JavaScript and Objective-C Entanglement

BigCatOS.com
2025.04.01

Contents

The Background Palette

The effect appears in the opening scene of *How To Train Your Dragon*. The effect also appears during the climactic battle in *Avatar: The Way Of Water*. No doubt the effect appears in many other situations — the subtle, practically unnoticeable change in background lighting over time, whereby the viewer isn't even aware it's happening, until it has. And maybe not even then.

I wanted a similar effect — an imperceptible background color change over time — for a JavaScript App embedded in an iOS / macOS Objective-C WKWebView, a sim that updates its state every minute, with a period of 24 hours * 60 minutes / hour = 1440 minutes.

The idea is, given a palette of background colors, spread them over the cycle, with the understanding that colors must change smoothly and continuously, and the palette is aware of key times specified in JavaScript variables such as **`dawn`**, **`sunrise`**, **`sunset`** and **`dusk`** so that the palette is pliable and useable throughout the year. Parsing a background palette should generate a simple array of 1440 elements, indexed by minute 0 - 1439, each with a 9 character JavaScript RGBA hex color value. The App uses its simulated minute to index into the array and set the background:

```
    var minuteToBackgroundColor = [] // #00ff00ff
```

Programmatically, a number of virtual rectangular palette sections delineating a portion of the day would work, each specifying a starting minute and color (rectangle left edge) and an ending minute and color (rectangle right edge). The difference between the end minute and the start minute is the rectangle's ***minute distance***, which can vary depending on the time of year. These virtual rectangles do not require a height as only the left-to-right spacing of the edges along the horizontal time axis is important, but assume their virtual height is at least one pixel so they are visible, Figure 1.

For example, this array of 6 arrays of 4 scalars describes the palette sections in a hypothetical day:

```
    var backgroundPalette = [
        [ solarMidnight, "#000000ff", dawn           "#3F4174FF" ], // 1
        [ dawn,          "#3F4174FF", sunrise,       "#0475FFFF" ], // 2
        [ sunrise,       "#0475FFFF", solarNoon,     "#04BAFFFF" ], // 3
        [ solarNoon,     "#04BAFFFF", sunset,        "#0475FFFF" ], // 4
        [ sunset,        "#0475FFFF", dusk,          "#3F4174FF" ], // 5
        [ dusk,          "#3F4174FF", solarMidnight, "#000000ff" ], // 6
    ]
```

Palette sections are reminiscent of a simple 1-stop color gradient, but no gradients are involved at this time, only the minute distance between two colors is of interest, used to smoothly interpolate intermediate colors.

Each entry increases in time except the last, which goes back in time to the first entry thus completing the daily cycle (think the ends of a thin strip of paper wrapped into a cylinder without twisting and glued together). Also, the right edge of a palette section rectangle is dual both in time and color with the left edge of the following palette section. A time-color linked list, vaguely.

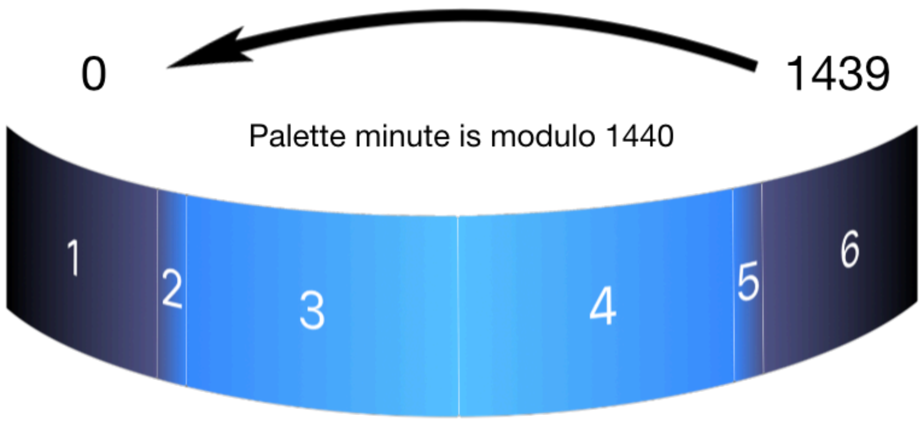


Figure 1
Virtual Palette Rectangles With Virtual Height > 0

This data structure is usable, it's simple and was the basis of an early version of dynamic background colors. It did evolve into a compactified structure that encodes the same information, while adding a new string keyword for use by the future Palette Editor:

```
var backgroundPalette = [
     [ "solarMidnight",  solarMidnight,   "#000000FF" ],
     [ "dawn",           dawn,            "#3F4174FF" ],
     [ "sunrise",        sunrise,         "#0475FFFF" ],
     [ "solarNoon",      solarNoon,       "#04BAFFFF" ],
     [ "sunset",         sunset,          "#0475FFFF" ],
     [ "dusk",           dusk,            "#3F4174FF" ],
]
```

Do not assume it represents a multi-stop color gradient (even though it can), the rectangle analogy continues to apply, with a common color between the left and right edges of two rectangles, and minute distances calculated using two adjacent palette sections.

Recursive Linear Color Interpolation

Color interpolation can be a complex subject depending on your needs, but this App's dynamic background colors feature is relatively minor, mostly in the surprise and delight category, so simple linear interpolation is sufficient. Turns out its first implementation was a surprise precisely because it was very undelightful.

Pseudo-code for linearly interpolating the color midway between two hex RGBA color values #R1G1B1A1 and #R2G2B2A2 is:

```
let midCol = '#' + (R1+R2)/2 + (G1+G2)/2 + (B1+B2)/2 + (A1+A2)/2
```

With appropriate rounding this always works, the new color is linearly in between the start and end colors, so there is a visually reasonable color path through the three colors.

Generalizing this for many intermediate colors means the common divisor of 2 must be replaced by the minute distance between the start and end colors, resulting in a color delta that's added repeatedly to the beginning color in order to reach the ending color.
But there's a problem — the generalization is unreliable, sometimes it works, other times we see banding. Banding produces inappropriate black or repeated colors, and occurs when the color delta becomes very small and rounds to zero. Also, duplicated colors must be intelligently distributed throughout the color path, not simply repeated at the end.

