sprite.js Documentation

Release 1.2.1

Batiste Bieler

Apr 22, 2022

Contents

1 Download sprite.js 3

2 Examples 5

3 Table of content 7

3.1 How to use sprite.js? 7
3.2 Performance and different ways to draw 8
3.3 Troubleshooting 8
3.4 Sprite.js API documentation 8
3.5 Changelog 20

4 Projects that Use Sprite.js 21

Index 23

Sprite.js is a framework that lets you create animations and games using sprites in an efficient way. The goal is to allow a common framework for desktop and mobile browsers and use the latest technology available on each platform. Sprite.js is tested on WebKit, Firefox, Android phones, Opera and IE9.

CHAPTER 1

Download sprite.js

To download the latest version of sprite.js go to the Github repository or use the direct link to the library.

CHAPTER 2

Examples

To see examples of what the framework has to offer have a look at the test files.

CHAPTER 3

Table of content

3.1 How to use sprite.js?

Example of basic sprite transformations:

```
// create the Scene object
var scene = sjs.Scene({w:640, h:480});

// load the images in parallel. When all the images are
// ready, the callback function is called.
scene.loadImages(['character.png'], function() {

    // create the Sprite object;
    var sp = scene.Sprite('character.png');

    // change the visible size of the sprite
    sp.size(55, 30);

    // apply the latest visual changes to the sprite
    // (draw if canvas, update attribute if DOM);
    sp.update();

    // change the offset of the image in the sprite
    // (this works the opposite way of a CSS background)
    sp.offset(50, 50);

    // various transformations
    sp.move(100, 100);
    sp.rotate(3.14 / 4);
    sp.scale(2);
    sp.setOpacity(0.8);

    sp.update();
});
```

If you want a more interactive demonstration of basic sprite manipulation, there is a good example in the tests: http://batiste.github.io/sprite.js/tests/visual_guide.html

3.2 Performance and different ways to draw

This library provides three rendering engines: HTML and canvas and WebGl. Please note that WebGl is still in an experimental state.

By default the HTML engine is used. The HTML engine displays sprites using DOM elements, while the canvas engine draw the sprites on the canvas. Each layer of the application can have a different engine. This enable you to mix the two techniques if needed.

To make use of canvas with a layer you need to specify it in the options:

```
var background = scene.Layer('background', {useCanvas:true});
```

The canvas will be automaticaly cleared by the game ticker. If you don't need it you can set the autoClear to false when building a layer:

```
var background = scene.Layer('background', {useCanvas:true, autoClear:false});
```

Performances with the particle test vary widely depending on the device, browser and platform:

Browsers	Chrome linux	Opera linux	Firefox linux	HTC Desire (webkit)	IE9
HTML engine	2000	60	500	120	30
Canvas engine	1300	100	300	80	600

3.3 Troubleshooting

- When using canvas, I get an "Uncaught Error: INDEX_SIZE_ERR: DOM Exception 1" in updateCanvas*

This error appears when canvas try to read an image out of the boundary of the image itself. Check that your cycle doesn't go off the boundaries, and that the size and offset are correct.

For any other undocumented issue, please fill a bug on this page: https://github.com/batiste/sprite.js/issues

3.4 Sprite.js API documentation

This documentation does not document all of Sprite.js features, but only those that are considered stable.

3.4.1 Scene object

The scene object is a DOM container where all the Sprites will be drawn. You always need to start by creating a Scene object.

```
// create a Scene object
var scene = sjs.Scene({w:640, h:480});
```

class sjs.**Scene**(*options*)
    Create a new Scene.

Options details:

`Scene.options.`**`parent`**
DOM parent of the scene. The default is document.body.

`Scene.options.`**`w`**
Width of the scene.

`Scene.options.`**`h`**
Height of the scene.

`Scene.options.`**`autoPause`**
Pause the scene when the user quit the current window. The default is true.

This is the list of the different methods available on the Scene object:

`Scene.`**`loadImages`**`(`*images*, *callback*`)`

Arguments

- **`images`** (*`Array`*) – An array of image sources.
- **`callback`** (*`function`*) – This gets called once all images are loaded.

Load the given array of image sources. When all images are loaded, the callback is executed.

`Scene.`**`reset`**`()`
Delete all layers present in this scene, delete the sprites and layers, and pause the ticker.

