

StrikeDisplay 1.0r1

Hi.

I'm Josh, and I'm an AS3 coder. Back when I was a kid it was "AS2" -- practically Javascript! It just seemed like innocent fun. Now I find myself embarrassing friends in social situations, ranting about Flex until everyone around me is in a coma. I've tried to get on the JS wagon, but I keep falling off.

What I need – to stretch a bad analogy – is a sponsor. If you like StrikeDisplay, send me an email. **If you use part of it on a site, or for an app, support my work and send Bitcoins to:**

1ANkrqM7CfGrap7fyini3rBRjFfPerQYD1

So now that we've been introduced, let's get down to the good stuff.

What is this?

StrikeDisplay is a pseudo-AS3 drawing and animation kit for Javascript and the HTML5 canvas. It overlays a simple display chain and event structure on the canvas, letting you create "Sprites" (in the AS3 sense), place them on a stage, add children to them, move them around, and have them react individually or in tandem with mouse input, without having to deal with what's going on under the surface. It redraws the necessary portions of the canvas automatically, figures out what's under the mouse and dispatches some ready-scoped events for you in a fast, sane, efficient manner. The point is to keep it extremely lightweight and very fast. It doesn't rely on any other frameworks. I personally hate JS and I really hate JQuery and all the other stuff that's supposed to make things easy, because they require no imagination and sites built with them all look the same. Meanwhile, as much as I've loved Proce55ing all these years, it's not really a good substitute for a lot of things that were easy to do back in Flash 5. I adore AS3 and I hope it never dies. I think Flash is beautiful. If you don't think so, close this document now and scram. You heard me. Vamoose.

Still here?

StrikeDisplay is more a proof of concept than anything else, but it's nice to have in your toolkit as an alternative in the future. The canvas is a lot slower than the Flash AVM, and there are a lot of things Flash does that would just be way too much overhead for Javascript to emulate. Still, anyone who writes AS3 will find

this kit very easy to work with. Unlike Gordon, which is a very cool project, this library doesn't try to actually convert SWF's into SVG or run any real bytecode. It's not an attempt to compile AS3 to Javascript the way Jangaroo is, although I think it could readily be used with Jangaroo as a "light" alternative to the AS3 libraries, and it'd probably feel even more like y'know, real object oriented code that way. But even in pure Javascript with StrikeDisplay, you can get away with a ton of stuff, without relearning anything from AS3. It just lets smart Flex/Flash coders like you and I do some of what we do in a canvas, without having to monkey with all of the lower-level junk that we're used to taking for granted.

You have to know some Javascript and be very comfortable with AS3 for this to make any sense, so if you're not, brush up. This is *unsupported software*. There's not exactly a phone bank over here in my garage... so please... don't ask me how to use it. If I can hack it together, you can hack it apart. I'm doing the best I can with the docs, but I work 12 hours a day as it is...

How it Works & Extending the API.

It's pretty trivial to add new graphics methods. Some of them I'm going to get around to, some of them I won't.

Let's look at how StrikeDisplay works. The goal here is not pixel manipulation; it's to get away from pixel manipulation. Every sprite has a graphics property. When a user invokes movement on a sprite or draws anything in its graphics, the sprite is invalidated. That means that on the next render loop, the bounding box where it was will be filled with the background color before anything else is drawn, and it will be rendered again (along with its children). In fact, at the moment, all objects on screen are re-rendered with every loop. I built a traverser that does a half-assed job of figuring out which objects not to re-render, but even in the cases where it works (and there are a lot of cases where it doesn't work) the collision calculation is just a lot slower than re-rendering these things to the canvas. So that's another to-do, maybe.

All the Sprite's position data is held in a 3x3 matrix (the `Sprite.transform` property). To draw a Sprite, the renderer just concatenates its matrix with each of its parents' matrices up to the stage (which is also a Sprite), then transforms the canvas to that, draws the thing, and restores the former canvas setting. When users write something like `mySprite.graphics.drawRect(0,0,20,20);` what they're doing is adding an object (really, a bunch of strings) onto the Graphics object's list of things to draw. On each internal loop, the renderer goes through that list, parses the

object to generate an appropriate translation, then runs an eval statement on the result to produce a drawing of the object. That's why for most graphics operations like `drawRect()` there is another function like `_drawRect()` which is called by the renderer. The internal `_function` also takes one extra argument, which is the context we're talking about. Each `StrikeDisplay` works on one and only one visible canvas; the reason for the context argument will be discussed below.

