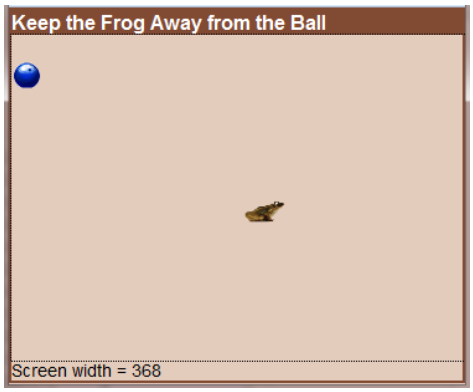
Responsive Animation with HTML5, CSS3 and the JavaScript Canvas

By Rob Robinson

Abstract



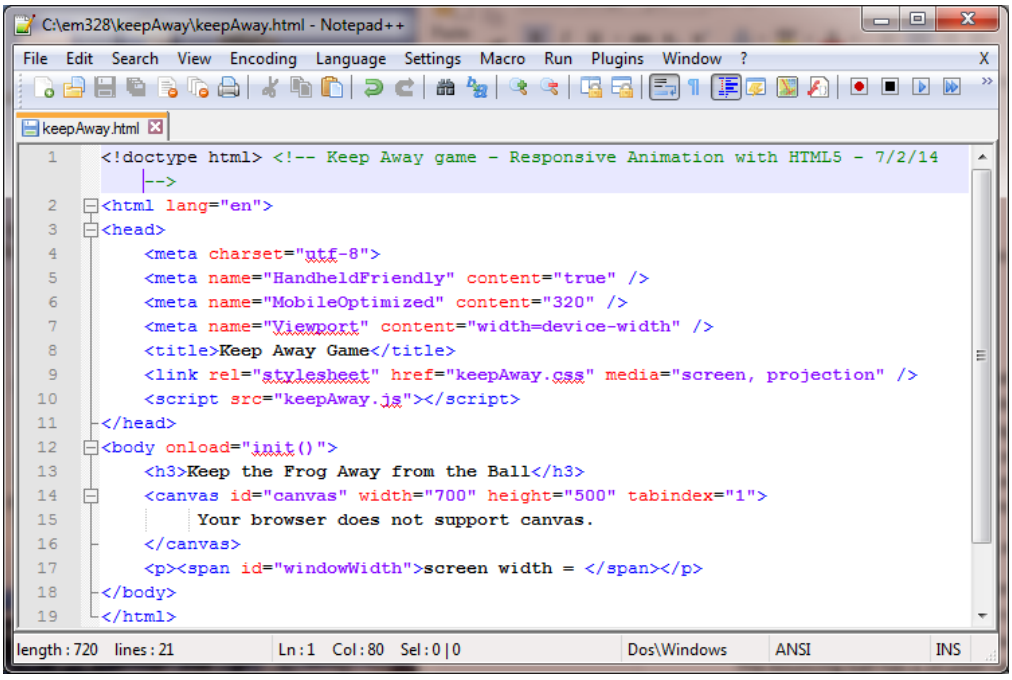
Creating HTML5 games and animated applications that run in any recent browser on any desktop, laptop, tablet or smart phone requires planning and a few special techniques. Topics covered are: Planning the scenes and interactions, designing the layout, detecting the device and window size, sizing the canvas, scaling the graphics, performing animations, responding to keyboard and mouse events on laptops and desktops, responding to touch events on tablets and smart phones, and completing the game with appropriate interactions. A simple "Keep Away" game will be used to demonstrate the techniques.

Planning the Ideal Window Size, scenes and Interactions

The ideal window size depends on the size and quality of the graphics to be used in the application and the area needed by the application for the various interactions involved in the entire process. Our example application will involve a "keep away" game between a computer-controlled bouncing ball and a user-controlled hopping frog. The bouncing ball has a 20 pixel radius and the hopping frog is around 80 pixels wide by 40 pixels high. At the end of its bounce, the ball can detect the location of the frog and its next bounce will change to that direction. The user must control the frog's direction using mouse, keyboard or touch events. The goal is to keep the frog from being squished by the ball. (See page 3 for thrilling scenes of the game.) A 700 x 500 pixel playing field will be ideal for devices that can provide at least that much space. There is no need for a playing field larger than that, regardless of screen size. A simple, single-color background will be used with no scrolling.

Laying out the HTML5 Page

Page layout is much easier, if you take full advantage of the new syntax and structure techniques of HTML5 and if you use the formatting features of CSS3 as much as possible. The code on the left shows a minimum amount of code needed in HTML to include a simple heading at the top of the page with a 700x500 default-sized canvas beneath the heading.



