

# Storage API

---

## Save/load functions using localStorage

---

## Overview

---

Origami Engine provides functions to save and load game data using browser localStorage. Data persists between sessions.

---

## Save 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:** Triggers a save event. You must implement custom serialization by manually saving data to localStorage.

### Example:

```
step(): void {
  if (keyboard_check_pressed(vk_f5)) {
    // Prepare save data
    const saveData = {
      level: room_get_name(),
      score: (window as any).score || 0,
      health: this.health,
      maxHealth: this.maxHealth,
      coins: this.coins,
      position: {
        x: this.x,
        y: this.y
      },
      timestamp: Date.now()
    };

    // Save to localStorage
    localStorage.setItem('saveData', JSON.stringify(saveData));

    // Trigger game save event
    if (game_save('slot1')) {
      show_debug_message.call(this, '✓ Game saved!');
    }
  }
}
```

---

## Multiple Save Slots

```
private currentSlot: number = 1;

saveGame(slot: number): void {
  const saveData = {
    level: room_get_name(),
    player: {
      health: this.health,
      x: this.x,
      y: this.y
    },
    inventory: this.inventory,
    flags: this.gameFlags,
    timestamp: Date.now()
  };

  // Save to specific slot
  localStorage.setItem(`saveSlot${slot}`, JSON.stringify(saveData));

  if (game_save(`slot${slot}`)) {
    console.log(`Saved to slot ${slot}`);
  }
}
```

---

## Load Functions

---

`game_load()`

Loads game data from localStorage.

**Syntax:** `game_load(slot)`

**Arguments:**

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

**Returns:** `boolean` - True if successful

**Description:** Triggers a load event. You must implement custom deserialization by manually reading data from localStorage.

**Example:**

```
create(): void {
  if (game_load('slot1')) {
    const saveDataStr = localStorage.getItem('saveData');
    if (saveDataStr) {
      const saveData = JSON.parse(saveDataStr);

      // Restore game state
      (window as any).score = saveData.score;
      this.health = saveData.health;
      this.maxHealth = saveData.maxHealth;
      this.coins = saveData.coins;
      this.x = saveData.position.x;
      this.y = saveData.position.y;

      // Go to saved level
      if (saveData.level) {
        await room_goto(saveData.level);
      }

      show_debug_message.call(this, '✓ Game loaded!');
    }
  }
}
```

---

## Load with Error Handling

```
loadGame(slot: number): boolean {
  try {
    const saveDataStr = localStorage.getItem(`saveSlot${slot}`);
    if (!saveDataStr) {
      console.log('No save data found');
      return false;
    }

    const saveData = JSON.parse(saveDataStr);

    // Validate save data
    if (!saveData.level || !saveData.player) {
      console.error('Invalid save data');
      return false;
    }

    // Restore state
    this.health = saveData.player.health;
    this.x = saveData.player.x;
    this.y = saveData.player.y;
    this.inventory = saveData.inventory || [];
    this.gameFlags = saveData.flags || {};

    // Go to saved room
    await room_goto(saveData.level);

    return game_load(`slot${slot}`);
  } catch (error) {
    console.error('Failed to load game:', error);
    return false;
  }
}
```

---

# Check Functions

---

`game_save_exists()`

Checks if a save file exists.

**Syntax:** `game_save_exists(slot)`

**Arguments:**

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

**Returns:** `boolean` - True if save exists

**Example:**

```
create(): void {
    // Check for existing save
    if (game_save_exists('slot1')) {
        this.showContinueButton = true;
    } else {
        this.showNewGameButton = true;
    }
}

draw(): void {
    if (this.showContinueButton) {
        draw_text(100, 100, 'Press C to Continue');
    }
    draw_text(100, 130, 'Press N for New Game');
}

step(): void {
    if (keyboard_check_pressed(vk_c) && game_save_exists('slot1')) {
        this.loadGame(1);
    }
    if (keyboard_check_pressed(vk_n)) {
        this.startNewGame();
    }
}
```

## List All Saves

```
getAllSaves(): SaveInfo[] {
    const saves: SaveInfo[] = [];

    for (let i = 1; i <= 3; i++) {
        if (game_save_exists(`slot${i}`)) {
            const saveDataStr = localStorage.getItem(`saveSlot${i}`);
            if (saveDataStr) {
                const saveData = JSON.parse(saveDataStr);
                saves.push({
                    slot: i,
                    level: saveData.level,
                    timestamp: saveData.timestamp,
                    score: saveData.score
                });
            }
        }
    }

    return saves;
}
```

---

## Delete Functions

---

`game_save_delete()`

Deletes a save file.

