Performance

Optimization techniques for smooth gameplay

Overview

Origami Engine runs at 60 FPS by default. Most games will run smoothly without optimization, but for games with many objects or complex logic, these techniques can help.

Performance Monitoring

FPS Counter

Press **F3** to show debug overlay with:

- **FPS** (frames per second) - Should be ~60
- **Instance count** - Total active objects
- **Draw calls** - Render operations per frame

Target: 60 FPS constant

Acceptable: 55-60 FPS with occasional drops

Problem: Below 50 FPS consistently

Browser Developer Tools

Performance Tab (Chrome/Edge):

1. Press F12
2. Go to Performance tab
3. Click Record
4. Play game for 5-10 seconds

5. Stop recording
6. Analyze:
    - **Yellow** = JavaScript execution
    - **Purple** = Rendering
    - **Green** = Painting

Look for:

- Long yellow bars (slow step() functions)
- Many purple bars (excessive drawing)
- Memory leaks (increasing heap size)

Instance Management

Limit Active Instances

```
step(): void {
  // Don't create unlimited particles
  if (instance_number('obj_particle') < 100) {
    await instance_create(this.x, this.y, 'obj_particle');
  }
}
```

Destroy Off-Screen Objects

```
step(): void {
  // Destroy bullets that leave the room
  if (this.x < 0 || this.x > room_width ||
      this.y < 0 || this.y > room_height) {
    instance_destroy.call(this);
  }
}
```

Use Object Pooling

Instead of creating/destroying frequently:

```
export class obj_bullet_pool extends GameObject {
  private pool: GameObject[] = [];
  private readonly POOL_SIZE = 50;

  create(): void {
    // Pre-create bullets
    for (let i = 0; i < this.POOL_SIZE; i++) {
      const bullet = await instance_create(-1000, -1000, 'obj_bullet');
      bullet.visible = false;
      this.pool.push(bullet);
    }
  }

  public getBullet(x: number, y: number): GameObject | null {
    // Find inactive bullet
    for (const bullet of this.pool) {
      if (!bullet.visible) {
        bullet.x = x;
        bullet.y = y;
        bullet.visible = true;
        bullet.speed = 10;
        return bullet;
      }
    }
    return null;
  }

  public returnBullet(bullet: GameObject): void {
    bullet.visible = false;
    bullet.x = -1000;
    bullet.y = -1000;
    bullet.speed = 0;
  }
}
```

Step Event Optimization

Reduce Collision Checks

Bad: Checking every frame

```
step(): void {
  // Expensive! 60 times per second
  const wall = place_meeting.call(this, this.x, this.y, 'obj_wall');
}
```

Good: Check only when moving

```
step(): void {
  // Only check if actually moving
  if (this.hspeed !== 0 || this.vspeed !== 0) {
    const wall = place_meeting.call(this, this.x + this.hspeed, this.y, 'obj_wa
  }
}
```

Batch Operations

Bad: Checking distance every frame

```
step(): void {
  const player = instance_find('obj_player', 0);
  const dist = point_distance(this.x, this.y, player.x, player.y);
  // ...
}
```

Good: Check every N frames

```
private checkTimer: number = 0;
private cachedDistance: number = 0;

step(): void {
  this.checkTimer++;

  // Update distance every 10 frames
  if (this.checkTimer % 10 === 0) {
    const player = instance_find('obj_player', 0);
    if (player) {
      this.cachedDistance = point_distance(this.x, this.y, player.x, player.y);
    }
  }

  // Use cached value
  if (this.cachedDistance < 100) {
    // Do something
  }
}
```

Early Returns

```
step(): void {
  // Skip if too far from player
  const player = instance_find('obj_player', 0);
  if (!player) return;

  const dist = point_distance(this.x, this.y, player.x, player.y);
  if (dist > 500) return; // Too far, skip logic

  // Expensive AI logic only runs when close
  this.runAI();
}
```

Avoid instance_find in Loops

Bad:

```
step(): void {
  for (let i = 0; i < 10; i++) {
    const player = instance_find('obj_player', 0); // 10 calls!
    // ...
  }
}
```

Good:

```
step(): void {
  const player = instance_find('obj_player', 0); // 1 call
  if (!player) return;

  for (let i = 0; i < 10; i++) {
    // Use cached reference
    const dist = point_distance(this.x, this.y, player.x, player.y);
    // ...
  }
}
```

Draw Event Optimization

Cache Expensive Calculations

Bad:

```
draw(): void {
  // Recalculating every draw call
  const barWidth = 50 * (this.health / this.maxHealth);
  draw_rectangle(this.x - 25, this.y - 35, this.x - 25 + barWidth, this.y - 30,
}
```

Good:

```
private cachedHealthBar: number = 50;

step(): void {
  // Calculate once per frame in step
  this.cachedHealthBar = 50 * (this.health / this.maxHealth);
}

draw(): void {
  // Use cached value
  draw_rectangle(this.x - 25, this.y - 35, this.x - 25 + this.cachedHealthBar,
}
```

Skip Off-Screen Drawing

```
draw(): void {
  // Don't draw if off-screen
  const margin = 64;
  if (this.x < view_xview[0] - margin ||
      this.x > view_xview[0] + view_wview[0] + margin ||
      this.y < view_yview[0] - margin ||
      this.y > view_yview[0] + view_hview[0] + margin) {
    return;
  }

  draw_self.call(this);
  // Other drawing...
}
```

Minimize State Changes

Bad:

```
draw(): void {
  draw_set_color('#FF0000');
  draw_rectangle(10, 10, 20, 20, false);
  draw_set_color('#00FF00');
  draw_rectangle(30, 10, 40, 20, false);
  draw_set_color('#FF0000'); // Redundant!
  draw_rectangle(50, 10, 60, 20, false);
}
```