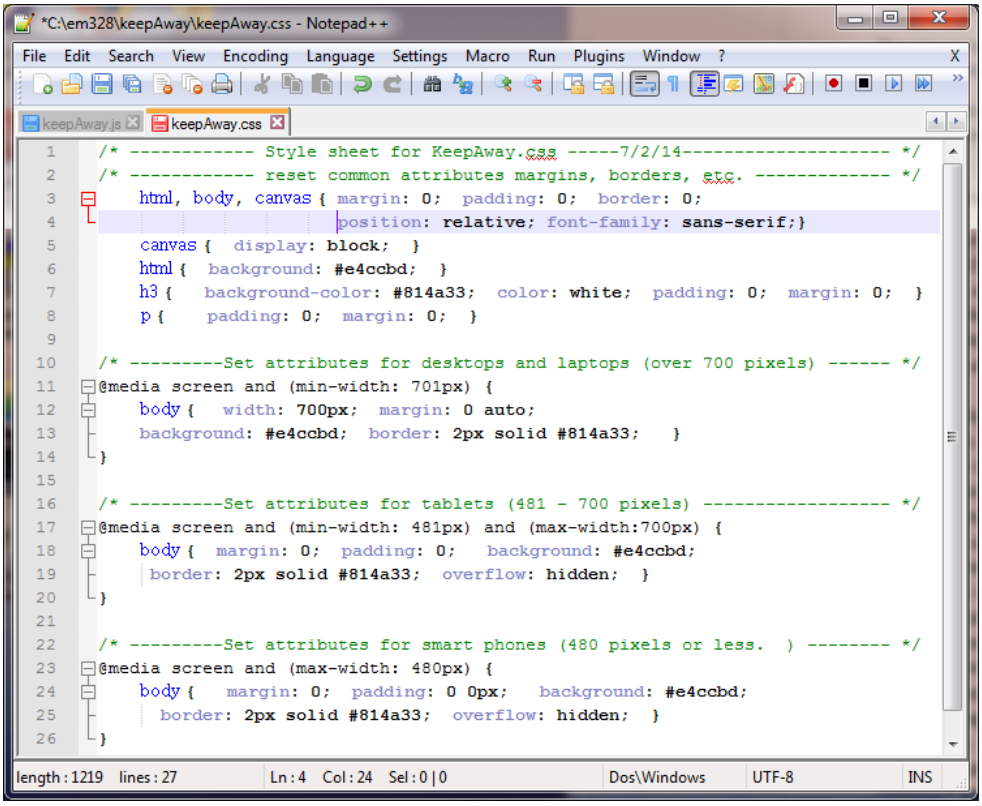
Notice the simplicity of the doctype and html tags. HTML5 is much less strict

on syntax than XHTML4 was. You can still use XHTML4 syntax, but it isn't necessary. The HTML5 specs put the burden back on the browsers, by documenting what the browser's behavior should be when attributes and punctuation marks are missing. There are a few things that are important, though, for responsive animation and interaction. The meta tags (lines 4-7) will help with screen and device detection; the link tag includes a media attribute that will help with screen detection and the canvas tags (lines 14-16) include a tabindex setting that will help to focus the keyboard events into the canvas object when the page loads. The statement between the canvas tags will only display if the browser version is too old to understand HTML5. The style sheet is linked in at line 9 and the JavaScript code is included at line 10.

The tags beneath the canvas (line 17) provide a place for displaying the screen width. This will be used for debugging only and can be removed for production.

If we run the page as is with no style sheet and no JavaScript code, the browser will display the page but it won't be pretty, and it won't resize itself to the existing screen size. That will happen in the CSS3 code and the JavaScript code.

Formatting Responsively by Detecting the Device and Window Size with CSS Media Queries



One of the great benefits of HTML5 allows developing a single set of source files that will detect the device and/or screen size and respond accordingly with appropriately sized components along with appropriate interactions with the user. Desktops and laptops will probably use bigger windows than the ideal size for the application, whereas tablets and smart phones will likely use a smaller than ideal window size. Most developers agree that detecting the screen/window size is more valuable and less error prone than detecting the device type. A CSS3 style sheet technique called media queries is a great tool for determining the size. Line 11, indicates that if this page is being displayed on a screen and it has a width of at least 701 pixels, then the styles contained between the curly braces should be applied. This is probably a desktop, laptop or tablet that is big enough screen for the ideal size and we can set the body width to the 700-pixel ideal size. Line 17 provides a section for styles that would apply to tablet-type screen sizes from 481 pixels to 700 pixels, and line 23 provides a section for most smart phone styles. Since our application is very simple, we will restrict the body width to 700 pixels if the screen is larger than that and we can use the "margin: 0 auto" setting to center the body in the window on wide screens. The tablet and smart phone styles will use a body that takes up the entire screen size.