A scheme that handles all these problems reuses the technique of finding the midpoint color, which we know always works, then doing the same thing for the left and right halves of the interval, until the interval becomes vanishingly small. Although it might feel like taking a limit in Calculus class, we'll find all these intermediate colors between the left and right edges of a virtual palette rectangle using the **`backgroundPlatte[]`** array and the recursive function **`interpolateColors()`**.

Three program listings follow that demonstrate this color interpolation technique:

- Listing 1 gives preliminary definitions used in the remaining listings.
- Recursion is pretty opaque until you wrap your head around it and realize how amazingly awesome it is! A recursive function typically contains in its definition instructions that do stuff, then a call to itself to do similar stuff based upon what it just did. And in order to keep from crashing by exhausting the system call stack, a recursive function always needs a way out, a testable condition that can activate a clean exit from the function. Makes perfect sense, right? As an example, see Listing 2.
- Listing 3 illustrates how to construct virtual palette rectangles from adjacent background palette array entries and call **`interpolateColors()`** on those rectangles.

```
 1  const kLastMinute     = 24 * 60        // palette minute modulus
 2  const kHexLenRGB      = 6
 3  const kHexLenRGBA     = 8
 4  const kHexLen         = kHexLenRGBA    // count of hex digits in a color
 5  const kHexColorRegEx = /[a-f\d]{2}/ig // decompose hex color into components
 6  const kBGPKey         = 0
 7  const kBGPMinute      = 1
 8  const kBGPColor       = 2
 9
10  function byMinuteOfDay( a, b ) {
11
12      // a and b are backgroundPalette arrays of 3: [ key, minute, color ]
13
14      return a[ kBGPMinute ] - b[ kBGPMinute ];
15
16  } // byMinuteOfDay
```

1	Define the number of minutes in a day.
2-4	Define the number of hex digits in a color.
5	Compile a regex to extract all the 2 hex-digit pairs that exist in a color string, and return them as an array. In this App, the array is length **`kHexLen/2`**.
6-8	Give mnemonics for the indices of a background palette section array of 3 and what they contain — Palette Editor keyword, name of JS variable specifying the minute of the day, color string for that minute.
10-16	Implement a background palette section array sort function. This sort function is a guard in case the background palette entries are not ordered properly — increasing minute order is required for the algorithms to work.

Listing 1
Preliminaries

```
1  function interpolateColors( begMin, begCol, endMin, endCol, cra ) {
2
3      let minuteDistance = endMin - begMin
4      if ( minuteDistance % 2 == 0 ) {
5          let begRGB = begCol.match( kHexColorRegEx )
6          let endRGB = endCol.match( kHexColorRegEx )
7          let midCol = '#'
8          for( let i = 0; i < kHexLen / 2; i++ ) {
9              let c = parseInt( begRGB[ i ], 16 ) + parseInt( endRGB[ i ], 16 )
10             midCol += Math.floor( c / 2 ).toString( 16 ).padStart( 2, '0' )
11         }
12         let midMin = begMin + ( minuteDistance / 2 )
13         cra[ midMin ] = midCol
14         interpolateColors( begMin, begCol, midMin, midCol, cra ) // left
15         interpolateColors( midMin, midCol, endMin, endCol, cra ) // right
16     } else {
17         cra[ begMin ] = begCol
18         cra[ endMin ] = endCol
19         if ( minuteDistance <= 1 ) return;
20         interpolateColors( begMin + 1, begCol, endMin, endCol, cra )
21     }
22
23 } // interpolateColors
```

1	The function is passed 5 arguments, 4 scalars from 2 background palette array entries and a reference to the colors result array `minuteToBackgroundColor[]`.
3	The number of colors to interpolate between **`begCol`** and **`endCol`**.
4	Check if this interval is even or odd — odd is how the recursion ends; for an even interval, e.g. 0-4, there is a precise midpoint, 2, handled by lines 5-15.
5-6	Split the colors into RGBA components and store in arrays of length 4.
7-11	Perform the actual interpolation and build **`midCol`**. Loop — convert the hex digits from strings to integers and sum them, find the component's midway value, convert back to a hex string padded with a leading zero if required, and suffix the result to **`midCol`**.
12-13	Compute the midway minute **`midMin`** and store the interpolated color in the result array.
14-15	Recurse the intervals left and right of **`midMin`** and **`midCol`**.
17-20	Handle an odd interval — update the interval's endpoint colors, terminate recursion if the interval has vanished, else continue recursion starting with a new, even, interval.

Listing 2 - **`interpolateColors()`**
Recursively Find Intermediate Colors

```
 1  function setupMinuteToBackgroundColor() {
 2
 3      minuteToBackgroundColor = []
 4      backgroundPalette.sort( byMinuteOfDay )
 5      let lrre = backgroundPalette.length - 1 // last rectangle's right edge
 6
 7      for ( let i = 0; i < lrre; i++ ) {
 8          let [ key0, begMin, begCol ] = backgroundPalette[ i + 0 ]
 9          let [ key1, endMin, endCol ] = backgroundPalette[ i + 1 ]
10          interpolateColors(
11              Math.round( begMin ), begCol, // rectangle's left edge
12              Math.round( endMin ), endCol, // rectangle's right edge
13              minuteToBackgroundColor       // colors result array
14          )
15      }
16
17      let [ key0, begMin, begCol ] = backgroundPalette[ 0 ]
18      let [ key1, endMin, endCol ] = backgroundPalette[ lrre ]
19      for ( let i = 0; i < kLastMinute; i++ ) {
20          if ( i < begMin ) minuteToBackgroundColor[ i ] = begCol
21          if ( i > endMin ) minuteToBackgroundColor[ i ] = begCol
22      }
23
24  } // setupMinuteToBackgroundColor
```

3-5	Initialize — clear the 1440 element array of background colors, sort `backgroundPalette[]` by time, and save the palette array index of the right edge of the rightmost virtual rectangle. We'll learn why shortly.
7-15	Loop — logically construct a virtual palette rectangle using the left edge of one palette array of 3, combined the right edge of the next palette array of 3, and call `interpolateColors()`. Each call stores the virtual palette rectangle's minute distance worth of colors in `minuteToBackgroundcolor[]`.
17-22	The revised background palette array does not require an explicit final entry that links back in time and color to the first entry. This code does that implicitly — minutes before the first palette rectangle, left edge, and after the last palette rectangle, right edge, where the daily cycle wraps / repeats, are dual in color.

