.7.0",
    "cypress": "^13.6.1",
    "eslint": "^8.50.0",
    "prettier": "^3.0.3"
  }
}
```

Razionale: Mantenere dipendenze al minimo riduce:

- Vulnerabilità di sicurezza
- Complexity di build
- Tempo di caricamento
- Necessità di aggiornamenti

2. Strutture Dati Fondamentali

2.1 Entità Core - Definizioni TypeScript

```
/**
 * Rappresentazione della posizione nel gioco
 * Usata per serpente, cibo, grid cells
 */
interface Point {
  x: number;  // [0-19] per griglia 20x20
  y: number;  // [0-19] per griglia 20x20
}

/**
 * Segmento del serpente
 * Mantiene posizione e proprietà visive
 */
interface SnakeSegment extends Point {
  segmentId: string;           // UUID per identificazione unica
  isHead: boolean;             // Discriminatore per testa vs corpo
  createdAt: number;           // Timestamp di creazione (per animazioni)
  color?: string;              // Override colore (per gradients futuri)
}

/**
 * Stato completo del serpente
 * Struttura ottimizzata per accesso veloce
 */
interface SnakeState {
  segments: SnakeSegment[];    // Array ordinato [testa, ..., coda]
  direction: Direction;        // Direzione corrente
  nextDirection: Direction;    // Input bufferato per prossimo tick
  length: number;              // Cache della lunghezza (O(1) lookup)
  isAlive: boolean;            // Flag per collisioni
  velocityMultiplier: number;  // Applicato dall'EvolutionSystem
}

/**
 * Enum per direzioni (memory efficient)
 */
enum Direction {
  UP = "UP",
  DOWN = "DOWN",
  LEFT = "LEFT",
  RIGHT = "RIGHT"
}

/**
 * Rappresentazione del cibo
 */
interface Food extends Point {
  foodId: string;              // UUID per tracking
  type: FoodType;              // Type: standard, special (per v2)
  spawnedAt: number;           // Timestamp spawn
  isActive: boolean;           // Flag per raccolta
```

```
}

enum FoodType {
  STANDARD = "standard",
  BONUS = "bonus"              // Riservato per future iterazioni
}

/**
 * Cella della griglia
 * Utilizzata per collision detection e spatial hashing
 */
interface GridCell extends Point {
  occupancy: CellOccupancy;    // Chi occupa questa cella
  renderData?: RenderData;     // Cache per rendering
}

enum CellOccupancy {
  EMPTY = 0,
  SNAKE_HEAD = 1,
  SNAKE_BODY = 2,
  FOOD = 3,
  WALL = 4
}

/**
 * Dati di rendering per ottimizzazione dirty rectangle
 */
interface RenderData {
  isDirty: boolean;            // Flag per necessità di ridisegno
  lastRenderedColor: string;   // Colore ultimo frame
  particlesFX?: ParticleEffect[];
}
```

2.2 Evolution System - Strutture Dati

```
/**
 * Definizione di uno stadio evolutivo
 * Immutabile - caricato da configurazione
 */
interface EvolutionStage {
  stageId: number;                  // [0-4]
  name: string;                     // "Base", "Crescente", etc.
  lengthRange: [number, number];    // [min, max]
  properties: StageProperties;
  effects: StageEffects;
  audio: StageAudio;
}

interface StageProperties {
  color: string;                    // Colore HEX (#RRGGBB)
  speedMultiplier: number;          // 1.0 = base, 1.05 = +5%
  texturePattern: TextureType;      // Per future rendering avanzati
}

interface StageEffects {
```

```
  particleType: ParticleType;       // "none", "trail_light", "trail_glow", etc.
  particleIntensity: number;        // [0-1] densità particelle
  glowIntensity: number;            // [0-1] intensità aura
  animationDuration: number;        // Milliseconda transizione
}

interface StageAudio {
  triggerSound: string;             // Nome file audio
  volume: number;                   // [0-1]
  pitch?: number;                   // Variazione pitch (future)
}

enum TextureType {
  SOLID = "solid",
  PATTERN = "pattern",
  GRADIENT = "gradient"
}

enum ParticleType {
  NONE = "none",
  TRAIL_LIGHT = "trail_light",
  TRAIL_GLOW = "trail_glow",
  EXPLOSION = "explosion",
  AURA_GLITCH = "aura_glitch"
}

/**
 * Stato progression/evoluzione
 */
interface EvolutionState {
  currentStage: EvolutionStage;
  previousStage: EvolutionStage | null;
  stageChangeTime: number;          // Timestamp ultimo cambio
  totalEvolutions: number;          // Contatore per analytics
  isTransitioning: boolean;         // Flag durante animazione
}
```

2.3 Game State Globale

```
/**
 * Stato globale del gioco
 * Struttura centralizzata per todo il game state
 */
interface GameState {
  gamePhase: GamePhase;             // MENU, PLAYING, PAUSED, GAMEOVER
  snake: SnakeState;
  food: Food;
  grid: GridCell[][];               // 20x20 spatial structure
  score: number;
  sessionStartTime: number;

  // Evoluzione
  evolutionState: EvolutionState;