`class` `Scene.`**`Layer`**`(`*name*[, *options*]`)`
Create a Layer object, see *the Layer section*.

`class` `Scene.`**`Sprite`**`(`[*source*, *layer|options*]`)`

Arguments

- **`source`** (*`string`*) – source of the image.
- **`layer`** (*`Layer`*) – the layer object.
- **`options`** (*`object`*) – options.

Create a Sprite object, see *the Sprite section*.

`class` `Scene.`**`Ticker`**`(`*paint*, *options*`)`

Arguments

- **`paint`** (*`function`*) – Gets called on every game tick. The ticker is passed as first paramater.
- **`options`** (*`object`*) – The possible options, see *the Ticker section*.

Create a Ticker object for this scene or reset the previous one.

`class` `Scene.`**`Cycle`**`(`*triplets*`)`

Arguments

- **`triplets`** (*`Array`*) – The triplets array.

Alias for sjs.Cycle, see *the Cycle section*.

`class` `Scene.`**`Input`**`()`
Alias for sjs.Input, see *the Input section*.

3.4.2 Sprite object

To create a sprite you should use the Scene.Sprite constructor:

class Scene.**Sprite**([*src*, *options*])

Arguments

- **src** – source of the sprite's image.
- **options** (*object*) – possible sprite options.

A Sprite can be instantiated from different objects and with a range of options. eg:

```
var foreground = scene.Layer("foreground");
var player = scene.Sprite("player.png", foreground);
```

You can create the sprite using the Layer directly instead of the Scene:

```
var foreground = scene.Layer("foreground");
var player = layer.Sprite("player.png");
```

You can also initialize any Sprite properties by passing an options object instead of the Layer object, eg:

```
var options = {layer:layer, x:10, size:[20, 20], y:15};
var sprite = scene.Sprite("mysprite.png", options);
```

All parameters are optional. If the layer is not specified, the default layer will be used. If you want to set the layer but not any image you can do so like this:

```
var sprite = scene.Sprite(false, {layer:layer, color:"#f11"});
```

Important methods

To update any visual changes to the view you should call the *update* method:

Sprite.**update**()

Apply the latest changes to the sprite's layer.

Sprite.**loadImg**(*source*[, *resetSize*])

Arguments

- **source** (*string*) – new image source.
- **resetSize** (*boolean*) – if true the size of the sprite will be reset by the new image.

Change the image sprite.

Sprite.**remove**()

Remove the DOM element if the HTML engine is used and enable the garbage collection of the object.

Sprite.**canvasUpdate**(*layer*)

Draw the sprite on a given Canvas layer. This doesn't work with an HTML layer.

With a canvas engine, the surface will be automatically cleared before each game tick. You will need to call update to draw the sprite on the canvas again. If you don't want to do this you can set the layer autoClear attribute to false.

Read only attributes

For technichal and performance reasons a Sprite's attributes needs to be changed using a setters method. The following attributes are *READ ONLY*:

`Sprite.`**`x`**
The horizontal position of the sprite as measured against the top left corner of the scene.

`Sprite.`**`y`**
The vertical position of the sprite as measured top left corder of the scene.

`Sprite.`**`w`**
Controls the horizontal visible surface of the image. To have a repeating sprite background you can set the width or height value bigger than the size of the image.

`Sprite.`**`h`**
Controls the vertical visible surface of the image.

`Sprite.`**`xoffset`**
The horizontal offset within the sprite's image from where to start painting the sprite's surface.

`Sprite.`**`yoffset`**
The verical offset within the sprite's from where to start painting the sprite's surface.

`Sprite.`**`xscale`**
Horizontal scaling.

`Sprite.`**`yscale`**
Vertical scaling.

`Sprite.`**`angle`**
Rotation of the sprite in radians.

`Sprite.`**`color`**
Background color of the sprite. Use the rgb/hexadecimal CSS notation.

