Common Patterns

Best practices and recipes for game development

Health System

A simple health system with damage and death:

```
export class obj_player extends GameObject {
  private health: number = 100;
  private maxHealth: number = 100;

  step(): void {
    // Take damage from enemies
    const enemy = instance_place.call(this, this.x, this.y, 'obj_enemy');
    if (enemy) {
      this.health -= 1;
      if (this.health <= 0) {
        game_restart();
      }
    }
  }

  draw(): void {
    draw_self.call(this);

    // Health bar above character
    const barWidth = 50;
    const healthPercent = this.health / this.maxHealth;

    // Background (red)
    draw_set_color('#FF0000');
    draw_rectangle(
      this.x - 25,
      this.y - 35,
      this.x + 25,
      this.y - 30,
      false
    );

    // Foreground (green, proportional to health)
    draw_set_color('#00FF00');
    draw_rectangle(
      this.x - 25,
      this.y - 35,
      this.x - 25 + (barWidth * healthPercent),
      this.y - 30,
      false
```

```
    );

    draw_set_color('#FFFFFF');
  }
}
```

Score System

Global Score Variable

```
// In src/main.ts, before engine.start():
(window as any).score = 0;
```

Collectible Object

```
export class obj_coin extends GameObject {
  create(): void {
    this.sprite_index = 'spr_coin';
    this.image_speed = 1.0;
  }

  step(): void {
    const player = instance_place.call(this, this.x, this.y, 'obj_player');
    if (player) {
      // Add to score
      (window as any).score += 10;
      // Destroy coin
      instance_destroy.call(this);
    }
  }
}
```

Score Display

```
export class obj_hud extends GameObject {
  draw(): void {
    draw_set_color('#FFFF00');
    draw_text(10, 10, `Score: ${(window as any).score}`);
    draw_set_color('#FFFFFF');
  }
}
```

Simple AI (Patrol)

An enemy that patrols back and forth:

```
export class obj_enemy extends GameObject {
  private moveDirection: number = 1; // 1 = right, -1 = left

  create(): void {
    this.sprite_index = 'spr_enemy';
    this.image_speed = 1.0;
  }

  step(): void {
    const speed = 2;
    this.x += speed * this.moveDirection;

    // Turn at walls
    if (place_meeting.call(this, this.x + speed, this.y, 'obj_wall')) {
      this.moveDirection *= -1;
      this.image_xscale *= -1; // Flip sprite
    }

    // Turn at edges (no ground ahead)
    if (!place_meeting.call(this, this.x, this.y + 32, 'obj_wall')) {
      this.moveDirection *= -1;
      this.image_xscale *= -1;
    }
  }
}
```

Platformer Physics

Complete platformer movement with gravity:

```
export class obj_player extends GameObject {
  private onGround: boolean = false;
  private readonly GRAVITY = 0.5;
  private readonly JUMP_POWER = -10;
  private readonly MOVE_SPEED = 4;

  create(): void {
    this.sprite_index = 'spr_player';
  }

  step(): void {
    // Horizontal movement
    if (keyboard_check(vk_d)) {
      if (!place_meeting.call(this, this.x + this.MOVE_SPEED, this.y, 'obj_wall
        this.x += this.MOVE_SPEED;
      }
    }
    if (keyboard_check(vk_a)) {
      if (!place_meeting.call(this, this.x - this.MOVE_SPEED, this.y, 'obj_wall
        this.x -= this.MOVE_SPEED;
      }
    }

    // Apply gravity
    this.vspeed += this.GRAVITY;

    // Check if on ground
    this.onGround = place_meeting.call(this, this.x, this.y + 1, 'obj_wall');

    // Jump
    if (keyboard_check_pressed(vk_space) && this.onGround) {
      this.vspeed = this.JUMP_POWER;
    }

    // Vertical collision
    if (place_meeting.call(this, this.x, this.y + this.vspeed, 'obj_wall')) {
      // Pixel-perfect collision
      while (!place_meeting.call(this, this.x, this.y + Math.sign(this.vspeed),
        this.y += Math.sign(this.vspeed);
      }
```

```
        this.vspeed = 0;
      } else {
        this.y += this.vspeed;
      }
    }
  }
```