  // Rendering cache
```

```
  dirtyRectangles: Rect[];          // Per ottimizzazione rendering
  cameraViewport: Viewport;

  // Audio state
  audioEnabled: boolean;
  masterVolume: number;

  // Stats
  foodEaten: number;
  maxSnakeLength: number;
  tickCounter: number;              // Per debugging
}

enum GamePhase {
  MENU = "MENU",
  PLAYING = "PLAYING",
  PAUSED = "PAUSED",
  GAMEOVER = "GAMEOVER",
  LEADERBOARD = "LEADERBOARD"
}

interface Viewport {
  x: number;
  y: number;
  width: number;
  height: number;
}
```

2.4 Input & Event System

```
/**
 * Input event normalizzato
 */
interface InputEvent {
  inputId: string;
  timestamp: number;
  deviceType: "keyboard" | "touch";
  direction: Direction;
  isValid: boolean;
}

/**
 * Input queue per buffering
 */
interface InputBuffer {
  queue: InputEvent[];
  maxSize: number;          // 5 events

  enqueue(event: InputEvent): void;
  dequeue(): InputEvent | null;
  peek(): InputEvent | null;
  clear(): void;
}

/**
```

```
 * Event bus - sistema pub/sub per loose coupling
 */
interface GameEvent {
  eventType: GameEventType;
  timestamp: number;
  data: any;
}

enum GameEventType {
  SNAKE_MOVED = "SNAKE_MOVED",
  FOOD_EATEN = "FOOD_EATEN",
  EVOLUTION_TRIGGERED = "EVOLUTION_TRIGGERED",
  COLLISION_DETECTED = "COLLISION_DETECTED",
  GAME_OVER = "GAME_OVER",
  SCORE_UPDATED = "SCORE_UPDATED",
  PAUSE_TOGGLED = "PAUSE_TOGGLED"
}
```

2.5 Persistence Layer

```
/**
 * High Score Entry persistente
 */
interface HighScoreEntry {
  id: string;                       // score_&lt;timestamp&gt;
  playerName: string;               // [1-20 caratteri]
  score: number;
  evolutionStageReached: number;    // 0-4
  sessionDuration: number;          // Secondi
  timestamp: string;                // ISO8601
  checksum: string;                 // MD5 per validazione
}

/**
 * Container per storage
 */
interface StorageData {
  version: string;                  // "1.0"
  dataVersion: number;              // Per migration future
  scores: HighScoreEntry[];         // Top 10 + current
  settings: UserSettings;
  metadata: StorageMetadata;
}

interface UserSettings {
  audioEnabled: boolean;
  masterVolume: number;             // [0-1]
  lastPlayedDate: string;           // ISO8601
}

interface StorageMetadata {
  totalSessions: number;
  createdAt: string;
  lastUpdatedAt: string;
  checksumValid: boolean;
```

```
}

/**
 * Chiavi localStorage
 */
const STORAGE_KEYS = {
  SCORES: "snakeEvolution_scores",
  SETTINGS: "snakeEvolution_settings",
  METADATA: "snakeEvolution_metadata",
  VERSION: "snakeEvolution_version"
} as const;
```

3. Algoritmi Critici e Implementazione

3.1 Collision Detection Algorithm

```
/**
 * Collision Detection - O(1) spatially optimized
 * Utilizza griglia per ridurre complessità
 */
class CollisionDetector {
  private grid: GridCell[][];
  private gridSize: number = 20;
  private cellSize: number = 25;

  /**
   * Controlla collisioni del serpente
   * @returns true se collisione rilevata
   */
  checkSnakeCollision(snake: SnakeState): boolean {
    const head = snake.segments[^0];

    // Controllo limiti (wall collision)
    if (head.x &lt; 0 || head.x &gt;= this.gridSize ||
        head.y &lt; 0 || head.y &gt;= this.gridSize) {
      return true;
    }

    // Controllo self-collision
    // Itera da segmento 4 in poi (primi 3-4 non possono collide)
    for (let i = 4; i &lt; snake.segments.length; i++) {
      const segment = snake.segments[i];
      if (head.x === segment.x &amp;&amp; head.y === segment.y) {
        return true;
      }
    }

    return false;
  }

  /**
   * Controlla raccolta cibo
   * @returns true se cibo toccato
```

```
   */
  checkFoodCollision(snakeHead: Point, food: Food): boolean {
    return snakeHead.x === food.x &amp;&amp; snakeHead.y === food.y;
  }

  /**
   * Aggiorna griglia spatiale
   * Chiamato una volta per frame per tracking occupancy
   */
  updateGridOccupancy(snake: SnakeState, food: Food): void {
    // Clear grid
    this.grid = Array(this.gridSize)
      .fill(null)
      .map(() =&gt; Array(this.gridSize)
        .fill(null)
        .map((_, idx) =&gt; ({
          x: idx,
          y: idx,
          occupancy: CellOccupancy.EMPTY
        })));

    // Mark snake
    snake.segments.forEach((seg, idx) =&gt; {
      const occupancy = idx === 0 ?
        CellOccupancy.SNAKE_HEAD :
        CellOccupancy.SNAKE_BODY;
      this.grid[seg.y][seg.x].occupancy = occupancy;
    });

