GameObject API

Complete GameObject reference

Overview

The `GameObject` class is the base class for all game objects in Origami Engine. Every custom object extends `GameObject` and inherits its properties and methods.

Class Definition

```
import { GameObject } from 'origami-runtime';

export class obj_player extends GameObject {
  // Your custom properties and methods
}
```

Built-in Properties

Position Properties

Property	Type	Description
`x`	number	Current X position in room
`y`	number	Current Y position in room
`xprevious`	number	X position from previous frame
`yprevious`	number	Y position from previous frame
`xstart`	number	Starting X position (set in room)
`ystart`	number	Starting Y position (set in room)

Example:

```
step(): void {
  // Check if moved
  if (this.x !== this.xprevious || this.y !== this.yprevious) {
    show_debug_message.call(this, 'Moved!');
  }

  // Return to start position
  if (keyboard_check_pressed(vk_r)) {
    this.x = this.xstart;
    this.y = this.ystart;
  }
}
```

Motion Properties

Property	Type	Description
`speed`	number	Movement speed (pixels per frame)
`direction`	number	Movement direction (0-360 degrees, GMS-style)
`hspeed`	number	Horizontal speed component
`vspeed`	number	Vertical speed component

How motion works:

1. Engine converts `speed` and `direction` → `hspeed` and `vspeed`
2. Engine adds `hspeed` to `x` and `vspeed` to `y`
3. Happens automatically every frame

Direction:

- `0` = Right (→)
- `90` = Up (↑)
- `180` = Left (←)
- `270` = Down (↓)

Example:

```
create(): void {
  this.speed = 5;
  this.direction = 45; // Move diagonally up-right
}

step(): void {
  // Or set speeds directly
  this.hspeed = 3;
  this.vspeed = -2;
}
```

Sprite Properties

Property	Type	Description
`sprite_index`	string | null	Current sprite name
`image_index`	number	Current animation frame (auto-increments)
`image_speed`	number	Animation speed (frames per game frame)
`image_alpha`	number	Transparency (0.0 = invisible, 1.0 = opaque)
`image_angle`	number	Rotation angle in degrees
`image_xscale`	number	Horizontal scale (1.0 = normal, -1.0 = flipped)
`image_yscale`	number	Vertical scale (1.0 = normal)

Example:

```
create(): void {
  this.sprite_index = 'spr_player';
  this.image_speed = 1.0;  // Normal animation speed
  this.image_alpha = 1.0;  // Fully opaque
  this.image_angle = 0;    // No rotation
  this.image_xscale = 1.0; // Normal size
  this.image_yscale = 1.0;
}

step(): void {
  // Flip sprite based on movement
  if (this.hspeed > 0) this.image_xscale = 1;
  if (this.hspeed < 0) this.image_xscale = -1;

  // Rotate sprite
  this.image_angle += 5;

  // Fade out
  this.image_alpha -= 0.01;
}
```

Rendering Properties

Property	Type	Description
`visible`	boolean	Whether instance is drawn (default: true)
`depth`	number	Draw order (higher values draw behind)
`order`	number	Step execution order (lower executes first)

Depth:

- Higher depth = Behind
- Lower depth = In front
- Default: 0

Common depth values:

- `-100` - HUD/UI (always on top)
- `-10` - Player
- `0` - Default (enemies, items)
- `10` - Walls, platforms
- `100` - Background decorations

Example:

```
create(): void {
  this.depth = -10;  // Draw in front of most objects
  this.visible = true;
  this.order = 0;
}
```

Persistence

Property	Type	Description
`persistent`	boolean	Survives room transitions (default: false)

Example:

```
create(): void {
  this.persistent = true; // Don't destroy when changing rooms
}

roomStart(): void {
  // Reset position at start of new room
  this.x = 100;
  this.y = 100;
}
```

Event Methods

All event methods are **optional**. Override them to define custom behavior.

`create()`

Syntax: `create(): void`

Called: Once when instance is created

Use for:

- Setting initial properties
- Assigning sprites
- Initializing variables

Example:

```
create(): void {
  this.sprite_index = 'spr_player';
  this.speed = 4;
  this.health = 100;
  this.maxHealth = 100;
}
```

`step()`

Syntax: `step(): void`

Called: Every frame (60 times per second)

Use for:

- Movement logic
- Collision detection
- Game state updates
- Input handling
- AI logic

Example:

```
step(): void {
  // Movement
  if (keyboard_check(vk_d)) this.x += 4;
  if (keyboard_check(vk_a)) this.x -= 4;

  // Collision
  if (place_meeting.call(this, this.x, this.y, 'obj_wall')) {
    this.x = this.xprevious;
  }

  // Health check
  if (this.health <= 0) {
    instance_destroy.call(this);
  }
}
```