Sprites can be cached as bitmaps, which essentially clears out their graphics drawing list and replaces it with a single stored image of whatever the sprite looked like on its original draw list. The original list is saved so you can uncache it later or modify it if you want to. Bitmap caching is faster in a lot of cases, but as of this release, the only things that cache automatically are `TextFields`. See the optimization notes at the end of this document for more info about caching.

Each `Sprite` has a `BoundingBox` object which is a rectangle in global space that surrounds it. `deriveBounds(s:Sprite);` returns the smallest known box around any given `Sprite` and *all of its children's bounding boxes* in global space. This is used to speed redraws. Only areas within bounding boxes are drawn on each render pass.

Now here's the kicker. There's an internal event system here that looks a lot like the AS3 event system. Included in this are `mouseover`, `mouseout`, `mousedown`, `mouseup`, `mousemove`, `click` and `loaded`. The render loop checks the mouse on each frame and narrows down which `Sprites` have bounding boxes under the mouse. That gets us halfway to knowing whether the `Sprite` is *actually* under the mouse; but `Sprites` have different shapes and there's only one way to figure out if they touch the pointer or not. You guessed it; this is what that context arguments in the `_internal_graphics_` functions are for. Every `StrikeDisplay` creates its own hidden canvas. When the render loop finds something in the display chain with a bounding box under the mouse, it clears that hidden canvas and calls the `_internal_graphics` routines on that `Sprite` *with the hidden canvas' context as an argument*. The `_internal` functions know to draw only in black on that canvas and ignore alpha settings, etc. Then the render loop takes the pixel from the hidden canvas that's under the mouse and checks if it's not white. If it is white, it goes to the next `Sprite` down under the mouse (for this purpose they're put in a list sorted by z-order beforehand).

Alright, so this all works the way you'd expect it to, and it's probably more than you need to know, but if you're going to add new graphics methods, hopefully

this will help you out. Beyond that, enjoy it and make some good games and send me a postcard with a buck or two in my email, you'll make my day.

Josh Strike

josh@joshstrike.com

==-----==

Documentation.

StrikeDisplay is invoked with one line:

```
var root = new StrikeDisplay("main", "#000000", 30, false);
```

Where "main" is the id of a canvas on your page. You can create multiple versions on separate canvases. You don't have to call your root object "root", but it's kinda fun, right?

You must nest a StrikeDisplay canvas inside a div that specifies position:relative or position:absolute in its style. Placing it in an unstyled <div align="center"> will break mouse functionality. If you want to center it, use

```
<div style="width:100px;margin:auto;">
```

And replace 100px with the width of your canvas. It is possible to place it directly in the body, but make sure your body tag has left and top margins set to zero.

```
Strike.StrikeDisplay(canvasID:String,  
backgroundColor:'#000000', framerate:Number,  
debug:Boolean)
```

StrikeDisplay takes the id of a canvas already loaded in the HTML. It creates an internal render loop. It generates a stage which is the base of the display chain. I find that a comfy way to name a StrikeDisplay in Javascript is 'root'.

ALL colors in the whole StrikeDisplay framework MUST be passed in '#000000' string format, with SIX HEX DIGITS. This is how they are parsed.

Properties:

canvas - The DOM canvas object corresponding to the canvasID passed in.

context - The canvas' 2d context.

stage - The base sprite to which you add children.

debug - Show or don't show bounding boxes on redraw.

Methods:

`zCompare(a:Sprite,b:Sprite):Number` - Returns 1 if `a` has a higher *global* z-position, otherwise returns -1.

All other methods are internal.

Strike.Mouse(display:StrikeDisplay)

Generated once for each StrikeDisplay. Takes the display as its only argument. Local mouse coordinates can also be obtained from the `mouseX` and `mouseY` properties of Sprites (including the stage, whose coordinates are global).

Properties:

`x`, `y` - global mouse coordinates.

All other properties are internal. Don't rely on them for external work.

Strike.Event(type:String [, currentTarget:Object, target:Object])

Watch out for the scope returned by the event. If you're using 'this' in the function called by the event, you need to bind the scope to the function you pass into the listener. See the examples below.

Properties:

`type` - String type.

`currentTarget` - The object to which the listener was attached.

`target` - In the case of automatically dispatched mouse events, this is the sprite which received the original event. If a sprite has a click listener on it, and one of its children gets clicked, it will dispatch an event in which it is the target and the child clicked is the `currentTarget`.