    // Mark food
    this.grid[food.y][food.x].occupancy = CellOccupancy.FOOD;
  }

  /**
   * Get cella occupancy - utility
   */
  getCellOccupancy(x: number, y: number): CellOccupancy {
    if (x &lt; 0 || x &gt;= this.gridSize || y &lt; 0 || y &gt;= this.gridSize) {
      return CellOccupancy.WALL;
    }
    return this.grid[y][x].occupancy;
  }
}
```

Analisi Complessità:

- **Time**: O(n) dove n = lunghezza serpente (per self-collision check)
- **Space**: O(400) = O(1) per griglia 20x20
- **Optimizzazione**: Per serpenti lunghi (> 50 segmenti), usare grid-based spatial hashing

3.2 Snake Movement Algorithm

```
/**
 * Movimento serpente - physics simulation
 */
class SnakeController {
  private moveTickRate: number = 100; // ms tra movimenti
  private lastMoveTime: number = 0;

  /**
   * Aggiorna posizione serpente
   * Responsabile della crescita e movimento
   */
  updateSnake(
    snake: SnakeState,
    direction: Direction,
    shouldGrow: boolean = false,
    velocityMult: number = 1.0
  ): void {
    // Calcola nuovo head position
    const head = snake.segments[^0];
    const newHead = this.calculateNewHeadPosition(head, direction);

    // Prepend new head
    snake.segments.unshift({
      ...newHead,
      segmentId: this.generateId(),
      isHead: true,
      createdAt: Date.now(),
      color: undefined  // Usa colore dello stadio
    });

    // Aggiorna head flag di vecchio head
    snake.segments[^1].isHead = false;

    // Pop tail se non growing
    if (!shouldGrow) {
      snake.segments.pop();
    }

    // Update cache
    snake.length = snake.segments.length;
    snake.direction = direction;
  }

  /**
   * Calcola nuova posizione head
   * @returns Nuovo punto
   */
  private calculateNewHeadPosition(head: Point, direction: Direction): Point {
    const moves = {
      UP: { dx: 0, dy: -1 },
      DOWN: { dx: 0, dy: 1 },
      LEFT: { dx: -1, dy: 0 },
      RIGHT: { dx: 1, dy: 0 }
    };
```

```
    const { dx, dy } = moves[direction];
    return {
      x: head.x + dx,
      y: head.y + dy
    };
  }

  private generateId(): string {
    return `seg_${Date.now()}_${Math.random()}`;
  }
}
```

3.3 Evolution Progression Algorithm

```
/**
 * Logica di evoluzione basata su milestone
 */
class EvolutionSystem {
  private stages: EvolutionStage[] = [];
  private stageConfig: Map&lt;number, EvolutionStage&gt; = new Map();

  constructor() {
    this.initializeStages();
  }

  /**
   * Controlla se serpente ha raggiunto nuovo stadio
   * Chiamato ogni volta che lunghezza cambia
   */
  checkAndApplyEvolution(
    snakeLength: number,
    currentStage: EvolutionState
  ): EvolutionState | null {
    // Trova stadio corrispondente a lunghezza
    const newStageId = this.getStageByLength(snakeLength);

    if (newStageId !== currentStage.currentStage.stageId) {
      // Evoluzione richiesta
      const newStage = this.stageConfig.get(newStageId)!;

      return {
        currentStage: newStage,
        previousStage: currentStage.currentStage,
        stageChangeTime: Date.now(),
        totalEvolutions: currentStage.totalEvolutions + 1,
        isTransitioning: true
      };
    }

    return null;
  }

  /**
   * Mappa lunghezza a stadio
```

```
   */
  private getStageByLength(length: number): number {
    if (length &lt; 11) return 0;      // Base
    if (length &lt; 26) return 1;      // Crescente
    if (length &lt; 51) return 2;      // Consapevole
    if (length &lt; 76) return 3;      // Potenziato
    return 4;                       // Leggendario
  }

  /**
   * Applica proprietà stadio al serpente
   */
  applyStageEffects(snake: SnakeState, stage: EvolutionStage): void {
    snake.velocityMultiplier = stage.properties.speedMultiplier;
    // Color applicato nel renderer
  }

  private initializeStages(): void {
    this.stageConfig.set(0, {
      stageId: 0,
      name: "Base",
      lengthRange: [0, 10],
      properties: {
        color: "#00AA00",
        speedMultiplier: 1.0,
        texturePattern: TextureType.SOLID
      },
      effects: {
        particleType: ParticleType.NONE,
        particleIntensity: 0,
        glowIntensity: 0,
        animationDuration: 0
      },
      audio: {
        triggerSound: "none",
        volume: 0
      }
    });

    // ... altri stadi (simile) ...
  }
}
```

3.4 Food Spawn Algorithm

```
/**
 * Spawn del cibo - random placement ottimizzato
 */
class FoodSpawner {
  private gridSize: number = 20;
  private spawnChance: number = 0.05;  // 5% cells libere
  private spawnInterval: number = 3000; // 3 secondi min tra spawn
  private lastSpawnTime: number = 0;