Listing 3 - `setupMinuteToBackgroundColor()`
Populate `minuteToBackgroundColor[]`

The Palette Editor

Instead of repeatedly hacking the App's source code trying to create a palette that matches your imagination, a better way is to write a GUI for visualizing and testing a palette. The Palette Editor (PE), Figure 2, assists by providing these capabilities:

- Open / visualize an existing background palette
- Design a palette
- Test a palette
- Save / package a palette

PE is an Objective-C / AppKit class that talks bi-directionally with the JavaScript App via:

- WebKit messaging using strings
- `evaluateJavaScript:` using objects and strings

Typically, the item that's passed back and forth is a `backgroundPalette` object. When the direction is JS → Obj-C `evaluateJavaScript:` translates the JS object to an NSObject. In the Obj-C → JS direction, PE converts the NSObject to a properly formatted string representation of the JS object before handing off to `evaluateJavaScript:`.

NOTE: background palettes are now named (e.g. Earth) and have other associated metadata, so soon you'll see a background palette *dictionary* — of course one of the dictionary's keys is `@"backgroundPalette"`, whose value is a standard background palette array.

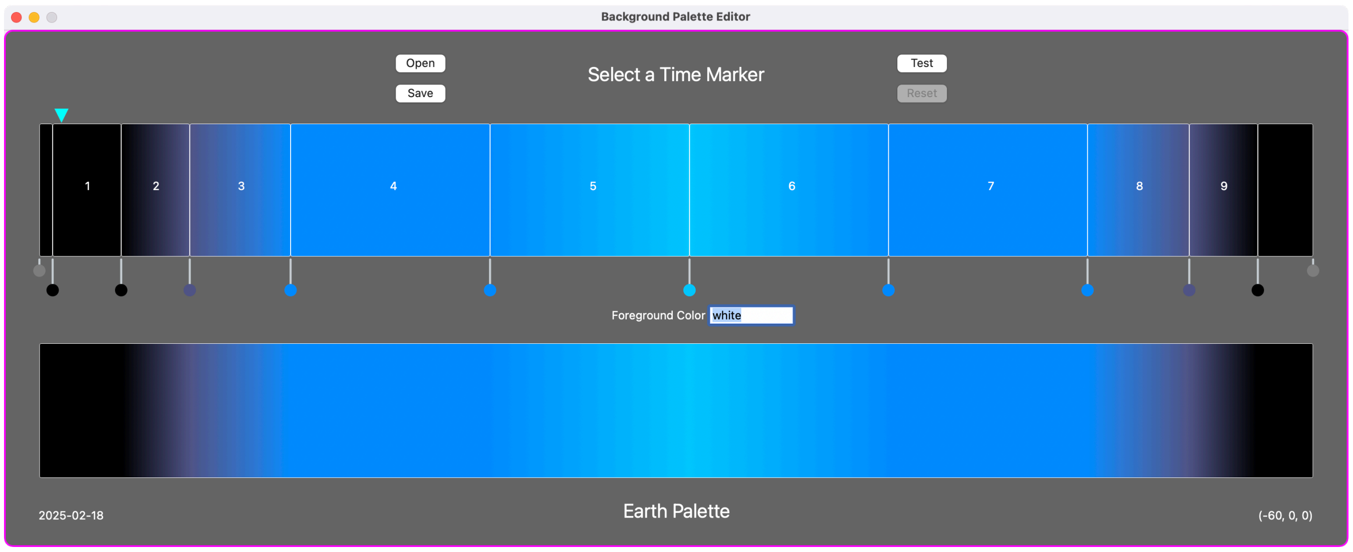


Figure 2
The Default Palette, Earth

A background palette array has increased in size because there are 4 additional time-color sections invented just for the App — here is Earth's new palette:

```
    var backgroundPalette = [
        [ "solarMidnight",  solarMidnight,   "#000000FF" ],
        [ "preDawn",        preDawn,         "#000000FF" ],
        [ "dawn",           dawn,            "#3F4174FF" ],
        [ "sunrise",        sunrise,         "#0475FFFF" ],
        [ "postSunrise",    postSunrise,     "#0475FFFF" ],
        [ "solarNoon",      solarNoon,       "#04BAFFFF" ],
        [ "preSunset",      preSunset,       "#0475FFFF" ],
        [ "sunset",         sunset,          "#0475FFFF" ],
        [ "dusk",           dusk,            "#3F4174FF" ],
        [ "postDusk",       postDusk,        "#000000FF" ],
    ];
```

10 palette entries corresponding to the 10 circular time marker points in Figure 2, that create 9 virtual palette rectangles. The two, shorter, gray time markers are static and represent the same minute, they are dual in time.

Each palette rectangle is assigned a color gradient with a single color stop; at the bottom of the editor is a different rendition of the background palette, this time a single view with a multi-stop color gradient — all generated from the same background palette array! Because these CAGradientLayer objects are so similar, PE creates them all from a template gradient, `self.gl0`:

```
    CAGradientLayer *gl0 = [CAGradientLayer layer];
    gl0.startPoint = CGPointMake(0.0, 0.5);
    gl0.endPoint = CGPointMake(1.0, 0.5);
    gl0.type = kCAGradientLayerAxial;
    gl0.locations = [NSArray arrayWithObjects:@0.0, @1.0, nil];
    gl0.borderWidth = 0.5;
    gl0.cornerRadius = 2;
    gl0.borderColor = [NSColor whiteColor].CGColor;
    self.gl0 = [NSKeyedArchiver archivedDataWithRootObject:gl0
        requiringSecureCoding:NO error:nil];
```

PE maintains two pools of objects that are configured at runtime. `self.vpr` is an array of NSView's that represent virtual palette rectangles, whose initial position and size are unknown. `self.tmv` is an array of circular TimeMarkerViews that point to the rectangles' background palette minute, but whose horizontal position is also unknown.

