Architecture

Engine design and structure

Overview

Origami Engine is designed as a lightweight, browser-based game engine inspired by GameMaker Studio 1.4. It's built with TypeScript and provides a familiar GameObject-based programming model.

Design Philosophy

1. GameMaker Studio Compatibility

Goal: Make GMS 1.4 developers feel at home

How:

- Same event structure (`create` , `step` , `draw`)
- Familiar global functions (`instance_create` , `place_meeting`)
- Same coordinate system (0,0 = top-left, Y+ = down)
- Compatible naming (vk_*, mb_*, etc.)

2. TypeScript First

Goal: Modern type safety with strict mode

How:

- All code written in TypeScript
- Strict compiler options enabled
- Full type definitions exported
- No `any` types in public API

3. Browser Native

Goal: Run anywhere without plugins

How:

- Pure JavaScript/TypeScript
- Canvas 2D API for rendering
- No WebGL requirement
- No external dependencies (runtime)

4. Simple and Transparent

Goal: Easy to understand and modify

How:

- Minimal abstraction layers
- Clear file organization
- Well-commented code
- No magic/hidden behavior

Core Architecture

Monorepo Structure

```
Origami-Engine/
├── packages/
│   ├── runtime/      # Core engine (published to npm)
│   └── cli/          # CLI tool (published to npm)
├── platformer/       # Example game
├── docs/             # Documentation
└── scripts/          # Build tools
```

Why monorepo?

- Shared tooling and configuration
- Atomic commits across packages
- Easy local development

- Single source of truth

Runtime Package

Directory Structure

```
packages/runtime/
├── src/
│   ├── core/
│   │   ├── GameObject.ts       # Base class
│   │   ├── GameEngine.ts       # Main engine loop
│   │   ├── SpriteManager.ts    # Sprite loading/animation
│   │   └── types.ts            # TypeScript types
│   ├── systems/
│   │   ├── CollisionSystem.ts  # AABB collision
│   │   ├── RenderSystem.ts     # Canvas drawing
│   │   ├── InputSystem.ts      # Keyboard/mouse
│   │   └── RoomSystem.ts       # Room management
│   ├── functions/
│   │   ├── instance.ts         # instance_* functions
│   │   ├── collision.ts        # place_meeting, etc.
│   │   ├── drawing.ts          # draw_* functions
│   │   ├── motion.ts           # point_distance, etc.
│   │   └── input.ts            # keyboard_check, etc.
│   └── index.ts                # Public exports
├── dist/                       # Compiled JS
└── DOCUMENTATION.md            # API reference
```

Game Loop

60 FPS fixed timestep:

```
class GameEngine {
  private readonly TARGET_FPS = 60;
  private readonly FRAME_TIME = 1000 / 60; // 16.67ms

  private gameLoop(): void {
    const now = performance.now();
    const delta = now - this.lastTime;

    if (delta >= this.FRAME_TIME) {
      this.update();        // Step all objects
      this.render();        // Draw all objects
      this.lastTime = now - (delta % this.FRAME_TIME);
    }

    requestAnimationFrame(() => this.gameLoop());
  }

  private update(): void {
    // 1. Update motion (hspeed/vspeed -> x/y)
    // 2. Call step() on all instances (by order)
    // 3. Update sprite animation
    // 4. Remove destroyed instances
  }

  private render(): void {
    // 1. Clear canvas
    // 2. Apply view/camera transform
    // 3. Draw instances (sorted by depth)
    // 4. Draw debug overlay (if enabled)
  }
}
```

Why fixed timestep?

- Predictable physics
- Consistent across devices
- Easier debugging
- GMS compatibility

Key Systems

GameObject System

Base class for all game objects:

```
export abstract class GameObject {
  // Position
  public x: number = 0;
  public y: number = 0;
  public xprevious: number = 0;
  public yprevious: number = 0;

  // Motion
  public speed: number = 0;
  public direction: number = 0;
  public hspeed: number = 0;
  public vspeed: number = 0;

  // Sprite
  public sprite_index: string | null = null;
  public image_index: number = 0;
  public image_speed: number = 1.0;

  // Rendering
  public visible: boolean = true;
  public depth: number = 0;

  // Events (override in subclass)
  public create?(): void;
  public step?(): void;
  public draw?(): void;
  public roomStart?(): void;
  public roomEnd?(): void;
}
```

Instance management:

- Engine maintains `Map<string, GameObject[]>` by type
- Fast lookups for collision/find operations

