Global Functions API

Instance, room, and game functions

Overview

Global functions are available throughout your game code. Most instance-related functions require `.call(this)` to specify which instance is calling them.

Instance Functions

`instance_create()`

Creates a new instance of an object.

Syntax: `await instance_create(x, y, objectType)`

Arguments:

- `x` (number) - X position
- `y` (number) - Y position
- `objectType` (string) - Object class name

Returns: `Promise<GameObject>` - The created instance

Example:

```
async step(): Promise<void> {
  if (keyboard_check_pressed(vk_space)) {
    const bullet = await instance_create(this.x, this.y, 'obj_bullet');
    bullet.direction = this.image_angle;
    bullet.speed = 10;
  }
}
```

Note: Must be awaited. The instance's `create()` event is called immediately.

`instance_destroy()`

Destroys an instance.

Syntax: `instance_destroy.call(this)` or `instance_destroy.call(otherInstance)`

Arguments: None

Returns: `void`

Example:

```
step(): void {
  // Destroy self
  if (this.health <= 0) {
    instance_destroy.call(this);
  }

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

`instance_exists()`

Checks if any instance of a type exists.

Syntax: `instance_exists(objectType)`

Arguments:

- `objectType` (string) - Object class name

Returns: `boolean` - True if at least one exists

Example:

```
step(): void {
  // Check if player still exists
  if (!instance_exists('obj_player')) {
    game_restart();
  }

  // Proceed to next level when all enemies defeated
  if (!instance_exists('obj_enemy')) {
    await room_goto('room_level2');
  }
}
```

`instance_find()`

Gets the nth instance of a type.

Syntax: `instance_find(objectType, n)`

Arguments:

- `objectType` (string) - Object class name
- `n` (number) - Index (0-based)

Returns: `GameObject | null` - The instance or null

Example:

```
step(): void {
  // Get first player instance
  const player = instance_find('obj_player', 0);
  if (player) {
    const dist = point_distance(this.x, this.y, player.x, player.y);
    if (dist < 100) {
      // Chase player
      move_towards_point.call(this, player.x, player.y, 3);
    }
  }
}
```

Note: Order is not guaranteed. Don't rely on specific instance order.

`instance_number()`

Counts instances of a type.

Syntax: `instance_number(objectType)`

Arguments:

- `objectType` (string) - Object class name

Returns: `number` - Count of instances

Example:

```
step(): void {
  const enemyCount = instance_number('obj_enemy');
  const coinCount = instance_number('obj_coin');

  // Victory condition
  if (enemyCount === 0 && coinCount === 0) {
    await room_goto('room_victory');
  }

  // Limit spawns
  if (instance_number('obj_particle') < 100) {
    await instance_create(this.x, this.y, 'obj_particle');
  }
}
```

`instance_position()`

Gets instance at exact point.

Syntax: `instance_position(x, y, objectType)`

Arguments:

- `x` (number) - X coordinate

- `y` (number) - Y coordinate
- `objectType` (string) - Object type

Returns: `GameObject | null` - Instance at that point or null

Example:

```
step(): void {
  // Check what's at mouse position
  const clicked = instance_position(mouse_x, mouse_y, 'obj_clickable');
  if (clicked && mouse_check_button_pressed(mb_left)) {
    instance_destroy.call(clicked);
  }
}
```

Room Functions

`room_goto()`

Transitions to another room.

Syntax: `await room_goto(roomName)`

Arguments:

- `roomName` (string) - Name of room to go to

Returns: `Promise<void>`

Description:

1. Calls `roomEnd()` on all instances
2. Destroys non-persistent instances
3. Loads new room
4. Creates room instances
5. Calls `roomStart()` on all instances

Example:

```
async step(): Promise<void> {
  // Door to next level
  const door = instance_place.call(this, this.x, this.y, 'obj_door');
  if (door && keyboard_check_pressed(vk_up)) {
    await room_goto('room_level2');
  }