**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)) {
    if (game_save_delete('slot1')) {
      localStorage.removeItem('saveData');
      show_debug_message.call(this, '✓ Save deleted');
    }
  }
}
```

---

## Delete with Confirmation

```
deleteSave(slot: number): void {
  // Confirm deletion
  const confirmed = confirm(`Delete save slot ${slot}?`);
  if (!confirmed) return;

  // Delete from localStorage
  localStorage.removeItem(`saveSlot${slot}`);

  // Trigger game delete event
  if (game_save_delete(`slot${slot}`)) {
    console.log(`Slot ${slot} deleted`);
  }
}
```

# Save Data Patterns

---

## Basic Player Data

```
interface SaveData {
  version: string;
  timestamp: number;
  level: string;
  player: {
    health: number;
    maxHealth: number;
    x: number;
    y: number;
    coins: number;
  };
}

savegame(): void {
  const saveData: SaveData = {
    version: '1.0.0',
    timestamp: Date.now(),
    level: room_get_name(),
    player: {
      health: this.health,
      maxHealth: this.maxHealth,
      x: this.x,
      y: this.y,
      coins: this.coins
    }
  };

  localStorage.setItem('gameData', JSON.stringify(saveData));
  game_save('main');
}
```

---

## Inventory System

```
interface InventoryItem {
  id: string;
  quantity: number;
}

saveInventory(): void {
  const saveData = {
    inventory: this.inventory.map(item => ({
      id: item.id,
      quantity: item.quantity
    })),
    equipped: {
      weapon: this.equippedWeapon,
      armor: this.equippedArmor
    }
  };

  localStorage.setItem('inventory', JSON.stringify(saveData));
}

loadInventory(): void {
  const data = localStorage.getItem('inventory');
  if (data) {
    const saveData = JSON.parse(data);
    this.inventory = saveData.inventory;
    this.equippedWeapon = saveData.equipped.weapon;
    this.equippedArmor = saveData.equipped.armor;
  }
}
```

---

## Progress Flags

```
interface GameFlags {
  [key: string]: boolean;
}

private flags: GameFlags = {};

setFlag(flag: string, value: boolean = true): void {
  this.flags[flag] = value;
  this.saveFlags();
}

hasFlag(flag: string): boolean {
  return this.flags[flag] || false;
}

saveFlags(): void {
  localStorage.setItem('gameFlags', JSON.stringify(this.flags));
}

loadFlags(): void {
  const data = localStorage.getItem('gameFlags');
  if (data) {
    this.flags = JSON.parse(data);
  }
}

// Usage
step(): void {
  const door = instance_place.call(this, this.x, this.y, 'obj_boss_door');
  if (door && this.hasFlag('defeatedBoss')) {
    door.locked = false;
  }
}
```

---

## Checkpoint System

```
private lastCheckpoint: {
  room: string;
  x: number;
  y: number;
} | null = null;

saveCheckpoint(): void {
  this.lastCheckpoint = {
    room: room_get_name(),
    x: this.x,
    y: this.y
  };

  localStorage.setItem('checkpoint', JSON.stringify(this.lastCheckpoint));
  show_debug_message.call(this, '✓ Checkpoint saved');
}

loadCheckpoint(): void {
  const data = localStorage.getItem('checkpoint');
  if (data) {
    this.lastCheckpoint = JSON.parse(data);

    if (this.lastCheckpoint) {
      await room_goto(this.lastCheckpoint.room);
      this.x = this.lastCheckpoint.x;
      this.y = this.lastCheckpoint.y;
    }
  }
}

// Save checkpoint when touching checkpoint object
step(): void {
  const checkpoint = instance_place.call(this, this.x, this.y, 'obj_checkpoint');
  if (checkpoint && !checkpoint.activated) {
    checkpoint.activated = true;
    this.saveCheckpoint();
  }
}
```

---

## Auto-Save

---

