In this article I will explain the implementation of an interface using HTML5 and JavaScript involving orbiting images and affording players/users the facility to click on images to bring them forward and present explanatory text. The script makes use of techniques that can be applied to a wide variety of applications.

My daughter, my toughest client, asked me to put together an application in which a set of images move in an orbit on and off screen. She wanted the orbit to not be strictly circular. One image remains in a fixed position in the lower right corner. The orbiting images grow in size as they come to the bottom of the screen as if emerging from far away and getting closer. Her idea was that these would resemble items on a lazy Susan type of tray holding goodies that may be present at a tea party. Figure 1 shows a screen shot. The images would be characters in the cast of a play, The Bachelor’s Tea Party by Stolen Chair (http://stolenchair.org), and text would appear holding brief descriptions of the cast members. She ended up not using the application, but I still enjoyed making it. The techniques I will explain (parameterized equations, coordinate transformations, parallel structures, application state and function called/calling tables) are applicable to a variety of challenges.

After all 8 images have cycled around twice, movement stops with the first image stopping at the bottom, with accompanying text. Figure 2 shows the screen at this state.

The player/viewer can click on any visible image and this will cause that image to cycle to the front and have its accompanying text displayed. Figure 3 shows a screen shot of what happens when the player/viewer clicks on the image of Annika, held by her father’s arm.

So what are the critical features of this application? My first focus was expressing the orbit. I was comfortable with animation, but it
took a little while to realize that I wanted an elliptical orbit, with
the center towards the lower right portion of the canvas, and that
the standard form parameterized equations for an ellipse would
produce what I needed. The second technique I brought to this
application was putting together a set of arrays to hold critical
information about each image. The technique is called parallel
structures and you might consider it a poor man’s substitute for
objects. Lastly, the challenge of this application involved different
application states: the initial orbiting, the images not in motion,
and the images in orbit until a certain one comes to the front. I
didn’t apply any general methodology beyond variable settings
for detecting application state, but it was a useful concept to guide
my programming. Each set of techniques has general utility for
games and other categories of applications. I will expand on each
of the three concepts. You can use the link given in the Learn More
section to see all the code.

Parameterized equations
You may remember the equation for an ellipse as being

\[(X/a)^2 + (Y/b)^2 = 1\]

This ellipse is centered at the origin with axes parallel to the x and
y axes. The equation for a rotated ellipse is as follows (I am writing
in a mixture of mathematics (superscripts), and JavaScript (* for
multiplication and Math.sin and Math.cos for the trig functions):)

\[(X\times Math.cos(f) + Y\times Math.sin(f))^2/a^2 + (X\times Math.sin(f) - Y\times Math.cos(f))^2/b^2 = 1\]

where \(f\) is the angle of rotation from the horizontal.

There are other canonical forms for a translated ellipse, that is, not
centered at the origin, or translated and rotated. However, it is not
clear how to use these equations to draw and re-draw objects on
the screen. What is needed for an application such as this one is
something called parameterized equations, namely an equation
for the horizontal (x) and tan equation for the vertical (y) position,
each in terms of a value termed the parameter. The parameterized
equations for an ellipse centered at the point originx, originy, and
rotated an angle of \(f\) radians from the horizontal are

\[X(t) = \text{origin}x + a\times \text{Math.cos(t)}\times \text{Math.cos(f)} - b\times \text{Math.sin(t)}\times \text{Math.}
\cos(f)\]

\[Y(t) = \text{origin}y + a\times \text{Math.cos(t)}\times \text{Math.sin(f)} - b\times \text{Math.sin(t)}\times \text{Math.}
\cos(f)\]


The parameter typically is written as \(t\) to convey that the positions
are calculated in terms of time. For my example, the parameter
corresponds to an angle, with the angle incremented by a set
amount held in the variable factor for each iteration of the
animation. The animation is set up in the typical way by a call to
`setInterval`. The values of \(\text{cos}(f)\) and \(\text{sin}(f)\) are computed once
only and stored as variables: costh and sinth.

Now, again for this application, my code must calculate a position
for not one but 8 images. I set up several arrays for this information
(see the next section). Lastly, though JavaScript does not complain
if values are supplied to the `drawImage` method that are too big for
the canvas, it does complain about values that are too small. This is
the reason for the calls to `Math.max`. Values that are off screen are
simply ignored. Listing 1 shows the `travel` function. The canvas
is cleared. An image of my daughter is drawn in the lower right
corner and then the for-loop computes the x and y position as well
as the size (I call it scale) of each of the images. Notice that the
statement for setting the y values have the opposite signs from the
parametric equation because of the upside down (vertical values
increasing down the page) coordinate system used for the canvas.

```javascript
function travel() {
    ctx.clearRect(0, 0, 900, 600);
    ctx.drawImage(aviva, 700, 400);
    for (var i=0; i<images.length; i++) {
        angle[i] = angle[i]+factor;
        costa[i] = Math.cos(angle[i]);
        sinta[i] = Math.sin(angle[i]);
        x[i] = Math.max(0, originx + a*costa[i]*costh
- b*sinta[i]*sinth);
        y[i] = Math.max(0, originy - a*costa[i]*sinth
- b*sinta[i]*costh);
        scale[i] = 40 + (y[i]/400) * 150;
        ctx.drawImage(images[i], x[i], y[i], scale[i], scale[i]);
    }
    keepgoing ();
}
```

Listing 1: The travel function

It is possible to draw an entire translated, rotated ellipse, not all
contained in the canvas, using these coordinate transformation
methods. Figure 4 shows such an ellipse. The rectangle
represents the canvas that is used in the orbiting application and the black
square represents the translated origin (center) of the ellipse. Notice
that the square is itself rotated. The canvas in the program that
produces Figure 4 is bigger.