  // Fall off map = restart
  if (this.y > room_height) {
    await room_goto(room_get_name());
  }
}
```

`room_get_name()`

Gets current room name.

Syntax: `room_get_name()`

Returns: `string` - Current room name

Example:

```
draw(): void {
  draw_text(10, 10, `Level: ${room_get_name()}`);
}
```

Game Functions

`game_end()`

Stops the game.

Syntax: `game_end()`

Arguments: None

Returns: `void`

Example:

```
step(): void {
  if (this.lives <= 0) {
    game_end();
  }

  if (keyboard_check_pressed(vk_escape)) {
    game_end();
  }
}
```

Note: Calls `gameEnd()` on all instances before stopping.

`game_restart()`

Restarts the game from the beginning.

Syntax: `await game_restart()`

Arguments: None

Returns: `Promise<void>`

Example:

```
async step(): Promise<void> {
  if (this.health <= 0) {
    await game_restart();
  }

  if (keyboard_check_pressed(vk_r)) {
    await game_restart();
  }
}
```

Note: Reloads the starting room and resets all instances.

Storage Functions

`game_save()`

Saves game data to localStorage.

Syntax: `game_save(slot)`

Arguments:

- `slot` (string | number) - Save slot identifier

Returns: `boolean` - True if successful

Description: Saves to browser localStorage. You must implement custom serialization.

Example:

```
step(): void {
  if (keyboard_check_pressed(vk_f5)) {
    // Save custom data
    const saveData = {
      level: room_get_name(),
      score: (window as any).score,
      health: this.health,
      timestamp: Date.now()
    };
    localStorage.setItem('saveData', JSON.stringify(saveData));

    if (game_save('slot1')) {
      show_debug_message.call(this, 'Game saved!');
    }
  }
}
```

`game_load()`

Loads game data from localStorage.

Syntax: `game_load(slot)`

Arguments:

- `slot` (string | number) - Save slot identifier

Returns: `boolean` - True if successful

Example:

```
create(): void {
  if (game_load('slot1')) {
    const saveData = JSON.parse(localStorage.getItem('saveData') || '{}');
    (window as any).score = saveData.score || 0;
    this.health = saveData.health || 100;

    if (saveData.level) {
      await room_goto(saveData.level);
    }
  }
}
```

`game_save_exists()`

Checks if a save exists.

Syntax: `game_save_exists(slot)`

Arguments:

- `slot` (string | number) - Save slot identifier

Returns: `boolean` - True if save exists

Example:

```
create(): void {
  if (game_save_exists('slot1')) {
    // Show "Continue" option
    this.showContinueButton = true;
  }
}
```

`game_save_delete()`

Deletes a save.

Syntax: `game_save_delete(slot)`

Arguments:

- `slot` (string | number) - Save slot identifier

Returns: `boolean` - True if successful

Example:

```
step(): void {
  if (keyboard_check_pressed(vk_delete)) {
    game_save_delete('slot1');
    localStorage.removeItem('saveData');
    show_debug_message.call(this, 'Save deleted');
  }
}
```

Random Functions

`random()`

Returns random float between 0 and n.

Syntax: `random(n)`

Arguments:

- `n` (number) - Maximum value (exclusive)

Returns: `number` - Random value [0, n)

Example:

```
create(): void {
  // Random speed between 0 and 5
  this.speed = random(5);

  // Random position
  this.x = random(room_width);
  this.y = random(room_height);

  // Random chance (50%)
  if (random(1) < 0.5) {
    this.sprite_index = 'spr_red';
  } else {
    this.sprite_index = 'spr_blue';
  }
}
```

`irandom()`

Returns random integer between 0 and n (inclusive).

Syntax: `irandom(n)`

Arguments:

- `n` (number) - Maximum value (inclusive)

Returns: `number` - Random integer [0, n]

Example:

```
create(): void {
  // Random dice roll (1-6)
  const roll = irandom(5) + 1;

  // Random health (50-100)
  this.health = 50 + irandom(50);