Setters

If you want to change any of those attributes use the following setters:

`Sprite.`**`setX`**(*x*)

`Sprite.`**`setY`**(*y*)

`Sprite.`**`setW`**(*w*)

`Sprite.`**`setH`**(*h*)

`Sprite.`**`setXOffset`**(*x*)

`Sprite.`**`setYOffset`**(*y*)

`Sprite.`**`setXScale`**(*xscale*)

`Sprite.`**`setYScale`**(*yscale*)

`Sprite.`**`setAngle`**(*radian*)

`Sprite.`**`setColor`**(*'#333'*)

`Sprite.`**`setOpacity`**(*0.5*)

`Sprite.`**`setBackgroundRepeat`**(*"repeat-y"*)

Or one of those helper methods:

`Sprite.`**`rotate`**(*radians*)

`Sprite.`**`scale`**(*xscale*[, *yscale*])
: If y is not defined, y will take the same value as x.

`Sprite.`**`move`**(*x*, *y*)
: Move the sprite in the direction of the vector (x, y) argument.

`Sprite.`**`position`**(*x*, *y*)
: Set the position of the sprite (left, top)

`Sprite.`**`offset`**(*x*, *y*)

`Sprite.`**`size`**(*w*, *h*)
: Set the width and height of the visible sprite.

Physics Engine

Sprites have methods to help you implement a basic physics engine:

`Sprite.`**`xv`**
: Horizontal velocity.

`Sprite.`**`yv`**
: Vertical velocity.

`Sprite.`**`rv`**
: Radial velocity

`Sprite.`**`applyVelocity`**()
: Apply all velocities on the current Sprite.

`Sprite.`**`reverseVelocity`**()
: Reverse all velocities on the current Sprite.

`Sprite.`**`applyXVelocity`**()
: Apply the horizontal xv velocity.

`Sprite.`**`applyYVelocity`**()
: Apply the vertical yv velocity.

`Sprite.`**`reverseXVelocity`**()
: Apply the horizontal xv velocity negatively.

`Sprite.`**`reverseYVelocity`**()
: Apply the vertical yv velocity negatively.

`Sprite.`**`rotateVelocity`**(*angle*)
: Rotate the velocity vector according to the provided angle.

`Sprite.`**`orientVelocity`**(*x*, *y*)
: Point the velocity vector in the direction of the point (x, y). The velocity intensity remains unchanged.

`Sprite.`**`distance`**(*sprite*)
: Returns the distance between the calling sprite's center and it's argument sprites center.

`Sprite.`**`distance`**(*x*, *y*)
: Return the distance between the sprite's center and the point (x, y)

Collision Detection

These methods are not included in the sprite.js core and needs to be loaded indenpendantly:

```
<script src="lib/collision.js"></script>
```

`Sprite.`**`isPointIn`**`(`*x*, *y*`)`
Returns true if the point (x, y) is within the sprite's surface.

`Sprite.`**`collidesWith`**`(`*sprite*`)`
Returns true if the sprite is in collision with the passed sprite

`Sprite.`**`collidesWithArray`**`(`*sprites*`)`

Arguments

- **`sprites`** (*`Array`*) – An array of Sprite objects.

Searches the passed array of sprites for a colliding sprite. If found, that sprite is returned.

Special Effects

There are two methods useful for creating special effects. You can use explode2 to separate the current sprite in two parts:

`Sprite.`**`explode2`**`(`[*position*, *horizontal=true*, *layer*]`)`

Arguments

- **`position`** (*`number`*) – The cut offset / position.
- **`horizontal`** (*`boolean`*) – Cut horizontaly if true, verticaly if false.

Returns an array of two new sprites that are the two parts of the sprite according to the given position. The default value for position is half the size of the sprite.

`Sprite.`**`explode4`**`(`[*x*, *y*, *layer*]`)`

Arguments

- **`x`** (*`number`*) – The x position where to cute.
- **`y`** (*`number`*) – The y position where to cute.
- **`layer`** (*`Layer`*) – the Layer where to create the new Sprites, default being the current sprite's Layer.

Return an array of four new sprites that are the split from the center (x, y). The default value for (x, y) is the center of the sprite.

3.4.3 List Object

A List is an iterable object that simplifies the management of an sprite (or any other object) array:

`class` `sjs.`**`List`**`(`[*objects*]`)`

Arguments

- **`objects`** (*`Array`*) – An array of objects.

Create the object list.

`List.`**`add`**`(`*object* | *objects*`)`
Add an object or an array of objects to the list.

`List.`**`remove`**`(`*object*`)`
Remove the first matching object from the list.