Sizing the Canvas

All other layout and formatting operations must be accomplished by the JavaScript code in the init() method, which is called when the page is first loaded. (See the body tag in the HTML code on page 1.) The canvas must be scaled to fit both the width and height of the available window. The user should not have to scroll the canvas up and down to play the game. If the window is wider than 700 pixels and taller than 540 pixels, the canvas size can retain its ideal size of 700 x 500 pixels and the scale factor can be set to 1.0. For smaller widths, the code described below checks the window size of the browser and determines the scaling needed for the canvas. This same scale factor will also be used to reduce the image sizes proportionately.

keepAway.js

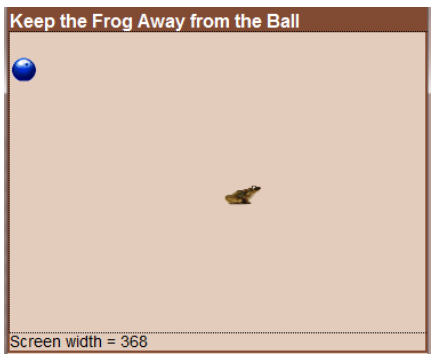
```
1  // keepAway.js - JavaScript Code for "Bowling for Frogs" Game  - 2/21/2015
2  var canvas, ctx, rect, offsetx, offsety;
3  var ball, frogSit, frogJump, frogLand, frogFlat, imageCount;
4  var ballx, bally, frogx, frogy, frogAngle, ballAngle;
5  var frogWidth, frogHeight, ballWidth, ballHeight, distance, scale;
6  var timer, millis, counter, bouncer, ballIdle, hopping, mouseisdown, gameOver;
7  var touchStarted, touchx, touchy;
8  var frogSitData, frogJumpData,  frogLandData, frogFlatData, frogBackgroundData;
9  var ballData, ballBackgroundData;
10
11 function init( ) {
12     // determine the canvas size
13     iwidth = window.innerWidth;
14     iheight = window.innerHeight;
15     widthSpan = document.getElementById('windowWidth');
16     widthSpan.innerHTML = "Screen width = " + iwidth;
17     if (iwidth >= 700 && iheight > 540 )
18     {
19         scale = 1.0;
20         canvasWidth = 700;
21         canvasHeight = 500;
22     }
23     else
24     {
25         scalex = iwidth / 700.0;
26         scaley = (iheight-40) / 500.0;
27         if ( scalex < scaley )
28            scale = scalex;
29         else
30            scale = scaley;
31         canvasWidth = (700) * scale;
32         canvasHeight = (500) * scale;
33     }
```

Lines 13-14 get the width and height available in the browser window. Lines 15-16 display the width in the span element just below the canvas (for debugging and testing only.)

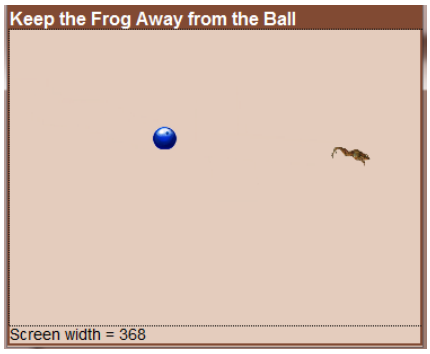
Lines 17-22 retain the ideal size for the canvas at 700 x 500 pixels and set the scale factor to 1.0 whenever the values exceed 700 pixels in height and 540 pixels in width.

Lines 23-33 determine the scale factor needed to keep the canvas width within the window (***scalex***), and the canvas height within the window (***scaley***). The height allows for 40 pixels needed for the information above and below the canvas.

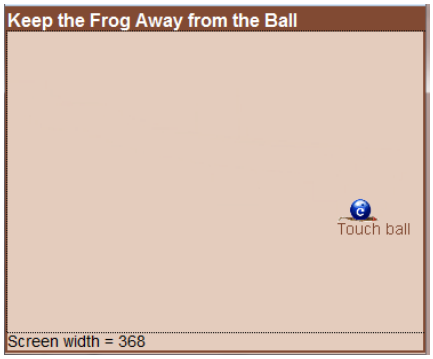
The smallest of these two scale factors must be used to ensure that the entire canvas will fit within the window and retain the same 700:500 proportions. The canvas width and height are each multiplied by the new ***scale*** factor to determine the new canvas size. The images and all other objects placed on the canvas will need to be reduced by this same scale factor to keep their proportions.