  /**
```

```
   * Cerca di spawnare cibo
   */
  trySpawnFood(
    currentFood: Food | null,
    snakeSegments: SnakeSegment[]
  ): Food | null {
    // Se cibo già esiste, non spawnare
    if (currentFood &amp;amp;&amp;amp; currentFood.isActive) {
      return null;
    }

    // Controlla intervallo tempo
    if (Date.now() - this.lastSpawnTime &amp;lt; this.spawnInterval) {
      return null;
    }

    // Calcola celle occupate
    const occupiedCells = new Set&amp;lt;string&amp;gt;();
    snakeSegments.forEach(seg =&amp;gt; {
      occupiedCells.add(`${seg.x},${seg.y}`);
    });

    // Trova celle libere
    const freeCells: Point[] = [];
    for (let x = 0; x &amp;lt; this.gridSize; x++) {
      for (let y = 0; y &amp;lt; this.gridSize; y++) {
        if (!occupiedCells.has(`${x},${y}`)) {
          freeCells.push({ x, y });
        }
      }
    }

    // Applica spawn chance
    if (Math.random() &amp;gt; this.spawnChance) {
      return null;
    }

    // Seleziona posizione random
    if (freeCells.length === 0) {
      return null; // Nessuna cella libera
    }

    const randomCell = freeCells[
      Math.floor(Math.random() * freeCells.length)
    ];

    const newFood: Food = {
      ...randomCell,
      foodId: `food_${Date.now()}`,
      type: FoodType.STANDARD,
      spawnedAt: Date.now(),
      isActive: true
    };

    this.lastSpawnTime = Date.now();
    return newFood;
```

```
  }
}
```

4. Architecture Patterns & Design

4.1 Event-Driven Architecture

```
/**
 * Event Bus - Mediator pattern per decoupling
 */
class EventBus {
  private subscribers: Map&lt;GameEventType, Function[]&gt; = new Map();

  /**
   * Subscribe a evento
   */
  subscribe(eventType: GameEventType, callback: (event: GameEvent) =&gt; void): Unsubscri
    if (!this.subscribers.has(eventType)) {
      this.subscribers.set(eventType, []);
    }

    this.subscribers.get(eventType)!.push(callback);

    // Restituisci unsubscribe function
    return () =&gt; {
      const handlers = this.subscribers.get(eventType)!;
      const idx = handlers.indexOf(callback);
      if (idx &gt; -1) handlers.splice(idx, 1);
    };
  }

  /**
   * Emit evento
   */
  emit(eventType: GameEventType, data?: any): void {
    const handlers = this.subscribers.get(eventType) || [];
    const event: GameEvent = {
      eventType,
      timestamp: Date.now(),
      data
    };

    handlers.forEach(handler =&gt; {
      try {
        handler(event);
      } catch (error) {
        Logger.error("Event handler error", { eventType, error });
      }
    });
  }

  /**
   * Clear handlers (cleanup)
```

```
   */
  clear(): void {
    this.subscribers.clear();
  }
}

type UnsubscribeFn = () =&gt; void;
```

4.2 State Management Pattern

```
/**
 * Store pattern - Single source of truth per game state
 */
class GameStore {
  private state: GameState;
  private subscribers: Set&lt;(state: GameState) =&gt; void&gt; = new Set();
  private history: GameState[] = [];
  private maxHistorySize: number = 100;

  constructor(initialState: GameState) {
    this.state = initialState;
  }

  /**
   * Getter - stato immutabile
   */
  getState(): Readonly&lt;GameState&gt; {
    return Object.freeze({ ...this.state });
  }

  /**
   * Reducer pattern
   */
  dispatch(action: GameAction): void {
    const newState = this.reduce(this.state, action);

    if (newState !== this.state) {
      this.history.push(this.state);
      if (this.history.length &gt; this.maxHistorySize) {
        this.history.shift();
      }

      this.state = newState;
      this.notifySubscribers();
    }
  }

  /**
   * Reducer function
   */
  private reduce(state: GameState, action: GameAction): GameState {
    switch (action.type) {
      case "UPDATE_SNAKE":
        return { ...state, snake: action.payload };
      case "UPDATE_SCORE":
```

```
        return { ...state, score: action.payload };
      case "SET_PHASE":
        return { ...state, gamePhase: action.payload };
      default:
        return state;
    }
  }

  /**
   * Subscribe a state changes
   */
  subscribe(callback: (state: GameState) =&gt; void): UnsubscribeFn {
    this.subscribers.add(callback);
    return () =&gt; this.subscribers.delete(callback);
  }

  private notifySubscribers(): void {
    this.subscribers.forEach(cb =&gt; cb(this.getState()));
  }

  /**
   * Undo per debugging
   */
  undo(): void {
    if (this.history.length &gt; 0) {
      this.state = this.history.pop()!;
      this.notifySubscribers();
    }
  }
}

interface GameAction {
  type: string;
  payload?: any;
}
```

4.3 Object Pooling per Performance