`List.`**`iterate`**`()`
Returns an object and increment the pointer. Returns false at the end of the list.

`List.list.`**`length`**
The length of the list.

`List.`**`list`**
Returns the underlying array that the List is managing.

Example of use:

```
var crates = sjs.List([crate1, crate2]);

var crate;
while(crate = crates.iterate()) {
    crate.applyVelocity();
    if(crate.y > 200) {
        // remove it from the list
        crates.remove(crate);
        // remove it from the DOM
        crate.remove();
    }
}
```

3.4.4 Ticker Object

Keeping track of time in javascript can be difficult. Sprite.js provides a Ticker object to deal with this issue.

A Ticker is an object that keeps track of time properly, so it's straight forward to render the changes in the scene. The Ticker gives accurate ticks. A game tick is the time between every Sprites/Physics update in your engine. To setup a ticker:

```
function paint(ticker) {

    myCycles.next(ticker.lastTicksElapsed);
    // do your animation and physics here

}
var ticker = scene.Ticker(paint);
```

`class` `Scene.`**`Ticker`**`(`*callback*, *options*`)`

Arguments

- **`paint`** (*`function`*) – Gets called at every game tick.
- **`options`** (*`object`*) – The possible options:

  `Scene.Ticker.options.`**`tickDuration`**
  Duration in milliseconds of each game tick.

  `Scene.Ticker.options.`**`useAnimationFrame`**
  If true the ticker will use a requestAnimationFrame callback instead of a standard setTimeout.

`Ticker.`**`run`**`()`
Start the ticker.

`Ticker.`**`pause`**`()`
Pause the ticker.

`Ticker.`**`resume`**`()`
Resume after a pause.

`Ticker.`**`lastTicksElapsed`**
lastTicksElapsed is the number of ticks elapsed during two runs of the paint function. If performances are good the value should be 1. If the number is higher than 1, it means that there have been more game ticks than calls to the paint function since the last time paint was called. In essence, there were dropped frames. The game loop can use the tick count to make sure that it's physics end up in the right state, regardless of what has been rendered.

`Ticker.`**`currentTick`**
The number of elapsed ticks that have been occurred since the creation the the ticker.

3.4.5 Cycle Object

A Cycle object manages sprite animations by moving the offsets within the viewport of the sprites.

This is an example cycle with 3 different offset, each lasting 5 game ticks:

```
var cycle = scene.Cycle([[0, 2, 5],
                         [30, 2, 5],
                         [60, 2, 5]]);
var sprite = scene.Sprite("walk.png");
cycle.addSprite(sprite);
cycle.update();

cycle.next(5).update();
```

Cycle complete reference:

class `sjs.`**`Cycle`**`(`*triplets*`)`

Arguments

- **`triplets`** (*`Array`*) – An array of triplets (xoffset, yoffset, ticks duration).

`Cycle.`**`addSprite`**`(`*sprite*`)`

Arguments

- **`sprite`** (*`Sprite`*) – Add a sprite to the cycle.

`Cycle.`**`addSprites`**`(`*sprites*`)`

Arguments

- **`sprites`** (*`Array`*) – Add an array of sprites to the cycle.

`Cycle.`**`removeSprite`**`(`*sprite*`)`

Arguments

- **`sprite`** (*`Sprite`*) – Remove a sprite from the cycle.

`Cycle.`**`next`**`([`*ticks*, *update*`])`

Arguments

- **ticks** (*number*) – The number of ticks you want to go forward. The default value is 1.
- **update** (*boolean*) – If true, the sprite's offsets will be automaticaly updated.

Calling the next method doesn't necessarily involve an offset change. It does only when all the ticks on current triplet have been consumed.

`Cycle.`**`go`**`(`*tick*`)`

Arguments

- **tick** (*number*) – Go to the passed tick in the triplets and apply the offsets.

`Cycle.`**`reset`**`(`*update*`)`

Arguments

- **update** (*boolean*) – If true, the sprite's offsets will be automaticaly updated.

Resets the cycle offsets to the original position.

`Cycle.`**`update`**`()`
Calls the update method on all the sprites.

`Cycle.`**`done`**
This attribute can be checked to determine if the cycle has completed. The value stays false if cycle is repeating.