Start of Game



Frog Hopping – Ball Bouncing



End of Game – Frog Flattened

Scaling the Graphics

```
34      // get the canvas element and scale its width and height.
35      canvas = document.getElementById('canvas');
36      if ( canvas ) {
37          canvas.width = canvasWidth;
38          canvas.height = canvasHeight;
39          rect = canvas.getBoundingClientRect();
40          offsetx = rect.left;
41          offsety = rect.top;
42
43          // Get a two-dimensional graphics context for the canvas
44          ctx = canvas.getContext("2d");
45          if ( ctx ) {
46              // Start the frog in the center of the canvas
47              // and the ball in the top-left corner
48              distance = 40 * scale; // distance of a frog's hop
49              frogx = canvasWidth / 2;
50              frogy = canvasHeight / 2;
51              frogAngle = Math.PI * 7.0 / 4.0;
52              ballx = 2;
53              bally = 22;
54              ballAngle = Math.PI * 7.0 / 4.0;
55              xbdist = distance * Math.cos(ballAngle);
56              ybdist = distance * Math.sin(ballAngle);
57              bouncer = 1;  // bouncer counts the frames in a bounce
58              counter = 1;  // counmter coumts the frames in a hop
59              ballIdle = true;
60              hopping = false;
61              mouseisdown = false;
62              gameOver = false;
63              touchStarted = false;
64              millis = 200;   // milliseconds for timer interval.
```

To work with a graphic object, we need to set up a reference variable that points to the object's locations in memory. Line 35 uses the ***getElementById*** function to point the variable, ***canvas***, to the canvas element in the Web page document. Line 36 verifies that the canvas object exists, so we can modify its width and height in lines 37 and 38. Lines 39-41 gets the coordinates for the corner points of the area used by the canvas. The variables offsetx and offsety are set to the left and top margin values that surround the canvas.

Line 44 will set the variable ***ctx*** to point to the "two-dimensional" context for the canvas. Line 45 makes sure that a useable context object was retrieved. The ***ctx*** variable will be used for all future graphics manipulations on the canvas. The context object has many useful functions for drawing lines, shapes and images and for creating special effects like gradients and patterns. Variables needed for the game are initialized to their starting values in lines 44-62.

```
66              // Load the images and set events to count the number
67              // of images that have finished loading into memory.
68              imageCount = 0;
69              frogSit = new Image();
70              frogSit.addEventListener("load", imageLoaded, false);
71              frogSit.src = "frogsit.gif";
72              frogJump = new Image();
73              frogJump.addEventListener("load", imageLoaded, false);
74              frogJump.src = "frogjump.gif";
75              frogLand = new Image();
76              frogLand.addEventListener("load", imageLoaded, false);
77              frogLand.src = "frogland.gif";
78              frogFlat = new Image();
79              frogFlat.addEventListener("load", imageLoaded, false);
80              frogFlat.src = "frogflat.gif";
81              ball = new Image();
82              ball.addEventListener("load", imageLoaded, false);
83              ball.src = "ball.gif";
84              wait = 0;
```

Loading the Images

Reference variables are established for the images in lines 66-84. The three frog images will provide the animation that simulates a frog hopping as it moves from sitting (***frogSit***) to jumping (***frogJump***) to landing (***frogLand***) and back to sitting (***frogSit***). A single image (***ball***) will be used to animate the ball. The ***frogFlat*** image is displayed when the ball bounces and flattens the frog to end the game.

The canvas cannot be displayed until each of the images are loaded. The event listener is added to detect when an image is loaded. The imageLoaded handler (discussed later) will delay the display of the canvas until all images have finished loading.

Registering Events for User Interaction

```
86              // register event listeners for keyboard mouse and touch events
87              canvas.focus(); // make the canvas active for keyboard input
88              canvas.addEventListener('keydown', keyPushed, false);
89              canvas.addEventListener('mousemove', moveMouse, false);
90              canvas.addEventListener('mousedown', downMouse, false);
91              canvas.addEventListener('mouseup',   upMouse, false);
92              canvas.addEventListener("touchstart", startTouch, false);
93              canvas.addEventListener("touchend", endTouch, false);
94              canvas.addEventListener("touchleave", endTouch, false);
95              window.addEventListener('keydown', keyPushed, false);
96          }
97      }
98  } // end init()
```

To respond to any type of device, we must register keyboards events, mouse events and touch events. Line 87 is important to make sure that keyboard input is directed to the canvas object. By default, it's directed to the body of the Web page. Line 87 calls the focus method to make the canvas active for keyboard input. Lines 88-95 specify the function that should be called whenever that type of event occurs. For example, the keydown event will result in calling the keyPushed function whenever the user presses down on a key. Line 92 registers a listener for the entire browser window to resolve a keyboard problem that occurs in some browsers that change the active focus whenever the mouse pointer gets moved outside the canvas area. Notice that the same keyPushed event handler function is called for the window listener as for the canvas listener.