Good:

```
draw(): void {
  // Batch same-color draws
  draw_set_color('#FF0000');
  draw_rectangle(10, 10, 20, 20, false);
  draw_rectangle(50, 10, 60, 20, false);

  draw_set_color('#00FF00');
  draw_rectangle(30, 10, 40, 20, false);
}
```

Asset Optimization

Sprite Optimization

Reduce sprite sizes:

- Use 32x32 instead of 64x64 when possible
- Compress PNGs with tools like TinyPNG
- Reduce animation frame counts
  - 4 frames instead of 8 for idle
  - 6 frames instead of 12 for run

Optimize bounding boxes:

- Use custom `bbox` in metadata.json
- Smaller boxes = faster collision

```
{
  "origin": { "x": 16, "y": 16 },
  "fps": 10,
  "bbox": {
    "left": 8,
    "top": 8,
    "right": 24,
    "bottom": 24
  }
}
```

Reduce Animation Speed

Lower `fps` in sprite metadata:

```
{
  "fps": 6  // Instead of 10
}
```

Or reduce `image_speed`:

```
create(): void {
  this.sprite_index = 'spr_player';
  this.image_speed = 0.5; // Half speed
}
```

Memory Management

Avoid Memory Leaks

Bad: Creating references that never get cleaned

```
step(): void {
  // Creates new array every frame!
  this.targets = [];
  const enemies = instance_find_all('obj_enemy'); // Memory leak!
  this.targets.push(...enemies);
}
```

Good: Reuse or clean up properly

```
private targets: GameObject[] = [];

step(): void {
  // Clear and reuse array
  this.targets.length = 0;

  const count = instance_number('obj_enemy');
  for (let i = 0; i < count; i++) {
    const enemy = instance_find('obj_enemy', i);
    if (enemy) this.targets.push(enemy);
  }
}
```

Clean Up on Destroy

```
create(): void {
  this.particles = [];
  this.timers = [];
}

// Clean up when destroyed
onDestroy(): void {
  this.particles = [];
  this.timers = [];
  // Clear any references
}
```

Advanced Optimizations

Spatial Partitioning

For games with hundreds of objects, divide room into grid:

```
export class obj_spatial_grid extends GameObject {
  private grid: Map<string, GameObject[]> = new Map();
  private readonly CELL_SIZE = 128;

  public addObject(obj: GameObject): void {
    const cell = this.getCell(obj.x, obj.y);
    if (!this.grid.has(cell)) {
      this.grid.set(cell, []);
    }
    this.grid.get(cell)!.push(obj);
  }

  public getNearby(x: number, y: number, radius: number): GameObject[] {
    const nearby: GameObject[] = [];
    const cells = this.getCellsInRadius(x, y, radius);

    for (const cell of cells) {
      const objects = this.grid.get(cell);
      if (objects) {
        nearby.push(...objects);
      }
    }

    return nearby;
  }

  private getCell(x: number, y: number): string {
    const cellX = Math.floor(x / this.CELL_SIZE);
    const cellY = Math.floor(y / this.CELL_SIZE);
    return `${cellX},${cellY}`;
  }

  private getCellsInRadius(x: number, y: number, radius: number): string[] {
    // Return cell keys within radius
    // Implementation...
    return [];
  }
}
```

Lazy Evaluation

Only calculate when needed:

```
private _cachedPath: Point[] | null = null;

get path(): Point[] {
  if (!this._cachedPath) {
    // Calculate path only when requested
    this._cachedPath = this.calculatePath();
  }
  return this._cachedPath;
}

step(): void {
  // Invalidate cache when position changes
  if (this.x !== this.xprevious || this.y !== this.yprevious) {
    this._cachedPath = null;
  }
}
```

Debug Optimization

Remove Debug Code

Before release:

```
// BAD: Left in production
step(): void {
  show_debug_message.call(this, `Position: ${this.x}, ${this.y}`); // Slow!
  console.log('Health:', this.health); // Slow!
}
```

Good: Use debug flag

```
private readonly DEBUG = false; // Set to false for release

step(): void {
  if (this.DEBUG) {
    show_debug_message.call(this, `Position: ${this.x}, ${this.y}`);
  }
}
```

Common Performance Issues

Too Many Particles

Symptom: FPS drops when explosions/effects happen

Solution: Limit particle count

```
if (instance_number('obj_particle') < 200) {
  await instance_create(this.x, this.y, 'obj_particle');
}
```

Heavy Collision Checks

Symptom: FPS drops with many enemies

Solution: Use spatial partitioning or distance checks first

```
// Check distance before expensive collision
const dist = point_distance(this.x, this.y, player.x, player.y);
if (dist < 32) {
  // Only check collision if very close
  if (place_meeting.call(this, this.x, this.y, 'obj_player')) {
    // Collision!
  }
}
```

Excessive Drawing

Symptom: Low FPS even with few objects

Solution: Reduce draw calls, skip off-screen

```
draw(): void {
  if (!this.isOnScreen()) return;
  draw_self.call(this);
}
```

Performance Checklist

Before release:

- ✅ Maintain 60 FPS in all situations
- ✅ Instance count under 1000
- ✅ Remove all debug logging
- ✅ Optimize sprite sizes
- ✅ Destroy off-screen objects
- ✅ Cache expensive calculations
- ✅ Use early returns in step()
- ✅ Skip off-screen drawing
- ✅ Test on slower devices/browsers
- ✅ Check for memory leaks (F12 Memory tab)

Next Steps

- **10-debugging.md** - Debug tools for profiling
- **40-common-patterns.md** - Efficient code patterns
- **41-deployment.md** - Optimization for deployment

← Back to Index