- Instances created via `instance_create()`
- Destroyed via `instance_destroy()`

Collision System

AABB (Axis-Aligned Bounding Box) collision:

```
class CollisionSystem {
  public placeMeeting(
    instance: GameObject,
    x: number,
    y: number,
    objectType: string
  ): boolean {
    const bbox1 = this.getBoundingBox(instance, x, y);
    const targets = this.engine.getInstances(objectType);

    for (const target of targets) {
      if (target === instance) continue;
      const bbox2 = this.getBoundingBox(target, target.x, target.y);

      if (this.aabbIntersect(bbox1, bbox2)) {
        return true;
      }
    }

    return false;
  }

  private aabbIntersect(a: BBox, b: BBox): boolean {
    return !(a.right < b.left ||
             a.left > b.right ||
             a.bottom < b.top ||
             a.top > b.bottom);
  }
}
```

Bounding box calculation:

- Defined in sprite metadata.json
- Defaults to full sprite size
- Supports custom bbox for tighter collision
- Accounts for sprite origin

Performance:

- O(n*m) worst case (all instances vs all targets)
- Optimized with early exits
- Future: Spatial hashing for large games

Render System

Canvas 2D rendering:

```
class RenderSystem {
  private ctx: CanvasRenderingContext2D;

  public render(instances: GameObject[]): void {
    // Clear screen
    this.ctx.fillStyle = this.backgroundColor;
    this.ctx.fillRect(0, 0, this.width, this.height);

    // Sort by depth (higher = behind)
    const sorted = instances.sort((a, b) => b.depth - a.depth);

    // Apply camera transform
    this.ctx.save();
    if (this.view) {
      this.ctx.translate(-this.view.x, -this.view.y);
    }

    // Draw each instance
    for (const instance of sorted) {
      if (!instance.visible) continue;

      if (instance.draw) {
        instance.draw();
      } else {
        draw_self.call(instance); // Auto-draw sprite
      }
    }

    this.ctx.restore();

    // Draw debug overlay (if enabled)
    if (this.debugMode) {
      this.drawDebugOverlay();
    }
  }
}
```

Drawing pipeline:

1. Clear canvas

2. Sort instances by depth
3. Apply camera transform
4. Draw instances (high depth to low depth)
5. Reset transform
6. Draw debug overlay

Input System

Event-based input with state tracking:

```
class InputSystem {
  private keysDown: Set<number> = new Set();
  private keysPressed: Set<number> = new Set();
  private keysReleased: Set<number> = new Set();

  constructor() {
    window.addEventListener('keydown', (e) => {
      if (!this.keysDown.has(e.keyCode)) {
        this.keysPressed.add(e.keyCode);
      }
      this.keysDown.add(e.keyCode);
    });

    window.addEventListener('keyup', (e) => {
      this.keysDown.delete(e.keyCode);
      this.keysReleased.add(e.keyCode);
    });
  }

  public update(): void {
    // Clear pressed/released at end of frame
    this.keysPressed.clear();
    this.keysReleased.clear();
  }

  public keyCheck(key: number): boolean {
    return this.keysDown.has(key);
  }

  public keyPressed(key: number): boolean {
    return this.keysPressed.has(key);
  }

  public keyReleased(key: number): boolean {
    return this.keysReleased.has(key);
  }
}
```

Mouse position calculated from canvas:

```
private updateMousePosition(e: MouseEvent): void {
  const rect = this.canvas.getBoundingClientRect();
  const scaleX = this.canvas.width / rect.width;
  const scaleY = this.canvas.height / rect.height;

  mouse_x = (e.clientX - rect.left) * scaleX + view_xview[0];
  mouse_y = (e.clientY - rect.top) * scaleY + view_yview[0];
}
```

Sprite System

Folder-based sprites:

```
sprites/spr_player/
├── metadata.json
├── frame_0.png
├── frame_1.png
└── frame_2.png
```

Loading pipeline:

```
class SpriteManager {
  private sprites: Map<string, Sprite> = new Map();

  public async loadSprite(name: string, path: string): Promise<void> {
    // 1. Load metadata.json
    const metadata = await fetch(`${path}/metadata.json`).then(r => r.json());

    // 2. Load all frames
    const frames: HTMLImageElement[] = [];
    let frameIndex = 0;
    while (true) {
      try {
        const img = await this.loadImage(`${path}/frame_${frameIndex}.png`);
        frames.push(img);
        frameIndex++;
      } catch {
        break; // No more frames
      }
    }

    // 3. Store sprite
    this.sprites.set(name, {
      frames,
      origin: metadata.origin,
      fps: metadata.fps,
      bbox: metadata.bbox
    });
  }
}
```