Responding to the Load Event for Images from Disk Files.

```
100 function imageLoaded( )
101 {   // Called when each image completes loading
102     imageCount++;
103     if (imageCount == 5)
104     {   // All five images are loaded, Set the image size to the scale
105         frogWidth = frogSit.width * scale;
106         frogHeight = frogSit.height * scale;
107         ballWidth = ball.width*scale;
108         ballHeight = ball.height*scale;
109
110         // Save the background where the images will be drawn.
111         frogBackgroundData = ctx.getImageData(frogx, frogy, frogWidth, frogHeight);
112         ballBackgroundData = ctx.getImageData(ballx, bally, ballWidth, ballHeight);
113
114         // Draw the images in scale and use getImageData to capture them in a
115         // more efficient format to avoid rescaling them each time they're drawn
116         ctx.drawImage(frogJump, frogx, frogy, frogWidth, frogHeight);
117         frogJumpData = ctx.getImageData(frogx, frogy, frogWidth, frogHeight);
118         ctx.putImageData(frogBackgroundData, frogx, frogy);
119         ctx.drawImage(frogLand, frogx, frogy, frogWidth, frogHeight);
120         frogLandData = ctx.getImageData(frogx, frogy, frogWidth, frogHeight);
121         ctx.putImageData(frogBackgroundData, frogx, frogy);
122         ctx.drawImage(frogFlat, frogx, frogy, frogWidth, frogHeight);
123         frogFlatData = ctx.getImageData(frogx, frogy, frogWidth, frogHeight);
124         ctx.putImageData(frogBackgroundData, frogx, frogy);
125         ctx.drawImage(frogSit, frogx, frogy, frogWidth, frogHeight);
126         frogSitData = ctx.getImageData(frogx, frogy, frogWidth, frogHeight);
127         ctx.drawImage(ball, ballx, bally, ballWidth, ballHeight);
128         ballData = ctx.getImageData(ballx, bally, ballWidth, ballHeight);
129     }
130 }
```

The ***imageLoaded*** function is an event handler that is called each time an image is finished loading from disk to memory. Since the loading process happens asynchronously and concurrently, the imageLoaded function must wait and count until all five images have been loaded. This occurs at lines 100-103. Lines 104-108 are executed after all five images are loaded and will determine the scaled width and height of the frog images and ball images. Lines 110 - 112 save the original

background area where the ball and frog will be placed. These background areas will be used to replace the original background and effectively erase the images from the screen.

The image files are compressed transparent gif files which can most easily be displayed at the desired scale using the ***drawImage*** method. The performance of later animations can be improved if we immediately capture the scaled-down image using the ***getImageData*** method. The ***getImageData*** method captures the bytes back from the canvas and returns the image as data which does not have to be decompressed and resized. With the ***ImageData*** format, the ***putImageData*** method can redraw the image on the screen much quicker than the ***drawImage*** method can. Lines 114-126 draw each frog image onto the screen and then get the decompressed and resized image back.

The ***getImageData(top, left, width, height)*** function retrieves the pixel data from a specific rectangular area on the canvas. The ***putImageData(top, left)*** function places the image data at the coordinates specified by ***top*** and ***left***. The ***putImageData*** function is much faster than the ***drawImage*** function because it doesn't have to do any calculations. It just copies the bytes directly to memory, because they're already resized and in the order they need to be to overwrite the existing bytes on the canvas. After line 128 is executed, the ball and the sitting frog are in position to start the game.

Handling User Interactions

User-controlled animations need to be responsive to any age of user as well as any size of device. The developer must be concerned with user issues like the fat fingers, slow fingers, jerky fingers, mouse antics and strange touch behaviors. It seems that users of all ages can adapt quite easily to anything that works well for normal pre-school children. If a young child is using the keyboard, they're probably going to keep a key down for a long time if they're trying to move an object. They'll usually use a mouse in the same way by clicking on an object and keeping the left button down as they try to get the object to move around. These user behaviors can cause type-ahead problems if they're using the keyboard and even unwanted mouse event problems if they're using the mouse or touch screens and pads. Our game is going to work the best if we respond to the user as quickly as possible and then ignore user events until our response (frog movement in our case) is finished.