```
/**
 * Object Pool - Riutilizzare oggetti per evitare GC pressure
 */
class ObjectPool&lt;T&gt; {
  private available: T[] = [];
  private inUse: Set&lt;T&gt; = new Set();
  private factory: () =&gt; T;
  private maxSize: number;

  constructor(factory: () =&gt; T, initialSize: number = 100) {
    this.factory = factory;
    this.maxSize = initialSize;

    for (let i = 0; i &lt; initialSize; i++) {
      this.available.push(factory());
    }
  }
```

```
  /**
   * Acquire oggetto dal pool
   */
  acquire(): T {
    let obj;
    if (this.available.length &gt; 0) {
      obj = this.available.pop()!;
    } else {
      obj = this.factory();
    }

    this.inUse.add(obj);
    return obj;
  }

  /**
   * Release oggetto al pool
   */
  release(obj: T): void {
    this.inUse.delete(obj);

    if (this.available.length &lt; this.maxSize) {
      this.available.push(obj);
    }
  }

  /**
   * Clear pool
   */
  clear(): void {
    this.available = [];
    this.inUse.clear();
  }

  getStats() {
    return {
      available: this.available.length,
      inUse: this.inUse.size,
      total: this.available.length + this.inUse.size
    };
  }
}

// Utilizzo per particelle
const particlePool = new ObjectPool(
  () =&gt; ({
    x: 0, y: 0, vx: 0, vy: 0, life: 0, maxLife: 0
  }),
  500
);
```

5. Performance Optimization Strategies

5.1 Rendering Optimization

```
/**
 * Dirty Rectangle Rendering
 * Redraw only changed areas
 */
class OptimizedCanvas2DRenderer {
  private canvas: HTMLCanvasElement;
  private ctx: CanvasRenderingContext2D;
  private dirtyRectangles: Rect[] = [];
  private lastFrameState: string = "";

  constructor(canvas: HTMLCanvasElement) {
    this.canvas = canvas;
    this.ctx = canvas.getContext("2d")!;
  }

  /**
   * Render frame con ottimizzazione dirty rectangles
   */
  render(state: GameState): void {
    const currentStateHash = this.hashState(state);

    if (currentStateHash === this.lastFrameState) {
      return; // Nessun cambiamento, skip render
    }

    // Full clear (meno costoso di ridisegnare parziale in Canvas 2D)
    this.ctx.fillStyle = "#000000";
    this.ctx.fillRect(0, 0, this.canvas.width, this.canvas.height);

    // Disegna griglia
    this.drawGrid();

    // Disegna cibo
    this.drawFood(state.food);

    // Disegna serpente
    this.drawSnake(state.snake);

    // Disegna UI overlay
    this.drawHUD(state.score, state.evolutionState);

    // Disegna effetti particellari
    state.dirtyRectangles.forEach(rect =&gt; {
      this.drawParticles(rect);
    });

    this.lastFrameState = currentStateHash;
  }

  private hashState(state: GameState): string {
    // Hash semplice dello stato (potrebbe usare hash algorithm migliore)
```

```
    return JSON.stringify({
      snakePos: state.snake.segments[^0],
      snakeLen: state.snake.length,
      foodPos: state.food,
      score: state.score,
      stage: state.evolutionState.currentStage.stageId
    });
  }

  private drawSnake(snake: SnakeState): void {
    const headColor = this.getStageColor(snake.velocityMultiplier);

    snake.segments.forEach((seg, idx) =&gt; {
      const cellSize = 25;
      const x = seg.x * cellSize;
      const y = seg.y * cellSize;

      // Draw rectangle
      this.ctx.fillStyle = idx === 0 ? headColor : this.darkenColor(headColor);
      this.ctx.fillRect(x + 1, y + 1, cellSize - 2, cellSize - 2);

      // Draw border
      this.ctx.strokeStyle = "#FFFFFF";
      this.ctx.lineWidth = 1;
      this.ctx.strokeRect(x + 1, y + 1, cellSize - 2, cellSize - 2);
    });
  }

  private drawFood(food: Food): void {
    const cellSize = 25;
    const x = food.x * cellSize;
    const y = food.y * cellSize;

    this.ctx.fillStyle = "#FF0000";
    this.ctx.beginPath();
    this.ctx.arc(x + cellSize/2, y + cellSize/2, 8, 0, Math.PI * 2);
    this.ctx.fill();
  }

  private drawGrid(): void {
    const cellSize = 25;
    const gridSize = 20;

    this.ctx.strokeStyle = "#333333";
    this.ctx.lineWidth = 0.5;

    for (let i = 0; i &lt;= gridSize; i++) {
      // Linee orizzontali
      this.ctx.beginPath();
      this.ctx.moveTo(0, i * cellSize);
      this.ctx.lineTo(gridSize * cellSize, i * cellSize);
      this.ctx.stroke();

      // Linee verticali
      this.ctx.beginPath();
      this.ctx.moveTo(i * cellSize, 0);
```