`Cycle.`**`repeat`**
If set to false, the cycle will stop automaticaly after one run. The default value is true.

3.4.6 Input Object

The input object remembers user inputs within each game tick:

```
var input  = scene.Input();
if(input.keyboard.right) {
    sprite.move(5, 0);
}
if(input.keyboard.z)
    console.log("Key z is down")
```

`class` `scene.`**`Input`**`()`
Creates an Input object for the scene.

`Input.`**`keyboard`**
Input.keyboard is a record of which key has been pressed or released. In addition to the normal keyboard keys, the keyboard object also keep track of these special keyboard states:

```
keyboard.up
keyboard.right
keyboard.up
keyboard.down
keyboard.enter
keyboard.space
keyboard.ctrl
keyboard.esc
```

`Input.`**`keydown`**
True if any key is down.

`Input.`**`mousedown`**
True if any mouse button is down.

If you need to know which key has been pressed or released during the last game tick, use these methods:

`Input.`**`keyPressed`**`(`*code*`)`

Arguments

- **code** (*string*) – The type of key you want to test. eg: "up", "left"

`Input.`**`keyReleased`**`(`*code*`)`

Arguments

- **code** (*string*) – The type of key you want to test. eg: "up", "left"

`Input.`**`mouse`**

The mouse object contains the position of the mouse and if the mouse is clicked.

```
if(mouse.position.x < scene.w / 2)
    player.move(-2, 0);

if(mouse.click)
    console.log(mouse.click.x, mouse.click.y);
```

Touch Events

A small swipe updates the keyboard in the wanted direction and a tap will act as the spacebar being pressed. The mouse position and clicks are also updated by the touch events.

3.4.7 Layer object

If you need to separate your sprites into logical layers, you can crate a Layer:

```
var background = scene.Layer('background', {
    useCanvas:true,
    autoClear:false
});
var sprite = background.Sprite('background.png');
```

`class` `Scene.`**`Layer`**`(`*name*[, *options*]`)`

Arguments

- **name** (*string*) – The name of the layer
- **options** (*object*) – an option object

Create the Layer object.

`Layer.options.`**`useCanvas`**

If true this layer will use the canvas element to draw the sprites. This enables you to mix HTML and canvas.

`Layer.options.`**`autoClear`**

If false this disables the automatic clearing of the canvas before every paint call.

`Layer.options.`**`parent`**

Sets a different DOM parent instead of the scene.

3.4.8 Dealing With Events

Sprite.js provides the Input helper object for managing keyboard input. If you need more complex events handling the recommanded way is to use event delegation on the Scene object or a specific Layer object:

```
var scene = sjs.Scene({w:640, h:480});
var frontLayer = scene.Layer("front");

frontLayer.dom.onclick = function(e) {
    var target = e.target || e.srcElement;
    target.className = 'selected';
}

scene.dom.onclick = function(e) {
    var target = e.target || e.srcElement;
    target.className = 'selected';
}
```

If you need to use events on a Sprite level you can do it if you use the HTML engine:

```
sprite.dom.addEventListener('click', function(e) {
    sprite.dom.className = 'selected';
}, true);
```

3.4.9 Extra Features

To use some of these feature, you must include an extra javascript files in your web page.

ScrollingSurface object

Scro This object is not included in sprite.js core and needs to be loaded independantly:

```
<script src="lib/scrolling.js"></script>
```

This object provide a simple and efficent way to display a moving background within a scene. The object buffers parts that have already been drawn and only redraw the necessary parts instead of the whole scene at every frame.

```
var surface = sjs.ScrollingSurface(scene, w, h, redrawCallback);

function redrawCallback(layer, x, y) {
    // draw the necessary sprites on the layer
    sprite.canvasUpdate(layer);
}

surface.move(5, 0);
surface.update();
```

The redrawCallback is called whenever a part of the surface needs to be updated. The absolute position on the surface is provided for you as an argument to redrawCallback so you may determine what to draw on this layer. The layer object has a width and height (layer.w, layer.h).

class `sjs.`**`ScrollingSurface`**(*scene*, *w*, *h*, *redrawCallback*)

Arguments

- **scene** (`Scene`) – The scene that will hold the surface.