Some behavior choices may have to wait until we've done some testing. For example, the ball will keep bouncing toward the frog at all times. It doesn't matter what the user does. But, what should the frog do? Should it keep hopping in the current direction, even if the user isn't clicking, dragging, swiping or pressing keys, or should it stop when ever the user stops generating events? Also, the ball can bounce off the edges of the canvas. Should the frog bounce off or stop when it gets to an edge? We'll have to answer these questions through experimentation. To start with, we'll have the frog hop continuously in the current direction but stop when it hits the wall. The user can start it again with a key press, mouse drag or finger swipe in an appropriate direction.

The Keyboard Event handler

```
132  function keyPushed( e )
133  {
134      // The following lines get the code value of the key
135      // Which may be recorded differently en each browser.
136      var evtobj=window.event? event : e //IE's window.event or Firefox'
137      var code=evtobj.charCode? mmevtobj.charCode : evtobj.keyCode
138      if (evtobj.preventDefault)
139          evtobj.preventDefault();
140      evtobj.returnValue = false;
141
142      // set the direction to hop. (keys turn in right angles.)
143      switch (code) {
144          case 37: // left arrow
145              frogAngle = Math.PI;
146              break;
147          case 38: // up arrow
148              frogAngle = Math.PI / 2.0;
149              break;
150          case 39: // right arrow
151              frogAngle = 0;
152              break;
153          case 40: // down arrow
154              frogAngle = Math.PI * 3.0 / 2.0;
155              break;
156          case 27: // Escape key, restart.
157              clearInterval( ballTimer );
158              ballTimer = 0;
159              init( );
160              return;
161              break;
162      }
163      hopping = true; // start the frog on any key
164      // If this is the first key down, start the ball and timer
165      if ( ballIdle )
166      {
167          ballIdle = false;
168          ballTimer = setInterval(hopAndBounce, millis);
169      }
170  }
```

There are three keyboard events, ***keydown***, ***keyup*** and ***keypressed***. The ***keydown*** event fires when the user pushes a key down. The ***keyup*** event fires when the user releases the key. Both of these events set the ***keycode*** variable to the key's numeric value. This isn't the ASCII value of the key because not all the keys generate ASCII values. The ***keyPressed*** event is called after ***keyup*** if the key generates an ASCII value. We're primarily interested in the arrow keys and they don't generate an ASCII value, so we'll use the ***keydown*** event. The event handler that was assigned starts at line 132 in the code above. Lines 136-140 are used to resolve the differences between the event object created and passed by Internet Explorer and the event object created and passed by Firefox. Lines 142-161 determine the key that was pressed and sets the angle that the frog should travel on its next jump. The ***hopping*** variable indicates whether the frog should be jumping or not. It's set to false when the game starts, but is set to true at line 163 when the user presses a key. The ***ballIdle*** variable is set to true when the game starts to keep the ball from bouncing. The first time the user generates an event, lines 164-169 will set ***ballIdle*** to false and will set a timer to start the action. The function ***hopAndBounce()*** will be called by the timer at intervals defined by ***millis***, which contains the number of milliseconds between each timer event.

The event handler also has to be sensitive to the way little kids may use the keyboard. It seems to work best for this game to have the frog start hopping when an arrow key is pressed in the direction of the arrow. The frog will continue hopping in the same direction until a different arrow key is pushed down. The first ***keydown*** event will also start the ball bouncing. Before each bounce, the ball's direction will be set toward the current location of the frog.

The Mouse Event Handlers

```
172  function downMouse( evt )
173  {   // Called when the mouse button is pressed down.
174      mouseisdown = true; // indicate the start of a click or drag
175      moveMouse( evt );
176  }
177
178  function upMouse( evt )
179  { // Called when the mouse button is released.
180      mouseisdown = false;
181  }
182
183  function moveMouse( evt )
184  { // Called when the mouse is moved across the canvas.
185
186      if (!evt)
187          var evt = event;
188      if (mouseisdown) // only act on dragging movements.
189      {
190          clickx = evt.clientX - offsetx;  // set a new frogAngle
191          clicky = evt.clientY - offsety;  // toward the click
192          frogAngle = Math.atan2((frogy-frogHeight/2.0-clicky),
193                                 (clickx-frogx+frogWidth/2.0));
194          if ( gameOver ) // restart game if its over and user clicks on ball
195          {
196              if (( clickx >= frogx ) && ( clickx <= frogx + frogWidth )
197               && ( clicky >= frogy ) && ( clicky <= frogy + frogHeight ))
198                  init();
199          }
200          else
201          {
202              hopping = true; // start the frog on any event
203              if (ballIdle)   // start the ball if its the first event
204              {
205                  ballIdle = false;
206                  // ballTimer calls hopAndBounce every "millis" milliseconds
207                  ballTimer = setInterval(hopAndBounce, millis);
208              }
209          }
210      }
211  }
```

Users can click to get the frog to hop toward the click or they can drag the mouse around to get the frog to hop toward the mouse pointer as it is being dragged.