State Machine

Managing complex behavior with states:

```
enum EnemyState {
  Patrol,
  Chase,
  Attack,
  Retreat
}

export class obj_enemy extends GameObject {
  private state: EnemyState = EnemyState.Patrol;
  private patrolDirection: number = 1;
  private readonly CHASE_DISTANCE = 200;
  private readonly ATTACK_DISTANCE = 50;

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

    const dist = point_distance(this.x, this.y, player.x, player.y);

    // State transitions
    switch (this.state) {
      case EnemyState.Patrol:
        if (dist < this.CHASE_DISTANCE) {
          this.state = EnemyState.Chase;
        }
        break;
      case EnemyState.Chase:
        if (dist < this.ATTACK_DISTANCE) {
          this.state = EnemyState.Attack;
        } else if (dist > this.CHASE_DISTANCE + 50) {
          this.state = EnemyState.Patrol;
        }
        break;
      case EnemyState.Attack:
        if (dist > this.ATTACK_DISTANCE + 20) {
          this.state = EnemyState.Chase;
        }
        break;
    }
```

```
    // State behavior
    switch (this.state) {
      case EnemyState.Patrol:
        this.handlePatrol();
        break;
      case EnemyState.Chase:
        this.handleChase(player);
        break;
      case EnemyState.Attack:
        this.handleAttack(player);
        break;
    }
  }

  private handlePatrol(): void {
    const speed = 1;
    this.x += speed * this.patrolDirection;

    if (place_meeting.call(this, this.x + speed, this.y, 'obj_wall')) {
      this.patrolDirection *= -1;
    }
  }

  private handleChase(player: GameObject): void {
    const speed = 3;
    const dir = point_direction(this.x, this.y, player.x, player.y);
    this.x += lengthdir_x(speed, dir);
    this.y += lengthdir_y(speed, dir);
  }

  private handleAttack(player: GameObject): void {
    // Attack logic here
  }

  draw(): void {
    draw_self.call(this);

    // Debug: show current state
    const stateName = EnemyState[this.state];
    draw_text(this.x, this.y - 30, stateName);
```

```
  }
}
```

Timer Pattern

Creating timers and delays:

```
export class obj_spawner extends GameObject {
  private spawnTimer: number = 0;
  private readonly SPAWN_DELAY = 120; // 2 seconds at 60 FPS

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

    if (this.spawnTimer >= this.SPAWN_DELAY) {
      await instance_create(this.x, this.y, 'obj_enemy');
      this.spawnTimer = 0; // Reset timer
    }
  }
}
```

Multiple Timers

```
export class obj_player extends GameObject {
  private shootTimer: number = 0;
  private invincibilityTimer: number = 0;

  step(): void {
    // Shoot cooldown (0.5 seconds)
    if (this.shootTimer > 0) {
      this.shootTimer--;
    }

    // Invincibility duration (2 seconds)
    if (this.invincibilityTimer > 0) {
      this.invincibilityTimer--;
    }

    // Shoot
    if (keyboard_check(vk_space) && this.shootTimer === 0) {
      await instance_create(this.x, this.y, 'obj_bullet');
      this.shootTimer = 30; // 0.5 seconds
    }

    // Take damage
    if (!this.invincibilityTimer) {
      const enemy = instance_place.call(this, this.x, this.y, 'obj_enemy');
      if (enemy) {
        this.health -= 10;
        this.invincibilityTimer = 120; // 2 seconds
      }
    }
  }

  draw(): void {
    // Flash while invincible
    if (this.invincibilityTimer > 0 && this.invincibilityTimer % 4 < 2) {
      // Skip drawing (flashing effect)
      return;
    }
    draw_self.call(this);
```

```
    }
}
```

Bullet Pattern

Shooting projectiles:

```
export class obj_player extends GameObject {
  private shootCooldown: number = 0;

  step(): void {
    if (this.shootCooldown > 0) {
      this.shootCooldown--;
    }

    // Shoot on spacebar
    if (keyboard_check(vk_space) && this.shootCooldown === 0) {
      await instance_create(this.x, this.y, 'obj_bullet');
      this.shootCooldown = 10; // 10 frames between shots
    }
  }
}

export class obj_bullet extends GameObject {
  create(): void {
    this.sprite_index = 'spr_bullet';
    this.speed = 10;
    this.direction = 0; // Right
  }

  step(): void {
    // Destroy if off-screen
    if (this.x > room_width || this.x < 0 || this.y > room_height || this.y < 
      instance_destroy.call(this);
    }

    // Destroy on wall hit
    if (place_meeting.call(this, this.x, this.y, 'obj_wall')) {
      instance_destroy.call(this);
    }

    // Damage enemies
    const enemy = instance_place.call(this, this.x, this.y, 'obj_enemy');
    if (enemy) {
      instance_destroy.call(enemy);
      instance_destroy.call(this);
    }
```

```
  }
}
```