- **w** (*number*) – The width of the surface.
- **h** (*number*) – The height of the surface.
- **redrawCallback** (*function*) – A function the surface will call when a piece of surface needs to be painted.

redrawCallback (*layer*, *x*, *y*)

Arguments

- **layer** (*Layer*) – A layer where you need to draw your sprites. The layer object has a width and height (layer.w, layer.h) that is smaller than the surface size.
- **x** (*number*) – The x position of the layer within the scrolling surface.
- **y** (*number*) – The y position of the layer within the scrolling surface.

ScrollingSurface.**move** (*x*, *y*)
Moves the surface offset in direction (x, y).

ScrollingSurface.**position** (*x*, *y*)
Sets the surface offset position to (x, y)

ScrollingSurface.**update** ()
Updates the latest changes to the surface and call the redrawCallback if necessary.

Math

Sprite.js comes packaged with a few basic math functions:

sjs.math.**hypo** (*x*, *y*)
Hypotenuse

sjs.math.**mod** (*n*, *base*)
A modulo function that return strictly positive result.

sjs.**normalVector** (*vx*, *vy*[, *intensity*])
Return a normal vector {x, y}. If you define the intensity, the vector will be multiplied by it.

Path Finding

This object is not included in sprite.js core and needs to be loaded independently:

```
<script src="lib/path.js"></script>
```

Sprite.js has a flexible path finding function:

sjs.path.**find** (*startNode*, *endNode*[, *maxVisit=1000*])
The algorithm return undefined if no path has been found and the startNode if a path is found. You can then follow the path using this code:

```
var node = sjs.path.find(startNode, endNode);
while(node) {
    console.log(node);
    node = node.parent;
}
```

A node object should implement those 4 methods:

class Node(...)
Define your own Node object.

Node.**neighbors**()
Must return a list of Nodes that are the neighbors of the current one.

Node.**distance**(*node*)
Returns the distance from this node to another one. It's mainly used as a hint for the algorithm to find a quicker way to the end. You may return 0 if you don't want to implement this method.

Node.**equals**(*node*)
Returns true if two nodes are identical, eg:

```
return this.x == node.x && this.y == node.y;
```

Node.**disabled**()
Returns true if the current node cannot be used to find the path.

The Entity/Component Model

If you wish to use a entity component model with Sprite.js I would recommend the use of this external library https://github.com/batiste/component-entity

3.5 Changelog

This file describe new features and incompatibilites for sprite.js

3.5.1 Release 1.2.1

- New custom events.
- Tweak the touchmove sensibility.
- Add support for smaller bounding box collisions.

CHAPTER 4

Projects that Use Sprite.js

- Steam is a platform game involving a physics puzzle.
- Webattle.js is a multiplayer HTML5 game using node.js.
- “The invasion of the evil lords”. is a demo RPG with different creatures and a boss.

Index

C

Cycle.addSprite() (*Cycle method*), 15
Cycle.addSprites() (*Cycle method*), 15
Cycle.done (*Cycle attribute*), 16
Cycle.go() (*Cycle method*), 16
Cycle.next() (*Cycle method*), 15
Cycle.removeSprite() (*Cycle method*), 15
Cycle.repeat (*Cycle attribute*), 16
Cycle.reset() (*Cycle method*), 16
Cycle.update() (*Cycle method*), 16

I

Input.keyboard (*Input attribute*), 16
Input.keydown (*Input attribute*), 16
Input.keyPressed() (*Input method*), 17
Input.keyReleased() (*Input method*), 17
Input.mouse (*Input attribute*), 17
Input.mousedown (*Input attribute*), 16

L

Layer.options.autoClear (*Layer.options attribute*), 17
Layer.options.parent (*Layer.options attribute*), 17
Layer.options.useCanvas (*Layer.options attribute*), 17
List.add() (*List method*), 13
List.iterate() (*List method*), 14
List.list (*List attribute*), 14
List.list.length (*List.list attribute*), 14
List.remove() (*List method*), 14

N

Node() (*class*), 19
Node.disabled() (*Node method*), 20
Node.distance() (*Node method*), 20
Node.equals() (*Node method*), 20
Node.neighbors() (*Node method*), 20