PE also instantiates NSColorPanel, obviously.

Obj-C Requests a JS Dictionary

Before PE can do anything it requests the App's current background palette, which arrives as an NSDictionary. The section **Palette Packaging, HTML, Bootstrapping** fully describes the dictionary, for now just know that the App maintains a list of several background palettes for the user to choose from, and that the first slot is reserved for PE testing:

```
let paletteDictionary = {} // a single palette
var paletteList       = [] // array of paletteDictionary objects
paletteList.push( { "paletteName" : "Test" } ) // PE test palette
```

PE calls the JS function `fetchBackgroundPalette()`, which returns the active palette dictionary / active background palette array (assumed to be Earth). PE configuration is finalized in the `evaluateJavaScript:` completion block:

- Generate the lower gradient view with 10 color stops, plus 1 "wrap" stop. This view depicts the original unmodified palette and is what the upper, editable, palette reverts to if **Reset** is clicked. See Listing 4.

- Reposition the time markers' x-coordinates to coincide with their minute of the day and set their fill color. See Listing 5.

- Reposition the 9 virtual palette rectangles' left edge to match its minute pointer and re-size its width to its new minute distance. Use 2 additional views to logically "wrap" the virtual rectangles. Add a 1-stop color gradient to each rectangle. See Listings 6a and 6b.

`self.wv` is the WkWebView:

```
[self.wv evaluateJavaScript:@“fetchBackgroundPalette ()"
         completionHandler:^(NSDictionary *pd, NSError *e) {

    self.backgroundPalette = pd[ @"backgroundPalette" ];

    [self resetGradientView];      // Listing 4
    [self resetTimeMarkerViews];   // Listing 5
    [self resetPaletteRectangles]; // Listings 6a and 6b

}];


function fetchBackgroundPalette () {

    paletteDictionary = paletteList[ activePaletteOrdinal ]
    paletteDictionary[ "backgroundPalette" ] = backgroundPalette
    return paletteDictionary

} // fetchBackgroundPalette
```

When these steps are finished both the upper 9+2 views (each with a 1-stop color gradient), and lower single view (with a (10+1)-stop color gradient) should be visually dual, and Figure 2 shows they are. Note each background palette rendering is 1440 pixels in width.

```
 1  - (void) resetGradientView {
 2
 3      NSMutableArray *cols = [NSMutableArray arrayWithCapacity:kBGPMaxViews];
 4      NSMutableArray *locs = [NSMutableArray arrayWithCapacity:kBGPMaxViews];
 5      NSArray *section;
 6      for ( NSInteger i = 0; i < [self.backgroundPalette count]; i++ ) {
 7          section = self.backgroundPalette[ i ];
 8          cols[ i ] = [self colorFromHexString:section[ kBGPColor ]];
 9          double xPos = [section[ kBGPMinute ] doubleValue];
10          locs[ i ] = [NSNumber numberWithDouble:xPos / kLastMinute];
11      }
12      [locs addObject:@1.0];
13      [cols addObject:cols[ 0 ]]; // wrap tail to head
14
15      CAGradientLayer *gl = [NSKeyedUnarchiver
16          unarchivedObjectOfClass: [CALayer class]
17                         fromData: self.gl0
18                            error: nil
19      ];
20      gl.colors = cols;
21      gl.locations = locs;
22      self.gradientView.layer = gl;
23
24  } // resetGradientView
```

3-4	Define an empty list of colors and their stop locations in the view, left edge is 0.0, right edge is 1.0. Colors are type CGColor, locations are type NSNumber.
6-11	Loop — using each background palette entry array of 3, append a new color to the `cols` array, and the color's fractional position along the view to the `locs` array.
12-13	Add a final stop at the view's right edge that wraps the first gradient color to it.
15-22	Initialize a fresh color gradient object, configure it's colors and locations, then add it to the gradient view.

Listing 4 - `resetGradientView`
One View With 10+1 Gradient Stops

```
 1  - (void) resetTimeMarkerViews {
 2  
 3      CGRect bgpFrame = self.bgpView.frame;
 4      for ( NSInteger i = 0 ; i < [self.backgroundPalette count]; i++ ) {
 5          NSArray *section = self.backgroundPalette[ i ];
 6          double xPos = [section[ kBGPMinute ] doubleValue];
 7          TimeMarkerView *tmv = self.tmv[ i ];
 8          CGRect f = tmv.frame;
 9          f.origin.x = bgpFrame.origin.x + xPos - ( f.size.width / 2.0 );
10          tmv.frame = f;
11          tmv.title = section[ kBGPKey ];
12          NSString *color = section[ kBGPColor ];
13          id fc = [self colorFromHexString:color];
14          tmv.fillColor = [NSColor colorWithCGColor:(__bridge CGColorRef)(fc)];
15          NSString *tt = [NSString stringWithFormat:@"%@ %d", tmv.title,
16              (int)roundf( xPos ) % kLastMinute];
17          [tmv setToolTip:tt];
18          [tmv setNeedsDisplay:YES];
19          NSClickGestureRecognizer *click = [[NSClickGestureRecognizer alloc]
20              initWithTarget:self action:@selector(timeMarkerTapped:)];
21          [tmv addGestureRecognizer:click];
22      }
23  
24  } // resetTimeMarkerViews
```

3	`self.bgpView` is the view containing all the virtual palette rectangles.
4-22	Loop — using each background palette entry array of 3, reposition the time marker and configure it.
6-10	Position the time marker to the left edge of its palette rectangle.
11-17	Set its title, color and tooltip. Hovering over the marker displays its name and minute.
19-21	Clicking on the time marker prepares its palette rectangle for editing.
	Not shown — the two gray markers are positioned at minute coordinates 0 and 1440.