  // Random choice
  const choices = ['red', 'green', 'blue'];
  const choice = choices[irandom(choices.length - 1)];
}
```

`random_range()`

Returns random float between min and max.

Syntax: `random_range(min, max)`

Arguments:

- `min` (number) - Minimum value
- `max` (number) - Maximum value

Returns: `number` - Random value [min, max]

Example:

```
create(): void {
  // Enemy speed between 2 and 5
  this.speed = random_range(2, 5);

  // Spawn in specific area
  this.x = random_range(100, 500);
  this.y = random_range(100, 300);

  // Random timer
  this.attackDelay = random_range(60, 180); // 1-3 seconds
}
```

Debug Functions

`show_debug_message()`

Logs message to console with object name.

Syntax: `show_debug_message.call(this, message)`

Arguments:

- `message` (string) - Message to log

Returns: `void`

Example:

```
step(): void {
  show_debug_message.call(this, `Position: ${this.x}, ${this.y}`);
  show_debug_message.call(this, `Health: ${this.health}`);
  show_debug_message.call(this, `State: ${this.state}`);
}
```

Output (browser console):

```
[obj_player] Position: 150, 200
[obj_player] Health: 95
[obj_player] State: running
```

Best practice: Remove or disable debug messages in production:

```
private readonly DEBUG = false;

step(): void {
  if (this.DEBUG) {
    show_debug_message.call(this, 'Debug info');
  }
}
```

Global Variables

These variables are automatically available:

Room Variables

- `room_width` (number) - Current room width in pixels
- `room_height` (number) - Current room height in pixels
- `room_speed` (number) - Current room FPS (usually 60)

Example:

```
step(): void {
  // Wrap around screen
  if (this.x > room_width) this.x = 0;
  if (this.x < 0) this.x = room_width;
  if (this.y > room_height) this.y = 0;
  if (this.y < 0) this.y = room_height;
}
```

Mouse Variables

- `mouse_x` (number) - Mouse X position in room coordinates
- `mouse_y` (number) - Mouse Y position in room coordinates

Example:

```
step(): void {
  // Point at mouse
  this.image_angle = point_direction(this.x, this.y, mouse_x, mouse_y);

  // Move to mouse on click
  if (mouse_check_button_pressed(mb_left)) {
    this.targetX = mouse_x;
    this.targetY = mouse_y;
  }
}
```

View Variables

Arrays indexed by view number (usually 0):

- `view_xview[0]` (number) - View X position in room
- `view_yview[0]` (number) - View Y position in room
- `view_wview[0]` (number) - View width
- `view_hview[0]` (number) - View height

Example:

```
draw(): void {
  // Draw UI relative to view
  const uiX = view_xview[0] + 10;
  const uiY = view_yview[0] + 10;
  draw_text(uiX, uiY, `Score: ${this.score}`);
}
```

Common Patterns

Spawner Pattern

```
export class obj_spawner extends GameObject {
  private spawnTimer: number = 0;
  private readonly SPAWN_DELAY = 120;

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

    if (this.spawnTimer >= this.SPAWN_DELAY) {
      // Spawn enemy at random position
      const x = random_range(this.x - 50, this.x + 50);
      const y = this.y;
      await instance_create(x, y, 'obj_enemy');

      this.spawnTimer = 0;
    }
  }
}
```

Manager Pattern

```
export class obj_game_manager extends GameObject {
  create(): void {
    this.persistent = true; // Survive room changes
    this.visible = false;   // Don't draw
    (window as any).score = 0;
  }

  step(): void {
    // Global game logic
    if (instance_number('obj_coin') === 0) {
      await room_goto('room_victory');
    }
  }

  draw(): void {
    // Draw HUD
    draw_text(10, 10, `Score: ${(window as any).score}`);
  }
}
```

Next Steps

- **20-api-gameobject.md** - GameObject API
- **22-api-drawing.md** - Drawing functions
- **23-api-collision.md** - Collision functions

← Back to Index