```
      this.ctx.lineTo(i * cellSize, gridSize * cellSize);
      this.ctx.stroke();
    }
  }

  private drawHUD(score: number, evolution: EvolutionState): void {
    this.ctx.fillStyle = "#FFFFFF";
    this.ctx.font = "14px Arial";
    this.ctx.fillText(`Score: ${score}`, 10, this.canvas.height - 10);
    this.ctx.fillText(`Stage: ${evolution.currentStage.name}`, 150, this.canvas.height -
  }

  private drawParticles(rect: Rect): void {
    // Rendering particelle da texture/sprite
  }

  private getStageColor(speedMult: number): string {
    if (speedMult &gt;= 1.20) return "#FFD700"; // Leggendario
    if (speedMult &gt;= 1.15) return "#FF4400"; // Potenziato
    if (speedMult &gt;= 1.10) return "#00DD00"; // Consapevole
    if (speedMult &gt;= 1.05) return "#0088FF"; // Crescente
    return "#00AA00"; // Base
  }

  private darkenColor(hex: string): string {
    const num = parseInt(hex.slice(1), 16);
    const amt = Math.round(2.55 * -30);
    const R = (num &gt;&gt; 16) + amt;
    const G = (num &gt;&gt; 8 &amp; 0x00FF) + amt;
    const B = (num &amp; 0x0000FF) + amt;
    return "#" + (0x1000000 + (R&lt;255?R&lt;1?0:R:255)*0x10000 +
      (G&lt;255?G&lt;1?0:G:255)*0x100 +
      (B&lt;255?B&lt;1?0:B:255))
      .toString(16).slice(1);
  }
}

interface Rect {
  x: number;
  y: number;
  width: number;
  height: number;
}
```

5.2 Input Buffering

```
/**
 * Input Queue per smooth input handling
 * Evita loss di comando tra frame
 */
class InputQueue {
  private queue: Direction[] = [];
  private maxQueueSize: number = 5;

  /**
```

```
   * Enqueue input
   */
  enqueue(direction: Direction): void {
    // Evita duplicati consecutivi
    if (this.queue[this.queue.length - 1] === direction) {
      return;
    }

    if (this.queue.length &lt; this.maxQueueSize) {
      this.queue.push(direction);
    } else {
      this.queue.shift();
      this.queue.push(direction);
    }
  }

  /**
   * Dequeue input
   */
  dequeue(): Direction | null {
    return this.queue.shift() || null;
  }

  /**
   * Peek senza rimuovere
   */
  peek(): Direction | null {
    return this.queue[^0] || null;
  }

  clear(): void {
    this.queue = [];
  }

  size(): number {
    return this.queue.length;
  }
}
```

6. Network & Storage Implementation

6.1 LocalStorage Persistence

```
/**
 * Storage Manager - CRUD operations
 */
class StorageManager {
  private storageKey = "snakeEvolution_scores";
  private currentData: StorageData | null = null;

  /**
   * Initialize storage
   */
```

```
  initialize(): void {
    const raw = localStorage.getItem(this.storageKey);

    if (raw) {
      try {
        this.currentData = JSON.parse(raw);
        this.validateData();
      } catch (error) {
        Logger.error("Storage parse error", { error });
        this.currentData = this.createEmptyStorage();
      }
    } else {
      this.currentData = this.createEmptyStorage();
    }
  }

  /**
   * Salva high score
   */
  saveHighScore(entry: HighScoreEntry): boolean {
    if (!this.currentData) return false;

    // Validazione
    if (!this.validateHighScoreEntry(entry)) {
      Logger.warn("Invalid high score entry", { entry });
      return false;
    }

    // Aggiungi entry
    this.currentData.scores.push(entry);

    // Ordina e mantieni top 10
    this.currentData.scores.sort((a, b) =&gt; b.score - a.score);
    this.currentData.scores = this.currentData.scores.slice(0, 10);

    // Update metadata
    this.currentData.metadata.lastUpdatedAt = new Date().toISOString();

    // Persisti
    return this.persist();
  }

  /**
   * Get top scores
   */
  getTopScores(): HighScoreEntry[] {
    return this.currentData?.scores || [];
  }

  /**
   * Persisti storage al localStorage
   */
  private persist(): boolean {
    try {
      const data = {
        ...this.currentData,
```

```
      metadata: {
        ...this.currentData!.metadata,
        checksumValid: true,
        lastUpdatedAt: new Date().toISOString()
      }
    };

    localStorage.setItem(this.storageKey, JSON.stringify(data));
    return true;
  } catch (error) {
    if (error instanceof DOMException &amp;&amp; error.code === 22) {
      // QuotaExceededError
      Logger.error("Storage quota exceeded", {});
      return false;
    }
    throw error;
  }
}

private validateData(): void {
  if (!this.currentData) return;

  // Verifica versione
  if (this.currentData.version !== "1.0") {
    Logger.warn("Storage version mismatch", {});
  }

  // Validazione checksum
  this.currentData.scores.forEach(score =&gt; {
    if (!this.validateChecksum(score)) {
      Logger.warn("Score checksum invalid", { scoreId: score.id });
    }
  });
}

private validateHighScoreEntry(entry: HighScoreEntry): boolean {
  return (
    entry.playerName.length &gt; 0 &amp;&amp; entry.playerName.length &lt;= 20 &amp;&am
    entry.score &gt;= 0 &amp;&amp;
    entry.evolutionStageReached &gt;= 0 &amp;&amp; entry.evolutionStageReached &lt;= 4
  );
}

private validateChecksum(entry: HighScoreEntry): boolean {
  // Implementazione MD5 checksum
  const data = `${entry.playerName}|${entry.score}|${entry.timestamp}`;
  // const computed = MD5(data);
  // return computed === entry.checksum;
  return true; // Placeholder
}

private createEmptyStorage(): StorageData {
  return {
    version: "1.0",
    dataVersion: 1,
    scores: [],
```