The ***mouseDown*** event calls function ***downMouse*** at lines 172-176. It sets the ***mouseisdown*** variable to true and then calls the ***moveMouse*** function to perform the action needed.

The ***mouseUp*** event calls function ***upMouse*** at lines 178-181, which sets the ***mouseisdown*** variable back to false.

The ***mouseMove*** event calls function ***moveMouse*** where the event handling is actually performed. Lines 186-187 gets the event object if the browser didn't pass the event as a parameter. Line 188 makes sure that the mouse button is currently down to make sure that the user is either clicking or dragging. Lines 190-193 use the current mouse coordinates to set the angle that the frog will jump in the future. The angle is computed to change the jumping direction toward the click. Lines 194-199 check to see if the event occurred after the game is over. If it is over, and the click is on the area occupied by the ball and the flattened frog, the ***init()*** function is called to start a new game. Lines 200-209 will start the game action if it hasn't started yet (***ballIdle is true***). The game action is started by creating a timer object, which will call the ***hopAndBounce*** function at regular intervals indicated by the number of milliseconds contained by the variable, ***millis***.

Animating the Frog and the Ball in the *hopAndBounce* Function

```
213  function hopAndBounce()
214  { // Called every time the ballTimer event occurs.
215      // hop the frog
216      if (counter == 1 && hopping)
217      {
218          ydist = distance * Math.sin(frogAngle);
219          xdist = distance * Math.cos(frogAngle);
220          // Don't hop if it would end up outside the canvas.
221          newx = frogx + xdist;
222          newy = frogy - ydist;
223          if (( newx < 0 ) || ( newx > canvasWidth - frogWidth )
224          || ( newy < 0 ) || ( newy > canvasHeight - frogHeight ))
225          {
226              hopping = false;
227              counter=0;
228          }
229          else
230              hopping = true;
231          if (hopping)
232          {
233              ctx.putImageData(frogBackgroundData, frogx, frogy);
234              frogx += xdist / 3.0;
235              frogy -= ydist / 3.0;
236              frogBackgroundData = ctx.getImageData(frogx, frogy, frogWidth, frogHeight);
237              ctx.putImageData(frogJumpData, frogx, frogy);
238          }
239      }
240      else if (counter == 2 && hopping) // erase this spot, land in new spot
241      {
242          ctx.putImageData(frogBackgroundData, frogx, frogy);
243          frogx += xdist / 3.0;
244          frogy -= ydist / 3.0;
245          frogBackgroundData = ctx.getImageData(frogx, frogy, frogWidth, frogHeight);
246          ctx.putImageData(frogLandData, frogx, frogy);
247      }
248      else if (counter == 3 && hopping) // erase this spot, sit in new spot
249      {
250          ctx.putImageData(frogBackgroundData, frogx, frogy);
251          frogx += xdist / 3.0;
252          frogy -= ydist / 3.0;
253          frogBackgroundData = ctx.getImageData(frogx, frogy, frogWidth, frogHeight);
254          ctx.putImageData(frogSitData, frogx, frogy);
255          counter = 0;
256      }
257      else
258          counter = 0;
259      counter++;
```

The ***hopAndBounce*** function is the game engine for the application. It's called at regular intervals by the ***ballTimer*** *object to keep the ball bouncing and the frog hopping.* Line 216 checks to see if the frog needs to make the first of three steps in the hop. Lines 218-222 compute the distance that the move will make in each of the X and Y directions. Lines 223-228 will stop the frog from hopping if any part of the frog is ***outside*** the canvas area. Lines 231-238 erase the current area occupied by the frog and draw the ***frogJump*** image in the new position.

If this is the second part of the jump, Lines 240-247 erase the current frog image and draw the ***frogLand*** image in its new position.

If this is the third part of the jump, Lines 248-256 erase the current frog image and draw the ***frogSit*** image in its new location.