Listing 5 - `resetTimeMarkerViews`
Configure and Position Time Markers on the Minute Axis

```
1  - (void) resetPaletteRectangles {
2
3      CGRect f;
4      id wrapColor = nil;
5      NSInteger bgpCount = [self.backgroundPalette count];
6      NSView *v;
7      for ( v in self.vpr ) { v.hidden = YES; }
8      for ( v in self.tmv ) { v.hidden = YES; }
9      for ( NSInteger i = 0; i < bgpCount; i++ ) {
10         v = self.tmv[ i ];
11         v.hidden = NO;
12         v = self.vpr[ i ];
13         v.hidden = ( i < bgpCount - 1 ? NO : YES );
14         NSArray *sec1 = self.backgroundPalette[ i ];
15         NSInteger next =  (i < bgpCount - 1 ? i + 1 : i);
16         NSArray *sec2  = self.backgroundPalette[ next ];
17         f = v.frame;
18         f.origin.x = [sec1[ kBGPMinute ] doubleValue];
19         f.size.width = [sec2[ kBGPMinute ] doubleValue] - f.origin.x;
20         v.frame = f;
21         id leftColor = [self colorFromHexString:sec1[ kBGPColor ]];
22         id rightColor = [self colorFromHexString:sec2[ kBGPColor ]];
23         CAGradientLayer *gl = [NSKeyedUnarchiver
24             unarchivedObjectOfClass: [CALayer class]
25                            fromData: self.gl0
26                               error: nil
27         ];
28         gl.frame = v.frame;
29         gl.colors = [NSArray arrayWithObjects:leftColor, rightColor, nil];
30         v.layer = gl;
31         if ( i == 0 ) wrapColor = leftColor;
32     }
```

3-8	Initialize — hide all virtual palette rectangles and time markers.
9-32	Loop — unhide a palette rectangle, position and size it, set its gradient.
14-20	A palette section defines a rectangle's x-coordinate. The rectangle's width is the minute distance to the next palette section's x-coordinate. Even thought the last rectangle has a width of zero, that's corrected in the wrap step, see Listing 6b.
21-30	Configure a new color gradient using colors from the left and right palette entries.
31	The wrap color linking the leftmost and rightmost palette rectangles. See Listing 6b.

Listing 6a - `resetPaletteRectangles`
9 Views Each With 1 Gradient Stop

```
33
34     NSView *nv; // neighbor view
35     for ( NSInteger i = bgpCount; i <= bgpCount + 1; i++ ) {
36         v = self.vpr[ i ];
37         f = v.frame;
38         NSTextField *tf = v.subviews[0]; tf.hidden = YES;
39         if ( i == bgpCount ) { // to be the leftmost palette rectangle
40             nv = self.vpr[0]; // right neighbor's left edge
41             f.origin.x = 0;
42             f.size.width = nv.frame.origin.x;
43         } else { // to be the rightmost palette rectangle
44             nv = self.vpr[ bgpCount - 1 ]; // left neighbor's right edge
45             f.origin.x = nv.frame.origin.x + nv.frame.size.width;
46             f.size.width = kLastMinute - f.origin.x;
47         }
48         v.frame = f;
49         v.hidden = NO;
50         CAGradientLayer *gl = [NSKeyedUnarchiver
51             unarchivedObjectOfClass: [CALayer class]
52                            fromData: self.gl0
53                               error: nil
54         ];
55         gl.frame = v.frame;
56         gl.colors = [NSArray arrayWithObjects:wrapColor, wrapColor, nil];
57         v.layer = gl;
58     }
59
60 } // resetPaletteRectangles
```

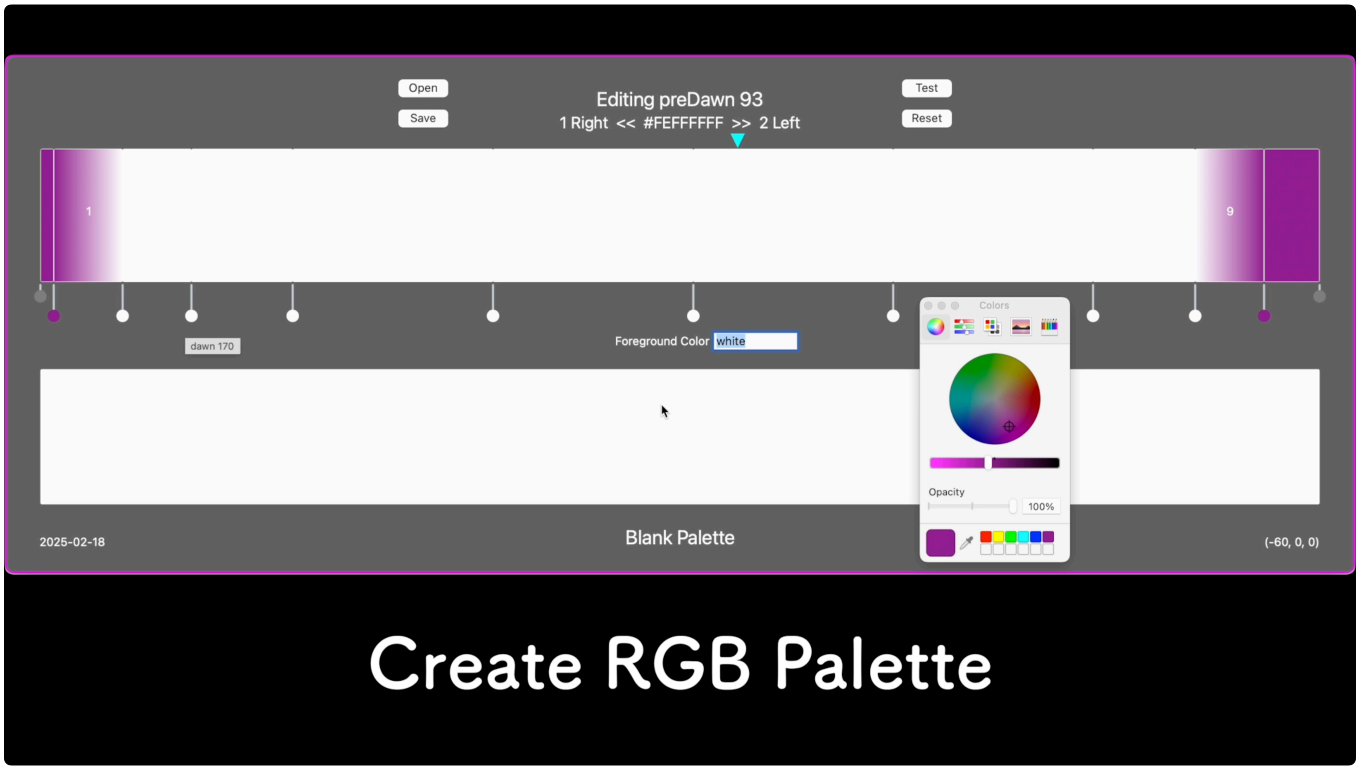
35-58	Loop — add 2 views to wrap the color joining SolarMidnight with PostDusk. Analogous to the wrapping in Listing 3, lines 17-22.
36	Grab an unused NSView from the virtual palette rectangle pool.
40-42	The left wrap view, filling minutes 0 to solarMidnight.
44-46	The right wrap view, filling minutes postDusk to 1439.
50-57	Configure a new color gradient with the left and right palette colors.