```
      settings: {
        audioEnabled: true,
        masterVolume: 0.8,
        lastPlayedDate: new Date().toISOString()
      },
      metadata: {
        totalSessions: 0,
        createdAt: new Date().toISOString(),
        lastUpdatedAt: new Date().toISOString(),
        checksumValid: true
      }
    };
  }

  /**
   * Clear all data (GDPR right to delete)
   */
  clearAllData(): void {
    localStorage.removeItem(this.storageKey);
    this.currentData = this.createEmptyStorage();
  }

  /**
   * Export data (GDPR right to data portability)
   */
  exportData(): string {
    return JSON.stringify(this.currentData, null, 2);
  }
}
```

7. Audio System Implementation

```
/**
 * Audio Manager - Web Audio API wrapper
 */
class AudioManager {
  private audioContext: AudioContext;
  private sounds: Map&lt;string, AudioBuffer&gt; = new Map();
  private masterGain: GainNode;
  private musicGain: GainNode;
  private sfxGain: GainNode;
  private isAudioEnabled: boolean = true;
  private currentMusic: AudioBufferSourceNode | null = null;

  constructor() {
    this.audioContext = new (window.AudioContext ||
      (window as any).webkitAudioContext)();

    // Setup gain nodes
    this.masterGain = this.audioContext.createGain();
    this.masterGain.connect(this.audioContext.destination);

    this.musicGain = this.audioContext.createGain();
    this.musicGain.connect(this.masterGain);
```

```
  this.sfxGain = this.audioContext.createGain();
  this.sfxGain.connect(this.masterGain);
}

/**
 * Carica audio asset
 */
async loadSound(name: string, url: string): Promise&lt;void&gt; {
  try {
    const response = await fetch(url);
    const arrayBuffer = await response.arrayBuffer();
    const audioBuffer = await this.audioContext.decodeAudioData(arrayBuffer);
    this.sounds.set(name, audioBuffer);
  } catch (error) {
    Logger.error("Audio load error", { name, error });
  }
}

/**
 * Play sound effect
 */
playSound(name: string, volume: number = 0.5): void {
  if (!this.isAudioEnabled) return;

  const buffer = this.sounds.get(name);
  if (!buffer) {
    Logger.warn("Sound not found", { name });
    return;
  }

  const source = this.audioContext.createBufferSource();
  source.buffer = buffer;

  const gain = this.audioContext.createGain();
  gain.gain.value = volume;
  gain.connect(this.sfxGain);

  source.connect(gain);
  source.start(0);
}

/**
 * Play looping background music
 */
playMusic(name: string, volume: number = 0.3): void {
  if (!this.isAudioEnabled) return;

  // Stop current music
  if (this.currentMusic) {
    this.currentMusic.stop();
  }

  const buffer = this.sounds.get(name);
  if (!buffer) return;
```

```
    const source = this.audioContext.createBufferSource();
    source.buffer = buffer;
    source.loop = true;

    const gain = this.audioContext.createGain();
    gain.gain.value = volume;
    gain.connect(this.musicGain);

    source.connect(gain);
    source.start(0);
    this.currentMusic = source;
  }

  /**
   * Set master volume [0-1]
   */
  setMasterVolume(volume: number): void {
    this.masterGain.gain.value = Math.max(0, Math.min(1, volume));
  }

  /**
   * Toggle audio
   */
  toggleAudio(enabled: boolean): void {
    this.isAudioEnabled = enabled;

    if (!enabled &amp;&amp; this.currentMusic) {
      this.currentMusic.stop();
      this.currentMusic = null;
    }
  }
}
```

8. Testing & Debugging Infrastructure

8.1 Logger Implementation

```
/**
 * Logger - Centralized logging
 */
class Logger {
  private static logs: any[] = [];
  private static maxLogs: number = 1000;

  static log(level: "INFO" | "WARN" | "ERROR" | "DEBUG", message: string, data?: any): vo
    const logEntry = {
      timestamp: new Date().toISOString(),
      level,
      message,
      data,
      url: window.location.href,
      userAgent: navigator.userAgent
    };
```

```
    // Console output
    const logMethod = level === "ERROR" ? console.error :
                      level === "WARN" ? console.warn :
                      level === "DEBUG" ? console.debug :
                      console.log;
    logMethod(`[${level}] ${message}`, data || "");

    // Store locally
    this.logs.push(logEntry);
    if (this.logs.length &gt; this.maxLogs) {
      this.logs.shift();
    }
  }

  static info(message: string, data?: any) { this.log("INFO", message, data); }
  static warn(message: string, data?: any) { this.log("WARN", message, data); }
  static error(message: string, data?: any) { this.log("ERROR", message, data); }
  static debug(message: string, data?: any) { this.log("DEBUG", message, data); }