Example 1:

```
function Holder(){
    var box = new Sprite();
    box.graphics.beginFill("#000000");
    box.graphics.drawRect(0,0,20,20);
    box.addEventListener("click",gotClick);
}
```

```
Holder.prototype.handleClick = function(evt) {
    //this is scoped to the event object. this!=holder.
}
```

Example 2:

```
function Holder(){
    var box = new Sprite();
    box.graphics.beginFill("#000000");
    box.graphics.drawRect(0,0,20,20);
    box.addEventListener("click",handleClick.bind(this));
}
Holder.prototype.handleClick = function(evt) {
    //this will be scoped to the Holder object because it's
been bound.
}
```

Strike.Point(x:Number, y:Number)

Properties:

x, y - Just about what you'd expect.

Strike.Matrix(a,b,c,d,tx,ty,u,v,w) -- All numbers only.
All optional. Defaults to an identity matrix.

NOTICE: THIS MATRIX IS CORRECT -- UNLIKE AS3's.
IDENTITY IS 1,0,0,0,1,0,0,0,1.

Properties:

All of the arguments are accessible.

Methods:

translate(x:Number,y:Number):void - Changes tx and ty. For Sprites, tx and ty are used to hold their position in relation to their parent.

scale(sx:Number,sy:Number):void - Sets the scale *including position*.

innerScale(x:Number,y:Number):void - Basically a deltaScale. Takes a new identity matrix and multiplies it by this.

rotate(deg:Number):void - Additively rotate the matrix by a number

of *degrees*.

`transformPoint(point:Point):Point` - Transform a point into the matrix's space.

`deltaTransformPoint(point:Point):Point` - Transform a point into the matrix's space *disregarding tx and ty*.

`multiply(m:Matrix):Matrix` - Returns a new matrix, this multiplied by the argument.

`inverse():Matrix` - Returns a new matrix which is the inverse of this one (i.e., this times that equals an identity matrix).

`getScaleX()` - Returns the current scaleX for the matrix.

`getScaleY()` - Yup.

`getRotation()` - Ditto.

Strike.Graphics(s:Sprite)

Properties:

All for internal use.

Methods:

`clear():void` - Clears the display list. Effectively empties out the graphics.

`beginFill(color:'#FFFFFF',[alpha:Number]):void` - Starts a fill before you draw a rect, circle or path.

`beginSimpleGradient(colorA:'#FFFFFF',alphaA:Number,colorB,alphaB,xa,xy,xb,yb):void` - Creates a linear gradient on all further draws until you use `endFill()`. xa, xb, ya and yb are the from and to points for the gradient.

`endFill():void` - Stops all further draws from filling.

`lineStyle(width:Number,color:'#000000',alpha:Number):v`

`oid` - Creates a stroke on all further draws until you set the `lineStyle` to a width of zero.

`moveTo(x,y):void` - Positions the pen to draw.

`lineTo(x,y):void` - Draws a path from the current pen position to `x` and `y`.

`curveTo(xa,ya,xb,yb):void` - Does a quadratic curve.

`endPath():void` - **Critical**. This must be called after each path in order to close it, fill it and stroke it, because of the way the canvas works.

`drawRect(x,y,w,h):void` - Same as in AS3.

`drawRoundRect(x,y,w,h,c):void` - Same as in AS3.

`drawCircle(x,y,radius):void` - Same as in AS3.

`drawTextLine(x,y,text,font,size,color[,checkOnly:Boolean]):Number` - Draws a line of text at the specified coordinates. Font must be a string with the font name only; this is reassembled with the size provided as a Canvas font descriptor. If `checkOnly` is set true, this method does not draw, but returns the number of characters that won't fit into the Sprite's current width (if that width is greater than zero). If `checkOnly` is false, it draws text up to the width of the Sprite (if specified), and returns the final pixel width of the line it drew. Used internally by TextField to mimic AS3 behavior. The reason for returning the character counts before drawing is to let TextField justify the line. The reason for returning width values after drawing is to let TextField center or right-align it. Mainly for internal use, but works as advertised; proceed with caution.