R

redrawCallback() (*built-in function*), 19

S

Scene.Cycle() (*class*), 9
Scene.Input() (*class*), 9
scene.Input() (*class*), 16
Scene.Layer() (*class*), 9, 17
Scene.loadImages() (*Scene method*), 9
Scene.options.autoPause (*Scene.options attribute*), 9
Scene.options.h (*Scene.options attribute*), 9
Scene.options.parent (*Scene.options attribute*), 9
Scene.options.w (*Scene.options attribute*), 9
Scene.reset() (*Scene method*), 9
Scene.Sprite() (*class*), 9, 10
Scene.Ticker() (*class*), 9, 14
Scene.Ticker.options.tickDuration (*Scene.Ticker.options attribute*), 14
Scene.Ticker.options.useAnimationFrame (*Scene.Ticker.options attribute*), 14
ScrollingSurface.move() (*ScrollingSurface method*), 19
ScrollingSurface.position() (*ScrollingSurface method*), 19
ScrollingSurface.update() (*ScrollingSurface method*), 19
sjs.Cycle() (*class*), 15
sjs.List() (*class*), 13
sjs.math.hypo() (*sjs.math method*), 19
sjs.math.mod() (*sjs.math method*), 19
sjs.normalVector() (*sjs method*), 19
sjs.path.find() (*sjs.path method*), 19
sjs.Scene() (*class*), 8
sjs.ScrollingSurface() (*class*), 18
Sprite.angle (*Sprite attribute*), 11
Sprite.applyVelocity() (*Sprite method*), 12
Sprite.applyXVelocity() (*Sprite method*), 12
Sprite.applyYVelocity() (*Sprite method*), 12

Sprite.canvasUpdate() (*Sprite method*), 10
Sprite.collidesWith() (*Sprite method*), 13
Sprite.collidesWithArray() (*Sprite method*), 13
Sprite.color (*Sprite attribute*), 11
Sprite.distance() (*Sprite method*), 12
Sprite.explode2() (*Sprite method*), 13
Sprite.explode4() (*Sprite method*), 13
Sprite.h (*Sprite attribute*), 11
Sprite.isPointIn() (*Sprite method*), 13
Sprite.loadImg() (*Sprite method*), 10
Sprite.move() (*Sprite method*), 12
Sprite.offset() (*Sprite method*), 12
Sprite.orientVelocity() (*Sprite method*), 12
Sprite.position() (*Sprite method*), 12
Sprite.remove() (*Sprite method*), 10
Sprite.reverseVelocity() (*Sprite method*), 12
Sprite.reverseXVelocity() (*Sprite method*), 12
Sprite.reverseYVelocity() (*Sprite method*), 12
Sprite.rotate() (*Sprite method*), 11
Sprite.rotateVelocity() (*Sprite method*), 12
Sprite.rv (*Sprite attribute*), 12
Sprite.scale() (*Sprite method*), 12
Sprite.setAngle() (*Sprite method*), 11
Sprite.setBackgroundRepeat() (*Sprite method*), 11
Sprite.setColor() (*Sprite method*), 11
Sprite.setH() (*Sprite method*), 11
Sprite.setOpacity() (*Sprite method*), 11
Sprite.setW() (*Sprite method*), 11
Sprite.setX() (*Sprite method*), 11
Sprite.setXOffset() (*Sprite method*), 11
Sprite.setXScale() (*Sprite method*), 11
Sprite.setY() (*Sprite method*), 11
Sprite.setYOffset() (*Sprite method*), 11
Sprite.setYScale() (*Sprite method*), 11
Sprite.size() (*Sprite method*), 12
Sprite.update() (*Sprite method*), 10
Sprite.w (*Sprite attribute*), 11
Sprite.x (*Sprite attribute*), 11
Sprite.xoffset (*Sprite attribute*), 11
Sprite.xscale (*Sprite attribute*), 11
Sprite.xv (*Sprite attribute*), 12
Sprite.y (*Sprite attribute*), 11
Sprite.yoffset (*Sprite attribute*), 11
Sprite.yscale (*Sprite attribute*), 11
Sprite.yv (*Sprite attribute*), 12

T

Ticker.currentTick (*Ticker attribute*), 15
Ticker.lastTicksElapsed (*Ticker attribute*), 15
Ticker.pause() (*Ticker method*), 15
Ticker.resume() (*Ticker method*), 15
Ticker.run() (*Ticker method*), 14