Listing 6b - `resetPaletteRectangles`
2 Wrap Views Each With 1 Gradient Stop

Editing a Background Palette

To edit the default palette, click a time marker and then select a color — that would make an edit to the Earth palette. Not to worry, you're working on a copy of the palette, you haven't saved the change to a file or pushed it back to the App.

To edit a different palette use Open, the App has a number of palettes you could choose to edit. One is Blank, a great palette to begin work on something new, perhaps a simple light spectrum, that will eventually be named RGB, so go ahead, click on Movie 1, Create RGB Palette:



Movie 1
Create RGB Palette

Briefly (ignoring wrapping), each time a color is selected it's applied to both the left view's right color and the right view's left color:

```
gradientLayer = (CAGradientLayer *)lv.layer; // left view
colors = [NSMutableArray arrayWithArray:gradientLayer.colors];
colors[ 1 ] = (id)newColor.CGColor;
gradientLayer.colors = colors;

gradientLayer = (CAGradientLayer *)rv.layer; // right view
colors = [NSMutableArray arrayWithArray:gradientLayer.colors];
colors[ 0 ] = (id)newColor.CGColor;
gradientLayer.colors = colors;
```

Obj-C Pushes an NSDictionary to JS

PE is not in a position to actually test a new background palette, that's the responsibility of the App. PE just transmits the new background palette and the sim starts using it immediately. Of course, sending JS an actual NSDictionary cannot work, but converting it to a string representation of a JS dictionary can — and that's exactly what happens. JS then evals the string-ified JS object to create an actual object.

The JS strings are multi-line for readability, so quoting uses template string backticks. Here's what the string-ified JS RGB background palette dictionary looks like (only two keys are shown, the rest are described in the section **Palette Packaging, HTML, Bootstrapping**):

```
paletteDictionary = {
    "paletteName"       : "RGB",
    "backgroundPalette" : \`backgroundPalette = [
     [ "solarMidnight",       solarMidnight,    "#7F007FFF" ],
     [ "preDawn",             preDawn,          "#FB0106FF" ],
     [ "dawn",                dawn,             "#FB0106FF" ],
     [ "sunrise",             sunrise,          "#FB0106FF" ],
     [ "postSunrise",         postSunrise,      "#FEFE0AFF" ],
     [ "solarNoon",           solarNoon,        "#21FE06FF" ],
     [ "preSunset",           preSunset,        "#20FEFEFF" ],
     [ "sunset",              sunset,           "#0000FEFF" ],
     [ "dusk",                dusk,             "#0000FEFF" ],
     [ "postDusk",            postDusk,         "#7F007FFF" ],
    ]\`,
}
```

Notice the value assigned to its "`backgroundPalette`" key is a JS array enclosed in ***escaped backticks***. Assume this string-ified JS dictionary is the value of the Obj-C NSString variable `*pd`, then when used as a parameter to `testBackgroundPalette()` the entire dictionary is enclosed in ***non-escaped backticks***:

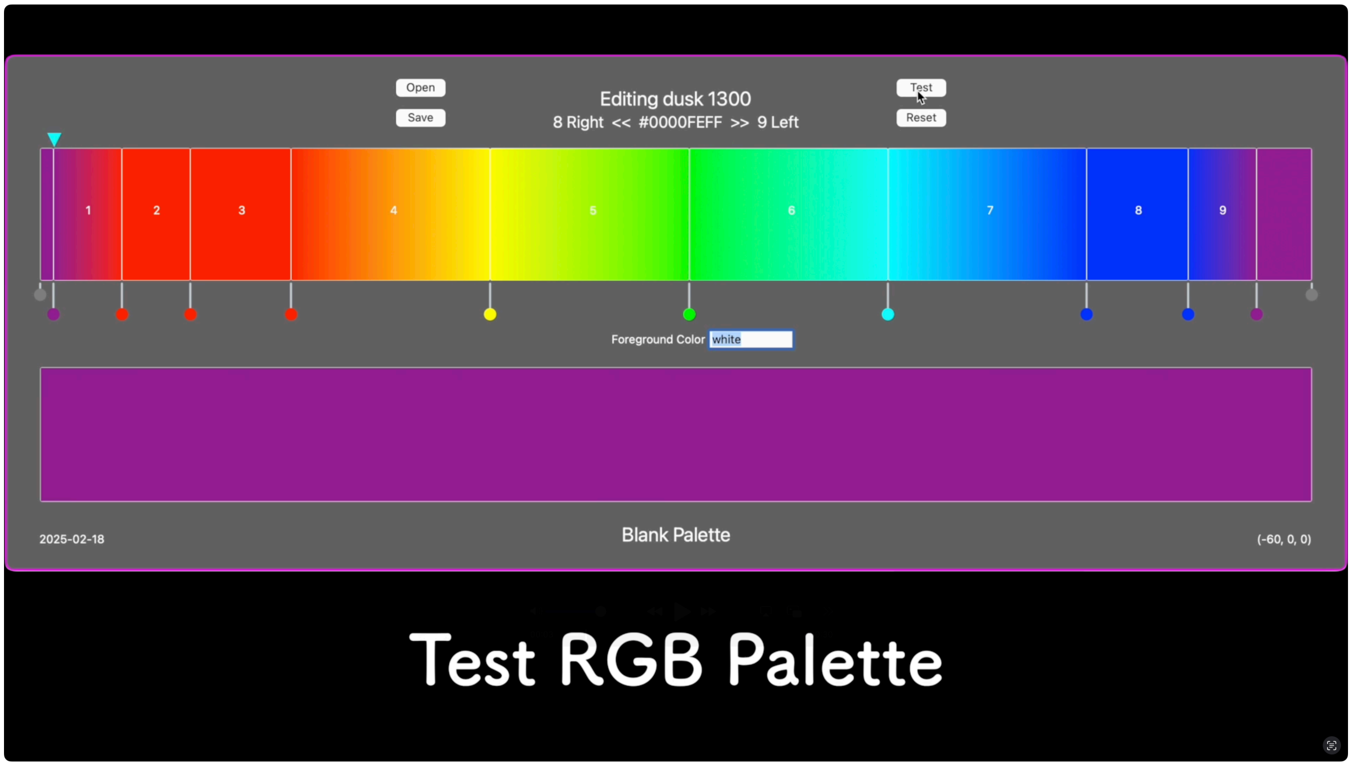
```
NSString *js = [NSString
    stringWithFormat:@"testBackgroundPalette( `%@` );", pd];
[self.wv evaluateJavaScript:js
         completionHandler:^(id rv, NSError *e) {
}];

function testBackgroundPalette ( newBGPDictionary ) {

   eval( newBGPDictionary ) // create paletteDictionary
   activePaletteOrdinal = 0 // reserved slot in paletteList
   paletteList[ activePaletteOrdinal ] = paletteDictionary
   eval( paletteDictionary[ "backgroundPalette" ] )
   setupMinuteToBackgroundColor() // using backgroundPalette

} // testBackgroundPalette
```

Testing a Background Palette

In **`testBackgroundPalette()`** the first **`eval()`** creates an actual JS dictionary named **`paletteDictionary{}`**, which is stored in the **`paletteList[]`** array (in its reserved slot) and made the active palette. Note that this **`eval()`** effectively removes the escape character from the **`"backgroundPalette"`** member, in preparation for the second **`eval()`**.

The second **`eval()`** creates an actual JS array named **`backgroundPalette[]`**, including evaluating the JS minute variables, and then **`setupMinuteToBackgroundColor()`** is called (see Listing 3).

The code to place the App is sim mode on the new background palette isn't shown, it's not relevant here — but the test playback is! While testing, the lower PE gradient view is usurped and used to display the changing background in real-time, so go ahead, click on Movie 2, Test RGB Palette:



Movie 2
Test RGB Palette

As the animation plays through its simulated minutes, JS sends WebKit messages to Obj-C with a minute value and its associated background color. PE uses this data to move the triangular minute pointer and to set the gradient view's start and stop colors to the color of the minute. Watch 24 hours of sim-time background color changes compressed to 30 real-time seconds.

JS Messages Obj-C

When the App's view controller creates the WkWebView it configures a user content controller that specifies a message name that JS is allowed to invoke. The view controller must also conform to the `<WKScriptMessageHandler>` protocol to receive the message:

```
WKWebViewConfiguration *config =
    [[WKWebViewConfiguration alloc] init];
[config.userContentController
    addScriptMessageHandler:self name:@"showTestPalette"];
self.wv = [[WKWebView alloc] initWithFrame:self.view.frame
    configuration:config];
[self.view addSubview:self.wv];
```

The JS simulation Mainloop calls the following function for each simulated minute, which posts a **showTestPalette** message to Obj-C, containing a simple "minute=color" string:

```
function updatePaletteEditor( minute ) {

    window.webkit.messageHandlers.showTestPalette.
        postMessage(
            minute + '=' +
            minuteToBackgroundColor[ minute ]
    )

} // updatePaletteEditor
```

The Obj-C delegate receives the message, verifies it's name, and forwards the NSString to PE:

```
- (void)userContentController:(WKUserContentController *)
                                              userContentController
      didReceiveScriptMessage:(WKScriptMessage *)message {

    if ([message.name isEqualToString:@"showTestPalette"]) {
        [self.ped showTestPalette:message.body];
    }

} // userContentController:didReceiveScriptMessage
```

And, finally, PE can update its views. If the extra hop from the view controller to PE is bothersome, think of the alternative: polling, ugh.

```
- (void) showTestPalette:(NSString *)msgBody {

    NSArray *t = [msgBody componentsSeparatedByString: @"="];
    [self movePETriangle:[t[ 0 ] integerValue]];
    id c = [self colorFromHexString:t[ 1 ]];
    CAGradientLayer *gl = [NSKeyedUnarchiver
        unarchivedObjectOfClass: [CALayer class]
                       fromData: self.gl0
                          error: nil
    ];
    NSArray *colors = [NSArray arrayWithObjects:c, c, nil];
    gl.colors = colors;
    self.gradientView.layer = gl;

} // showTestPalette
```