Follow/Chase Pattern

Object that follows another:

```
export class obj_enemy extends GameObject {
  private readonly FOLLOW_SPEED = 2;
  private readonly STOP_DISTANCE = 50;

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

    const dist = point_distance(this.x, this.y, player.x, player.y);

    // Only follow if far enough away
    if (dist > this.STOP_DISTANCE) {
      const dir = point_direction(this.x, this.y, player.x, player.y);
      this.x += lengthdir_x(this.FOLLOW_SPEED, dir);
      this.y += lengthdir_y(this.FOLLOW_SPEED, dir);
    }
  }
}
```

Camera Shake

Screen shake effect:

```
export class obj_camera extends GameObject {
  private shakeAmount: number = 0;
  private shakeDuration: number = 0;

  step(): void {
    if (this.shakeDuration > 0) {
      this.shakeDuration--;

      // Apply random offset
      const offsetX = random_range(-this.shakeAmount, this.shakeAmount);
      const offsetY = random_range(-this.shakeAmount, this.shakeAmount);

      // Modify view position (requires view system)
      // view_xview[0] += offsetX;
      // view_yview[0] += offsetY;
    }
  }

  public shake(amount: number, duration: number): void {
    this.shakeAmount = amount;
    this.shakeDuration = duration;
  }
}

// Trigger shake from another object:
// const camera = instance_find('obj_camera', 0);
// if (camera) camera.shake(5, 20); // Shake 5 pixels for 20 frames
```

Particle System

Simple particle effects:

```
export class obj_particle extends GameObject {
  private life: number = 60; // 1 second
  private vx: number = 0;
  private vy: number = 0;

  create(): void {
    this.sprite_index = 'spr_particle';
    this.vx = random_range(-3, 3);
    this.vy = random_range(-5, -2);
  }

  step(): void {
    // Move
    this.x += this.vx;
    this.y += this.vy;

    // Gravity
    this.vy += 0.2;

    // Fade out
    this.image_alpha = this.life / 60;

    // Lifetime
    this.life--;
    if (this.life <= 0) {
      instance_destroy.call(this);
    }
  }
}

// Create particles on event:
async createExplosion(x: number, y: number): Promise<void> {
  for (let i = 0; i < 10; i++) {
    await instance_create(x, y, 'obj_particle');
  }
}
```

Respawn System

Checkpoint and respawn:

```
export class obj_player extends GameObject {
  private respawnX: number = 100;
  private respawnY: number = 100;

  roomStart(): void {
    // Spawn at last checkpoint
    this.x = this.respawnX;
    this.y = this.respawnY;
  }

  step(): void {
    // Check for checkpoint
    const checkpoint = instance_place.call(this, this.x, this.y, 'obj_checkpoi
    if (checkpoint) {
      this.respawnX = checkpoint.x;
      this.respawnY = checkpoint.y;
      // Visual feedback
      show_debug_message.call(this, '✓ Checkpoint!');
    }

    // Fall off map = respawn
    if (this.y > room_height) {
      this.respawn();
    }

    // Die = respawn
    if (this.health <= 0) {
      this.respawn();
    }
  }

  private respawn(): void {
    this.x = this.respawnX;
    this.y = this.respawnY;
    this.health = 100;
    this.vspeed = 0;
    this.hspeed = 0;
  }
}
```

Save/Load System

Persistent data between sessions:

```
export class obj_game_manager extends GameObject {
  create(): void {
    this.persistent = true;
    this.loadGame();
  }

  public saveGame(): void {
    const saveData = {
      level: room_get_name(),
      score: (window as any).score || 0,
      playerHealth: 100, // Get from player object
      timestamp: Date.now()
    };

    localStorage.setItem('savegame', JSON.stringify(saveData));
    console.log('Game saved!');
  }

  public loadGame(): void {
    const saveDataStr = localStorage.getItem('savegame');
    if (!saveDataStr) {
      console.log('No save data found');
      return;
    }

    const saveData = JSON.parse(saveDataStr);
    (window as any).score = saveData.score;
    // Load other data...

    console.log('Game loaded!');
  }
}

// Save on specific events:
// obj_game_manager.saveGame(); // On checkpoint
// window.addEventListener('beforeunload', () => saveGame()); // On page close
```

Next Steps

- **04-gameobjects.md** - GameObject fundamentals
- **08-input.md** - Input handling
- **09-drawing.md** - Drawing techniques
- **20-api-gameobject.md** - Complete API reference

← Back to Index