![Figure 4: Translated, rotated ellipse](image)

The way to produce an ellipse as shown in Figure 4 is to apply
transformations to the coordinate system and then draw a circle.
Listing 2 shows the code.
Parameterized equations exist for other mathematical curves in 2-D and 3-D.

Once I remembered parameterized equations, the critical insight was that my code could draw outside the canvas, that is, send values that were greater than the canvas limits, without any harm. Some may argue that this is wasted computation, but testing for values being inside the canvas probably would take up more time.

Parallel structures

I decided to keep information relating to the file for each image, the accompanying text, the position and the scale of each of the images, in a set of arrays. I use arrays for what you may think of as the intermediate values as well. The arrays are `images`, `scale`, `x`, `y`, `costa`, `sinta`, `angle`, and `msgs`. The `images` and `msgs` do not change. The `images` array is constructed as shown in Listing 3.

<table>
<thead>
<tr>
<th>Function name and task</th>
<th>Called / Invoked by</th>
<th>Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>init</code> sets arrays and calls</td>
<td><code>setInterval</code> to start initial</td>
<td><code>setuptravel</code></td>
</tr>
<tr>
<td><code>setuptravel</code> invokes</td>
<td></td>
<td><code>init</code></td>
</tr>
<tr>
<td><code>setInterval</code> to start</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>addEventList</code> to start</td>
<td>action of <code>addEventList</code> in</td>
<td><code>overcheck</code></td>
</tr>
<tr>
<td><code>x</code></td>
<td><code>setInterval</code> to start</td>
<td></td>
</tr>
<tr>
<td><code>y</code></td>
<td><code>addEventList</code> in <code>enableclicks</code></td>
<td></td>
</tr>
<tr>
<td><code>costa</code></td>
<td><code>overcheck</code></td>
<td></td>
</tr>
<tr>
<td><code>sinta</code></td>
<td><code>overcheck</code></td>
<td></td>
</tr>
<tr>
<td><code>angle</code></td>
<td><code>overcheck</code></td>
<td></td>
</tr>
<tr>
<td><code>msgs</code></td>
<td><code>overcheck</code></td>
<td></td>
</tr>
<tr>
<td><code>travel</code></td>
<td>action of <code>setInterval</code> in</td>
<td></td>
</tr>
<tr>
<td><code>travel</code> is one check if</td>
<td><code>travel</code></td>
<td></td>
</tr>
<tr>
<td><code>startturn</code></td>
<td><code>travel</code></td>
<td></td>
</tr>
<tr>
<td>this is during the initial</td>
<td><code>enableclicks</code></td>
<td></td>
</tr>
<tr>
<td>orbit (first is true)</td>
<td><code>enableclicks</code></td>
<td></td>
</tr>
<tr>
<td>and another if waiting for a</td>
<td><code>enableclicks</code></td>
<td></td>
</tr>
<tr>
<td>image to come to the front.</td>
<td><code>enableclicks</code></td>
<td></td>
</tr>
<tr>
<td>This function draws the text</td>
<td><code>enableclicks</code></td>
<td></td>
</tr>
<tr>
<td>on the page.</td>
<td><code>enableclicks</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: Function called/calling table.

Listing 3: Code setting up our images array

The values in each of the other arrays change each iteration. As I said before, this technique, called parallel structures, is a way to manage sets of data. It is an alternative to objects. You can examine the full code (link given below) and decide when you want to make use of the technique. My decision to compute the `costh` and `sint` values ahead of time as well as create the `Image` objects was based on removing as much computation as possible from the animation.

Application state

Most applications can be said to be in distinct states at different times during the application. In this application the first state is the orbiting of the images, in which all the images come around 2 times. The next state has no animation going on, but an event is set up to 'listen for' a mouse click. When the mouse click happens, the handler for that event sets up the next state: animation occurs until a specific image is at the front. HTML5 and JavaScript did not provide me a direct way to program these states, but, as I indicated earlier, it was useful to articulate to myself what I wanted to happen. The states are defined by the values of certain variables, whether or not animation is happening as specified by a call of `setInterval` and whether event handling is set up for clicking on the canvas. Table 1 is a function calling/called by chart.
Keep in mind that there are many ways to define and use functions in an application. Using many smaller functions is better than using fewer, bigger ones, but it is your call on what to do. You also have the option of transforming (translating and rotating) the canvas for each image. I decided that calculating the positions and scale was preferable. I do recommend preparing a function table such as this one for documentation and also for reference during development.

**Learn more**

There are many sources, online and in-print and some sort of e-books, for learning HTML5 and JavaScript techniques. Here are links to my recent books and the website for this example.


- HTML5 and JavaScript Projects, [http://www.apress.com/9781430240327](http://www.apress.com/9781430240327). This book is more advanced than the first one. This has chapters on bouncing graphics, images and videos. Other chapters describe other ways to combine video and canvas, directions for making origami, using algebra and geometry as well as images and videos. The book also has uses of the Google Maps API, geolocation, and php and MySQL. Several of the chapters describe ways to build sets of applications, which can be applied to the orbiting images application.

- To see the orbiting images application in action and to view the source code, go to [http://faculty.purchase.edu/jeanine.meyer/html5/teapartytest.html](http://faculty.purchase.edu/jeanine.meyer/html5/teapartytest.html) and, when the images stop revolving, click on any visible image for it to move to the front.

Jeanine Meyer lives just north of New York City and currently teaches at Purchase College/SUNY after many years at IBM, doing research on robotics and manufacturing and consulting on educational grants. She likes providing programming examples for her Mathematics/Computer Science and New Media students and really, really likes working with images and video clips of her granddaughter and other family members.