`drawImage(name:String, src:'images/file.jpg' [, x:Number, y:Number, dw:Number, dh:Number, sx:Number, sy:Number, sw:Number, sh:Number]):void` - Draws an image into the graphics of the Sprite. If the image has already been loaded, attempts to use the loaded version. To-do: An unload feature. The image draws into the sprite's graphics at optional the `x` and `y` positions provided, to a width of `dw` by `dh` (destination width, destination height). If you do not provide `dw` and `dh` parameters, the image will be drawn at its original size, *but the Sprite will not take on the width and height of the image until the*

image has fully loaded. The optional sx, sy, sw and sh parameters control what part of the source image you want to copy from. If you don't use them, the whole source image will be copied into the graphics. **The graphics' sprite will dispatch a "loaded" event whenever this load completes.**

`attachDiv(id:String, x:Number, y:Number):void` - Right now, StrikeDisplay doesn't support text within the canvas. It will probably be the next thing I add. However, this lets you take any div and attach it to a Sprite's graphics, set its position in relation to the Sprite it's inside of, and have it move around with the Sprite as you please. Of course, other divs are always going to be positioned on top of the canvas. For now, this is the text solution for the framework.

`removeDiv(id:String):Boolean` - The div will no longer be hooked up to the Graphic, but that basically means it's going to just sit where it was left. Use this in conjunction with javascript to set the style.top and style.left of your div to make it disappear.

`setDivProperties(id:String, props:Object):Boolean` - Properties used are {x:Number, y:Number}. More to be added.

`deriveAlpha():void` - Returns the global cumulative alpha of the holding Sprite and all its parents.

Strike.Sprite([display...do not use this flag, it creates the stage])

The Sprite is the basic building block of StrikeDisplay.

Properties:

`name:String` - Arbitrary, not necessary.

`parent:Sprite` - The parent sprite in the display chain (or null for the stage).

`invalid:Boolean` - Whether the sprite will be redrawn on the next render loop.

`x, y, scaleX, scaleY, width, height, rotation, alpha, mouseX, mouseY` - The position of the sprite in relationship to its

parents. They work like you'd expect them to.

`shadow:DropShadow` - A drop-shadow to apply to the Sprite.

`blockRedraw:Boolean` - Prevents the Sprite *and all its children* from being redrawn. Powerful. Use with caution.

`stage:Sprite` - Reference to the stage, if this sprite is on the display chain; otherwise null.

`cacheAsBitmap:Boolean` - Draw the Sprite's *graphics contents* as a single bitmap on all future redraws. Much faster in some cases. *Does not cache children*. A few notes are in order...

When a sprite is cached, its draw list at `sprite.graphics.list` is copied for reversion in case it's uncached. The saved copy is at `sprite.graphics._cacheList`. The saved ingredients of the original bounding box are at `sprite.graphics._cacheCorners`. Drawing to a cached sprite's graphics causes it to be uncached, redrawn, and cached again automatically on the next frame.

`clippingSprite:Sprite` - Roughly equivalent to the mask feature in AS3, with a few caveats. Any sprite can be used as a clipping mask on another sprite or TextField. When sprite "A" is set as the clippingSprite of "B", sprite "A" becomes a child of "B" and its position is translated, regardless of its position in the display chain prior. Opacity levels are not supported. As of version 1.0, children are now clipped along with their parent. However, unpredictable behavior may occur when attempting to clip on multiple levels down the display chain. To wit, this may not work at all and is completely untested. Additionally, be aware that the mask executes *after* any filters on the sprite, so a blurred sprite will have a hard edge at the edge of its mask.

`filters:Array` - Filters to place on a Sprite. Once set, you must redefine this property to make changes to a filter. Filters are executed in the order placed, with the exception of `DropShadowFilter`, which is run during the draw using the native Canvas API. All other filters cause sprites to be cached as bitmaps as long as they're applied. Set this to a new Array, not null, to clear your filters. Filters only apply to individual sprites, they don't cascade to children.

`mouseEnabled:Boolean` - Defaults to true. If set false, overrides all mouse events.

`_bb:BoundingBox` - The bounding box for the Sprite. Sometimes useful to know. A quick note on `BoundingBox()`, because it's not covered really in this documentation. All graphics methods add to a list of corners for the `Sprite.graphics` object. These are what's used to figure out the size of the bounding box for the Sprite and, subsequently, for its parents. This isn't a perfect system. Check out what happens to a bounding box when you rotate a circle, for example. But it's fast. Fast enough to pass for Flash in some circumstances. Bounding boxes exist for redraw speed, and are not used to check mouse events. They shouldn't be relied on for collisions testing, for example.

Methods:

`addEventListener(type:String, fun:Function):void` - Adds a `Sprite.Event` listener. Built-in types dispatched automatically for Sprites are: "mouseover", "mouseout", "mousedown", "mouseup", "mousemove", "click" and "loaded" (any time an image is loaded into a Sprite).