Palette Packaging, HTML, Bootstrapping

Once testing is complete, clicking the PE Save button writes two JS statements to a file (e.g. RBG.js) on the host computer’s filesystem, see Listing 7.

```
1   paletteDictionary = {
2       "author"            : "S. O. Lidie",
3       "backgroundPalette" : `backgroundPalette = [
4        [ "solarMidnight",      solarMidnight,       "#7F007FFF" ],
5        [ "preDawn",            preDawn,             "#FB0106FF" ],
6        [ "dawn",               dawn,                "#FB0106FF" ],
7        [ "sunrise",            sunrise,             "#FB0106FF" ],
8        [ "postSunrise",        postSunrise,         "#FEFE0AFF" ],
9        [ "solarNoon",          solarNoon,           "#21FE06FF" ],
10       [ "preSunset",          preSunset,           "#20FEFEFF" ],
11       [ "sunset",             sunset,              "#0000FEFF" ],
12       [ "dusk",               dusk,                "#0000FEFF" ],
13       [ "postDusk",           postDusk,            "#7F007FFF" ],
14      ]`,
15      "base64PNG-160x18"  : "iVBORw0KGgoAAAANSUhE ...
                               ... UgAAAvZ+97UVORK5CYII=",
16      "foregroundColor"   : "white",
17      "paletteName"       : "RGB",
18      "version"           : "1.0",
19      "when"              : "2025-02-24",
20      "where"             : "(-60, 0, 0)",
21   }
22   paletteList.push( paletteDictionary )
```

1-21	Define `paletteDictionary{}`.
3-14	RGB’s background palette array.
15	Icon-sized image of the palette’s NSViews.
16	Fill color of any lettering painted over the palette view.
19-20	Simulated date and GPS coordinates of palette creation.
22	Append `paletteDictionary{}` to `paletteList[]`.

Listing 7 - `RGB.js`
A Palette Dictionary Ready for Installation

Ensure the App has access to RGB.js and source it in the HTML **`<head>`** section, after the background palette declarations but before the App itself:

```
<head>

     <!-- Background Palette
      Each .js src is a paletteDictionary{} that pushes itself
      onto paletteList[]. The 0-th palette entry is reserved
      for Palette Editor's test palette.
     -->
     <script>
         let paletteDictionary = {}
         var paletteList       = []
         paletteList.push( { "paletteName" : "Test" } )
     </script>
     <script type="text/javascript" src="Earth.js"></script>
     <script type="text/javascript" src="RGB.js"  ></script>
     <!-- Background Palette ->

     <script type="text/javascript" src="App.js"  ></script>

 </head>
```

Now that the App has a **`paletteList[]`** array full of **`paletteDictionary{}`** creatures, simply change:

```
backgroundPalette[] = [ [ … ], ]
```

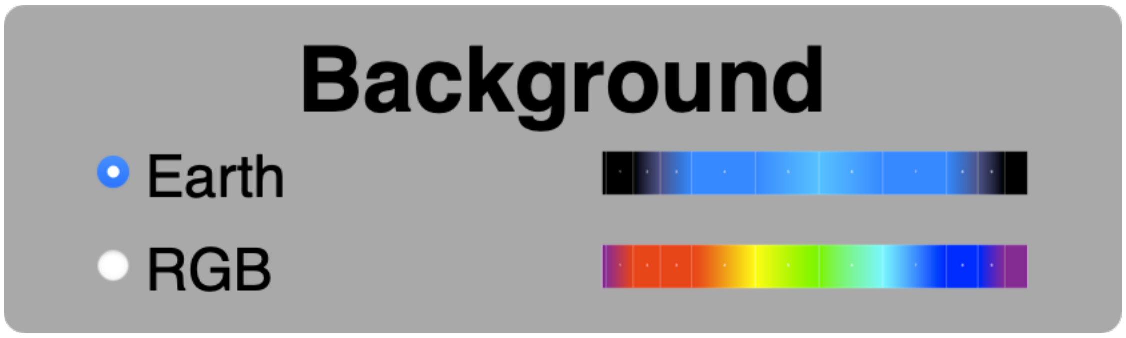
to this:

```
activePaletteOrdinal = 1
paletteDictionary = paletteList[ activePaletteOrdinal ]
eval ( paletteDictionary[ "backgroundPalette" ] )
```

and the cutover is complete.

Elsewhere, the App provides a customization panel that's generated by iterating `paletteList[]` and creating DOM radio buttons on-the-fly from the palette dictionary metadata `"paletteName" and "base64PNG-160x18"`. Notice the test palette in slot zero is skipped:

```
// Background palette radio buttons.

for ( n = 1; n < paletteList.length; n++ ) {
    makeRadiobutton( paletteList[ n ] );
}
```



On a click the App finds the checked radio button's palette name to get the palette dictionary and initializes the background colors, as usual:

```
pn = document.querySelector('input[type="radio"]:checked').value;
for ( let p = 1; p < paletteList.length; p++ ) {
    let pd = paletteList[ p ]
    if ( pd[ "paletteName" ] != pn ) { continue }
    activePaletteOrdinal = p
    eval ( pd[ "backgroundPalette" ] )
    setupMinuteToBackgroundColor( )
    break
}
```

Logical Dualities

The background palette is a singleton used by multiple languages for multiple purposes. Consider the Earth palette, abstractly:

- In the context of the JS App, the background palette describes, visually, a fixed space with color changing in time. Given any App background space, as you examine a pixel, any pixel, you see the same color, until there's a time change — you wait a moment.

- In the context of the Obj-C PE, the background palette describes, visually, a fixed time with color changing in space. Given virtual palette rectangles with height = 1, as you examine a pixel, any pixel, you see the same color, until there's a space change — you shift your gaze.

- The two descriptions are dual, they represent the same object.

- The background palette simultaneously defines one space with a single multi-stop color gradient, as well as multiple spaces each with a one-stop color gradient.

- By definition, the background palette requires that its rightmost edge identify with its leftmost edge. This is what has been called wrapping — or the right edge of the palette is time-color dual with the left edge. This requirement was implemented variously, see Listing 3, Listing 4 and Listing 6b.

- This time-color edge duality creates a virtual background palette that is finite yet unbounded in time and color.