```
private autoSaveTimer: number = 0;
private readonly AUTO_SAVE_INTERVAL = 3600; // 60 seconds at 60 FPS

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

    if (this.autoSaveTimer >= this.AUTO_SAVE_INTERVAL) {
        this.autoSave();
        this.autoSaveTimer = 0;
    }
}

autoSave(): void {
    const saveData = {
        level: room_get_name(),
        player: {
            health: this.health,
            x: this.x,
            y: this.y
        },
        timestamp: Date.now()
    };

    localStorage.setItem('autoSave', JSON.stringify(saveData));
    console.log('Auto-saved');
}
```

---

## Save on Exit

---

```
create(): void {
  // Save when closing browser
  window.addEventListener('beforeunload', () => {
    this.quickSave();
  });
}

quickSave(): void {
  const saveData = {
    level: room_get_name(),
    health: this.health,
    x: this.x,
    y: this.y,
    timestamp: Date.now()
  };

  localStorage.setItem('quickSave', JSON.stringify(saveData));
}
```

---

# Versioning

---

```
private readonly SAVE_VERSION = '1.0.0';

savegame(): void {
  const saveData = {
    version: this.SAVE_VERSION,
    // ... other data
  };

  localStorage.setItem('saveData', JSON.stringify(saveData));
}

loadGame(): boolean {
  const data = localStorage.getItem('saveData');
  if (!data) return false;

  const saveData = JSON.parse(data);

  // Check version compatibility
  if (saveData.version !== this.SAVE_VERSION) {
    console.warn('Save version mismatch');
    return this.migrateOldSave(saveData);
  }

  // Load normally
  return true;
}

migrateOldSave(oldData: any): boolean {
  // Migrate old save to new format
  const newData = {
    version: this.SAVE_VERSION,
    // Convert old data to new format
  };

  localStorage.setItem('saveData', JSON.stringify(newData));
  return true;
}
```

---

## Storage Limits

---

### Check Available Space

```
getStorageUsage(): { used: number; total: number } {  
  let total = 0;  
  for (let key in localStorage) {  
    if (localStorage.hasOwnProperty(key)) {  
      total += localStorage[key].length + key.length;  
    }  
  }  
  
  return {  
    used: total,  
    total: 5 * 1024 * 1024 // ~5MB typical limit  
  };  
}
```

---

### Compress Save Data

```
// Simple compression using JSON.stringify with no whitespace  
savegame(): void {  
  const saveData = {  
    // ... your data  
  };  
  
  // Compact JSON (no whitespace)  
  const compacted = JSON.stringify(saveData);  
  
  localStorage.setItem('saveData', compacted);  
}
```

---

# Best Practices

---

1. **Always use try-catch:** localStorage can fail
  2. **Version your saves:** Include version number
  3. **Validate loaded data:** Check for required fields
  4. **Clear old saves:** Remove deprecated data
  5. **Test save/load:** Regularly test the full cycle
  6. **Provide feedback:** Show "Saving..." messages
  7. **Don't save every frame:** Use timers or specific events
- 

## Common Issues

---

### localStorage Not Available

```
isLocalStorageAvailable(): boolean {
  try {
    const test = '__test__';
    localStorage.setItem(test, test);
    localStorage.removeItem(test);
    return true;
  } catch (e) {
    return false;
  }
}

// Usage
if (!this.isLocalStorageAvailable()) {
  console.warn('localStorage not available');
  // Use alternative storage or disable saves
}
```

---

## Quota Exceeded

```
savegame(): void {  
  try {  
    const saveData = JSON.stringify(this.gameState);  
    localStorage.setItem('saveData', saveData);  
  } catch (e) {  
    if (e.name === 'QuotaExceededError') {  
      console.error('Storage quota exceeded');  
      // Clear old data or compress  
    }  
  }  
}
```

---

## Next Steps

- [21-api-global-functions.md](#) - Global functions
- [40-common-patterns.md](#) - Save/load patterns
- [41-deployment.md](#) - Deployment considerations

---

[← Back to Index](#)