`removeEventListener(type:String, fun:Function):void` - Removes a listener, if there is one. If the target function was bound to something else, just specify the target without binding. For example, if you added a listener like this so you could call a prototype function:

```
spriteChild.addEventListener("click",this.foo.bind(this));
spriteChild.prototype.foo = function(evt) {}
```

you should remove the listener like this, without the binding:

```
spriteChild.removeEventListener("click",this.foo);
```

Note that `bind()` stores a copy of its target method as the `_method` parameter, which is used automatically if `removeEventListener` finds a bound function.

`hasEventListener(type:String, fun:Function):Boolean`
Similar to `removeEventListener`, this also checks for the original function if it was bound to a different scope.

`destroyEventListeners():void` - Does a little clean-up on the main event list in the hopes Javascript might garbage collect some of these functions. For safety, call this after you remove all event listeners from a Sprite, before you remove it from the stage. For good measure, use

delete(sprite) on it if you're not going to use it anymore. Clean your references! Javascript GC implementations are terrible in most browsers.

`addChild(sprite:Sprite):void`

`addChildAt(sprite:Sprite, index:Number):void` - more forgiving than AS3 ever was.

`getChildAt(index:Number):void`

`removeChild(sprite:Sprite):void` - this is more forgiving than AS3, too.

`removeChildAt(sprite:Sprite):void`

`setChildIndex(sprite:Sprite, index:Number):void`

`localToGlobal(point:Point):Point` - Takes a point in the sprite's space and returns one in the global canvas space.

`globalToLocal(point:Point):Point` - The opposite of the above.

`concatMatrix():Matrix` - Returns a concatenated transform matrix for this Sprite and its parents up to the stage.

`dispatchEvent(event:StrikeEvent):void` - Dispatches an event with this Sprite as the target. The sprite dispatching the event always overrides any prior target set on the event. This is important to note if you're going to have sprites re-dispatching events from other sprites.

Strike.TextField([display...do not use this flag])

Extends Sprite and contains all Sprite functionality, but does not support graphics being drawn into it. Can have children and clippingSprite masks. Automatically cached as a bitmap unless otherwise specified.

Properties:

(ALL SPRITE PROPERTIES)

`text:String` - Text to display. Use \n for line breaks.

`multiline:Boolean` - Allow both multiline text with breaks and automatic word wrapping. There is no separate `wordwrap` property.

`width, height:Number` - Width and height of the field in pixels. Overrides normal sprite scaling behavior when specified. Use `scaleX/scaleY` to scale textfields.

`textWidth (read-only):Number` - The width of the field's bounding box.

`textHeight (read-only):Number` - The height of the field's bounding box.

Methods:

(ALL SPRITE METHODS)

`setTextFormat(format:TextFormat):void` - Sets formatting on the text for the field. Currently, each field can only have one `TextFormat`.

`Strike.TextFormat`(`font:String`,`size:Number`
[,`color:String`="#000000",`align:String`="left"])

TextFormat objects are applied to TextFields, and once set, also serve as the default TextFormat when future changes are made to the text.

Properties:

`font:String` - CSS-recognizable name of a font already loaded. For example, "Arial".

`size:Number` - Point size of the font.

`color:String` - A "#000000" formatted string.

`align:String` - Can be "left", "center", "right" or "justify". Note that justification randomly distributes spaces to fill out lines. Operations that force a textfield to redraw, including uncaching it, will cause rivers in the text to shift randomly. To avoid this, don't show justified fields until they're final and (auto)-cached.

Strike.Tween(sprite:Sprite, property:String, easing:Function, from:Number, to:Number, seconds:Number [, display:StrikeDisplay])

Simple tween class. Basically works similarly to AS3 fl.transitions.Tween. Notice that seconds is the number of seconds for the tween, you don't set ticks with a seconds flag, you just put in the number of seconds. *You do not have to pass a sprite in, you can pass any object with any property you want to tween...* but because of the nature of the StrikeDisplay event hierarchy, if you pass something other than a Sprite already attached to the display chain, you **MUST** pass a reference to the StrikeDisplay. Otherwise nothing will happen.

Supported tweening functions:

EaseNone, EaseOut, EaseInOut.

Methods:

stop():void - Jane, get me off this crazy Tween...

addEventListener(type:String, fun:Function):void - Adds a StrikeEvent listener. The only built-in type dispatched by Tween is "motionFinished".

removeEventListener(type:String, fun:Function):void - Removes a listener. Negotiable.

hasEventListener(type:String, fun:Function):Boolean - Checks for a listener.

destroyEventListeners():void - See Sprite for more info.