`draw()`

Syntax: `draw(): void`

Called: Every frame during rendering phase

Use for:

- Custom drawing

- Visual effects
- UI elements
- Debug visualization

Default behavior: If not defined, automatically calls `draw_self.call(this)`

Example:

```
draw(): void {
  // Draw the sprite
  draw_self.call(this);

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

  draw_set_color('#FF0000');
  draw_rectangle(
    this.x - 25,
    this.y - 35,
    this.x - 25 + (barWidth * healthPercent),
    this.y - 30,
    false
  );
  draw_set_color('#FFFFFF');
}
```

`gameStart()`

Syntax: `gameStart(): void`

Called: Once when the game first begins

Use for:

- One-time initialization
- Loading saved data
- Setting up persistent managers

Example:

```
gameStart(): void {
  // Load saved data
  const savedScore = localStorage.getItem('score');
  if (savedScore) {
    (window as any).score = parseInt(savedScore);
  }
}
```

`gameEnd()`

Syntax: `gameEnd(): void`

Called: When the game ends (via `game_end()`)

Use for:

- Cleanup
- Saving final state
- Resetting global variables

Example:

```
gameEnd(): void {
  // Save final score
  localStorage.setItem('score', (window as any).score.toString());
}
```

`roomStart()`

Syntax: `roomStart(): void`

Called: When entering a room (including first room)

Use for:

- Room-specific initialization
- Resetting position
- Loading level data

Example:

```
roomStart(): void {
  console.log('Entered room:', room_get_name());
  this.health = this.maxHealth; // Reset health
}
```

`roomEnd()`

Syntax: `roomEnd(): void`

Called: When leaving a room

Use for:

- Cleanup
- Saving room state
- Stopping sounds

Example:

```
roomEnd(): void {
  // Save checkpoint
  localStorage.setItem('lastRoom', room_get_name());
  localStorage.setItem('playerHealth', this.health.toString());
}
```

Execution Order

Each Frame

1. **Motion System**:
   - Convert `speed` / `direction` → `hspeed` / `vspeed`
   - Add `hspeed` to `x`, `vspeed` to `y`
   - Update `xprevious` / `yprevious`

2. **Step Events**:
    - Call `step()` on all instances (sorted by `order`)
3. **Animation System**:
    - Increment `image_index` based on `image_speed` and sprite FPS
4. **Destroy Marked Instances**:
    - Remove instances marked for destruction
5. **Draw Events**:
    - Sort instances by `depth` (high to low)
    - Call `draw()` or auto-draw sprite
    - Draw debug overlay (if enabled)

Custom Properties

Add your own properties:

```
export class obj_player extends GameObject {
  // Custom properties
  private health: number = 100;
  private maxHealth: number = 100;
  private coins: number = 0;
  private invincible: boolean = false;
  private invincibilityTimer: number = 0;

  // Use in events
  step(): void {
    if (this.invincibilityTimer > 0) {
      this.invincibilityTimer--;
    } else {
      this.invincible = false;
    }

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

State Machines

Use enums for state management:

```
enum PlayerState {
  Idle,
  Running,
  Jumping,
  Falling,
  Attacking
}

export class obj_player extends GameObject {
  private state: PlayerState = PlayerState.Idle;

  step(): void {
    switch (this.state) {
      case PlayerState.Idle:
        this.handleIdle();
        break;
      case PlayerState.Running:
        this.handleRunning();
        break;
      case PlayerState.Jumping:
        this.handleJumping();
        break;
      // ...
    }
  }

  private handleIdle(): void {
    if (keyboard_check(vk_d) || keyboard_check(vk_a)) {
      this.state = PlayerState.Running;
    }
    if (keyboard_check_pressed(vk_space)) {
      this.state = PlayerState.Jumping;
    }
  }

  // ... other state handlers
}
```

Best Practices

1. Use Private Members

```
// ✅ GOOD
private health: number = 100;

// ❌ BAD
health: number = 100; // Accessible from outside
```

2. Initialize in create()

```
// ✅ GOOD
create(): void {
  this.sprite_index = 'spr_player';
  this.speed = 4;
}

// ❌ BAD
// Relying on default values without explicit init
```

3. Clean Up Resources

```
roomEnd(): void {
  // Clear arrays, timers, references
  this.bullets = [];
  this.targets = [];
}
```

4. Use Type Safety

```
// ✅ GOOD
private health: number = 100;
private name: string = 'Player';

// ❌ BAD
private health: any = 100;
```

Next Steps

- **04-gameobjects.md** - GameObject usage guide
- **21-api-global-functions.md** - Global functions
- **40-common-patterns.md** - Design patterns

← Back to Index