  /**
   * Get logs for debugging
   */
  static getLogs(): any[] {
    return [...this.logs];
  }

  /**
   * Export logs
   */
  static exportLogs(): string {
    return JSON.stringify(this.logs, null, 2);
  }

  /**
   * Clear logs
   */
  static clear(): void {
    this.logs = [];
  }
}
```

8.2 Performance Profiler

```
/**
 * Performance Profiler - FPS tracking &amp;amp; frame time analysis
 */
class PerformanceProfiler {
  private frameTimes: number[] = [];
  private maxSamples: number = 60;
  private lastFrameTime: number = 0;

  /**
   * Record frame time
   */
```

```
  recordFrame(): void {
    const now = performance.now();

    if (this.lastFrameTime &gt; 0) {
      const frameTime = now - this.lastFrameTime;
      this.frameTimes.push(frameTime);

      if (this.frameTimes.length &gt; this.maxSamples) {
        this.frameTimes.shift();
      }
    }

    this.lastFrameTime = now;
  }

  /**
   * Get average FPS
   */
  getAverageFPS(): number {
    if (this.frameTimes.length === 0) return 0;

    const avgFrameTime = this.frameTimes.reduce((a, b) =&gt; a + b, 0) /
                         this.frameTimes.length;
    return 1000 / avgFrameTime;
  }

  /**
   * Get min/max frame time
   */
  getFrameTimeStats(): { min: number; max: number; avg: number } {
    if (this.frameTimes.length === 0) {
      return { min: 0, max: 0, avg: 0 };
    }

    return {
      min: Math.min(...this.frameTimes),
      max: Math.max(...this.frameTimes),
      avg: this.frameTimes.reduce((a, b) =&gt; a + b, 0) / this.frameTimes.length
    };
  }

  /**
   * Get performance report
   */
  getReport(): object {
    return {
      fps: this.getAverageFPS().toFixed(2),
      frameTimeStats: this.getFrameTimeStats(),
      memoryUsage: (performance as any).memory?.usedJSHeapSize / 1048576,
      timestamp: new Date().toISOString()
    };
  }
}
```

9. Build & Deployment Configuration

9.1 Webpack Configuration

```
// webpack.config.js
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  mode: 'production',
  entry: './src/main.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.[contenthash].js',
    clean: true
  },

  devtool: 'source-map',

  module: {
    rules: [
      {
        test: /\.js$/,
        use: 'babel-loader',
        exclude: /node_modules/
      },
      {
        test: /\.css$/,
        use: [MiniCssExtractPlugin.loader, 'css-loader']
      },
      {
        test: /\.(png|jpg|wav|mp3)$/,
        type: 'asset',
        parser: {
          dataUrlCondition: {
            maxSize: 8 * 1024 // 8kb
          }
        }
      }
    ]
  },

  plugins: [
    new HtmlWebpackPlugin({
      template: './index.html',
      minify: true
    }),
    new MiniCssExtractPlugin({
      filename: 'style.[contenthash].css'
    })
  ],

  optimization: {
    minimize: true,
```

```
    minimizer: ['...']
  }
};
```

10. Critical Path Analysis

10.1 Game Loop Latency Budget

```
Frame Duration Target: 16.67ms (60 FPS)

Breakdown:
├─ Input Processing: 1ms max
├─ Update Logic: 3ms max
│  ├─ Snake Movement: 0.5ms
│  ├─ Collision Detection: 1ms
│  └─ Evolution Check: 0.5ms
├─ Rendering: 10ms max
│  ├─ Clear Canvas: 1ms
│  ├─ Draw Background: 1ms
│  ├─ Draw Entities: 5ms
│  └─ Draw UI: 3ms
├─ Audio: 1ms max
└─ Overhead: 1.67ms

Total: 16.67ms (100%)
```

11. Appendici

11.1 Glossario Tecnico

- **Canvas 2D API**: API per disegnare grafica 2D in HTML5
- **Event Bus**: Pattern mediator per pub/sub messaging
- **Object Pool**: Pre-allocazione di oggetti riutilizzabili
- **Dirty Rectangle**: Rendering solo aree cambiate
- **Frame Buffer**: Memoria per rendering frame corrente
- **GC (Garbage Collection)**: Liberazione memoria automatica
- **Checksum**: Hash per validazione integrità dati

11.2 Browser Compatibility

Browser	Min Version	Support
Chrome	60+	✓ Full
Firefox	55+	✓ Full

Browser	Min Version	Support
Safari	11+	✓ Full
Edge	79+	✓ Full
Mobile Safari	11+	✓ Full
Chrome Android	60+	✓ Full

11.3 Resource Budget

- **Bundle Size**: < 500KB (minified + gzipped)
- **Assets**: < 14.5MB (audio + graphics)
- **Memory**: < 50MB runtime
- **Load Time**: < 2 seconds (target)

Documento Completato

Senior Software Architect
Data: Novembre 2025
Status: Production Ready

Questo documento serve come riferimento completo per l'implementazione tecnica di Snake Evolution. Ogni sezione fornisce specifiche concrete, pattern architetturali, e linee guida per garantire un'implementazione robusta e performante.
[1]

⁂

1. PRD_SnakeEvolution.pdf