Example:

```
var tween = new Tween(someSprite,"alpha",EaseOut,1,0,3);
```

Notice that EaseOut is not in quotes. That's because it's referring to an internal EaseOut function in StrikeDisplay. If your tween is not working, this might be why.

Strike.DropShadowFilter(x:Number, y:Number, blur:Number [,color:String="#000000", alpha:Number=1])
Applies a drop shadow during the draw phase of a sprite's graphics, using the native Canvas API.

Strike.BrightnessFilter(value:Number)
Adds or subtracts the given value to the red, green and blue channels of a sprite's graphics. The values themselves range between 0 and 256. Doesn't affect the alpha channel.

Strike.ColorFilter(r:Number, g:Number, b:Number, a:Number)
Adds or subtracts the given values to the red, green, blue and alpha channels of a sprite's graphics.

Strike.ConvolutionFilter(c:Array)
Takes a 3x3 convolution kernel (as a single array of 9 values) and applies it to the sprite's graphics.

Strike.BlurFilter(radius:int [,iterations=1])
Thanks to Mario Klingemann (<http://quasimondo.com>) for his super-fast open source StackBlur algorithm. This does what you expect it to do.

Strike.Function.bind(obj:Function)
Generic function method that applies a function on demand to another function. Used to fix the scope problems with events in javascript. Used internally. With StrikeDisplay events, the scope returned to the listening function is the listening function itself. You can tap into this if you need it. It's already included in most other frameworks.

Strike.Function.inherits(obj:Function)
StrikeDisplay implements a very basic inheritance system. You can make use

of this, for example to extend the Sprite class. Multiple inheritance is not supported, nor is passing arguments to the superclass constructor. You must include a call to `Function.supercall(instance)` in the first line of the subclass constructor, to bind it to the superclass. After you do this, you should not redefine the new function's prototype, although you can of course add to it. Here's an example of how to extend Sprite:

```
function Box(size) {  
    Box.supercall(this);  
    this.graphics.beginFill("#000000",1);  
    this.graphics.drawRect(0,0,size,size);  
    this.size = size;  
}  
Box.inherits(Sprite);
```

Notes on optimization.

Performance in StrikeDisplay can vary wildly, particularly in regards to caching and filters. While caching is a major boon for moving multiple bitmaps around, there are a few things you should know to make the most out of it.

- * **Caching is much better for relatively small vectors where the graphics are close to the sprite's origin.** Caching takes the entire bounded area of the sprite, stuffs it in a canvas, and blits that the next frame. So it always includes the sprite's origin. Meaning if you take a sprite and draw a tiny square (not a child, but a `graphics.drawRect`) at 1000px to the right of its origin, the cached image will be at least 1000px wide, and will include a lot of dead alpha space that has to be drawn. This can really slow down rendering.

- * **Cached bitmaps draw slower than simple shapes.** If you have less than half a dozen curves in a vector, it's probably faster not to cache that graphic (unless it has a gradient fill or something like that).

- * **If you must scale a cached sprite, cache first, scale later.** This one's really, really critical. Caching always takes an original drawing of the sprite, turns its graphics into a single image, and then puts it back on the display chain. So a cached sprite *always* scales up with the resolution it was originally drawn at, regardless of when you cache it. However, if you scale it *before* caching it, then the cache canvas won't be the same size as the image area it's being blitted

into, and the scaling will occur in the DOM layer instead of through StrikeDisplay's matrix. If that doesn't make sense, don't worry; just don't do it. The best thing you can do to scale **cached** sprites is add them as a child of another sprite, and scale the outer one. There is much less speed penalty for that. At the very least, *cache first, scale later*. Doing it the opposite way will be about 90% slower, until browsers start getting serious about optimizing their get/putImageData calls.

* **Filtering == Caching.** All filters in version 1.0 besides the DropShadowFilter cause the sprite to be cached. The implication is, if you have to filter and scale, filter first, scale later. Or filter it in a child and scale the parent. That's very fast. Both ways, if a sprite was not explicitly cached prior to having a filter put on it, it will be uncached and drawn as a vector when the filter is removed.

© 2010 - 2012 • Josh Strike • <http://joshstrike.com> • <http://thestrikeagency.com>
This software is provided as-is with no warranty express or implied. This software may be freely copied so long as this header is included. Copying of the software implies no transfer of ownership. This software may not be re-sold commercially, in whole or in part, without the express written permission of the author.