Animation:

```
private updateAnimation(instance: GameObject, sprite: Sprite): void {
  instance.image_index += instance.image_speed * (sprite.fps / 60);

  // Loop back to start
  if (instance.image_index >= sprite.frames.length) {
    instance.image_index = 0;
  }
}
```

Room System

JSON-based rooms:

```
interface RoomDefinition {
  name: string;
  width: number;
  height: number;
  speed: number;
  backgroundColor: string;
  instances: InstanceDefinition[];
  views: ViewDefinition[];
}

class RoomSystem {
  public async loadRoom(name: string): Promise<void> {
    // 1. Load room JSON
    const room = await fetch(`rooms/${name}.json`).then(r => r.json());

    // 2. Destroy non-persistent instances
    this.engine.clearInstances({ keepPersistent: true });

    // 3. Create room instances
    for (const def of room.instances) {
      await instance_create(def.x, def.y, def.object);
    }

    // 4. Set up views/camera
    if (room.views.length > 0) {
      this.setupView(room.views[0]);
    }

    // 5. Call roomStart() on all instances
    for (const instance of this.engine.getAllInstances()) {
      if (instance.roomStart) {
        instance.roomStart();
      }
    }
  }
}
```

CLI Package

Command Structure

```
packages/cli/src/
├── commands/
│   ├── create.ts      # ori create
│   ├── update.ts      # ori update/check/full
│   ├── dev.ts         # ori dev
│   └── build.ts       # ori build
├── utils/
│   ├── config-manager.ts
│   ├── git-operations.ts
│   ├── template-fetcher.ts
│   ├── migration-runner.ts
│   └── backup-manager.ts
└── index.ts           # CLI entry point
```

Update System

Three modes:

1. **Check**: Read-only version info
2. **Update**: Engine only, no code changes
3. **Full**: Engine + AST migrations

Migration pipeline:

```
class MigrationRunner {
  public async run(): Promise<void> {
    // 1. Parse TypeScript with Babel
    const ast = parse(code, { sourceType: 'module', plugins: ['typescript'] });

    // 2. Transform AST
    traverse(ast, {
      CallExpression(path) {
        // Rename deprecated functions
        if (path.node.callee.name === 'instance_create_layer') {
          path.node.callee.name = 'instance_create';
        }
      }
    });

    // 3. Generate updated code
    const output = generate(ast);

    // 4. Write back to file
    fs.writeFileSync(filePath, output.code);
  }
}
```

Performance Considerations

Bottlenecks

1. **Collision checks**: O(n*m) for all instances
2. **Drawing**: Canvas state changes expensive
3. **Instance creation**: Object allocation

Optimizations

1. **Spatial partitioning** (planned):
    - Grid-based bucketing
    - Only check nearby instances

2. **Object pooling** (user-implemented):
    - Reuse instances instead of create/destroy
    - Especially for particles/bullets
3. **Culling**:
    - Skip off-screen rendering
    - Early returns in step() for distant objects

Extension Points

Custom Systems

Add new systems without modifying core:

```
class AudioSystem {
  private sounds: Map<string, HTMLAudioElement> = new Map();

  public playSound(name: string): void {
    const sound = this.sounds.get(name);
    if (sound) {
      sound.currentTime = 0;
      sound.play();
    }
  }
}

// Register with engine
engine.registerSystem('audio', new AudioSystem());
```

Custom Events

Add new GameObject events:

```
// In engine loop
for (const instance of instances) {
  if ('onCollision' in instance) {
    instance.onCollision();
  }
}
```

Future Architecture

Planned Improvements

1. **ECS (Entity Component System)** option:
   - Alternative to GameObject inheritance
   - Better performance for large games
   - Optional, not required
2. **Web Workers**:
   - Off-thread collision detection
   - Async asset loading
3. **WebGL Renderer**:
   - Optional high-performance rendering
   - Fallback to Canvas 2D
4. **Plugin System**:
   - Third-party extensions
   - Audio, networking, etc.

Resources

- **TypeScript Handbook**
- **Canvas API Docs**
- **GameMaker Studio 1.4 Docs**

- **90-contributing.md** - How to contribute

← Back to Index