Lines 257-258 reset the jump counter to zero to start a new jump. Line 259 increments the counter for the next part of the jump, when the ***hopAndBounce*** function is called again.

```
261     // bounce the ball, but erase it first in current spot
262     ctx.putImageData(ballBackgroundData, ballx, bally);
263
264     ballx += xbdist / 3.0;
265     if (bouncer == 1) // draw in high point of bounce
266     {
267         if (ybdist >= 0)
268             bally += ybdist;
269         else
270             bally -= ybdist * 2;
271         ballBackgroundData = ctx.getImageData(ballx, bally, ballWidth, ballHeight);
272         ctx.putImageData(ballData, ballx, bally);
273     }
274     else if (bouncer == 2) // draw in middle point of descent
275     {
276         if (ybdist >= 0)
277             bally -= ybdist;
278         else
279             bally += ybdist / 2.0;
280         ballBackgroundData = ctx.getImageData(ballx, bally, ballWidth, ballHeight);
281         ctx.putImageData(ballData, ballx, bally);
282     }
283     else if (bouncer = 3) // draw in final point of the bounce
284     {
285         if (ybdist >= 0)
286             bally -= ybdist;
287         else
288             bally -= ybdist / 2.0;
289         ballBackgroundData = ctx.getImageData(ballx, bally, ballWidth, ballHeight);
290         ctx.putImageData(ballData, ballx, bally);
291         ballAngle = Math.atan2((bally-frogy),(frogx-ballx));
292         xbdist = distance * 0.5 * Math.cos(ballAngle);  // set the distance for
293         ybdist = distance * 0.5 * Math.sin(ballAngle);  // the next bounce.
294         bouncer = 0;
295     }
296
```

The bouncing ball is also animated by displaying it in three different positions. Lines 261-262 erase the ball in its current position. The variable ***bouncer*** determines which position will be displayed at each timer event. When ***bouncer*** is 1, the ball is displayed at its high point in lines 265-273.

Lines 274-282 draw the ball in the middle of its descent and lines 283-290 draw it in its final point of the bounce.

Lines 291-294 reset the angle that the ball will move during its next bounce. A little trigonometry helps to insure that the ball will always bounce toward the frog's current position, keeping the game in high suspense as the plot thickens. (This is as close to humor as the author can get.)

```
297     // Check to see if the ball hit the frog
298     if ( ballx + ballWidth > frogx && ballx < frogx + frogWidth
299       && bally + ballHeight > frogy && bally < frogy + frogHeight )
300     {
301         // stop the timer and display a squished frog
302         clearInterval(ballTimer);
303         ballTimer = 0;
304         gameOver = true;
305         ctx.putImageData(ballBackgroundData, ballx, bally);
306         ctx.putImageData(frogBackgroundData, frogx, frogy);
307         ctx.putImageData(frogFlatData, frogx, frogy);
308
309         // Touching or clicking on the ball will restart the game.
310         ctx.fillStyle = "#814a33";
311         ctx.font = "14px Arial";
312         ctx.fillText("Touch ball", frogx, frogy + frogHeight + 10);
313     }
314     bouncer++;
315 }
```

Lines 298-299 check to see if the ball hit the frog during this portion of the bounce. Each image is stored as a rectangle of pixels, so the code estimates that the ball hits the frog whenever the rectangle of the ball image intersects the rectangle of the frog image. When the ball hits the frog, Lines 301-304 stop the timer and end the game. Lines 305-307 erase the current frog and ball images display the ***frogFlat*** image, which shows the ball sitting on a flattened frog. The message displayed at line 312 helps to show the user that the game can be restarted by touching or clicking on the ball. The game can also be restarted by pressing the escape key.

Handling the Touch Events

```
317  function startTouch(evt)
318  {
319      if (!evt)
320          var evt = event;
321      evt.preventDefault();
322      var touches = evt.changedTouches;
323      if (touches.length == 1)
324      {
325          touchx = touches[0].pageX - offsetx;
326          touchy = touches[0].pageY - offsety;
327          touchStarted = true;
328      }
329  }
330
331  function endTouch(evt)
332  {
333      if (!evt)
334          var evt = event;
335      evt.preventDefault();
336      var touches = evt.changedTouches;
337      if (touches.length == 1 && touchStarted)
338      {
339          newx = touches[0].pageX - offsetx;
340          newy = touches[0].pageY - offsety;
341          touchStarted = false;
342      }
343      frogAngle = Math.atan2((touchy - newy),
344                             (newx - touchx));
345
346      if ( gameOver )// restart game if its over and user touches ball
347      {
348          if (( newx >= frogx ) && newx <= (frogx + frogWidth )
349           && ( newy >= frogy ) && newy <= (frogy + frogHeight ))
350              init();
351      }
352      else
353      {
354          hopping = true; // start the frog on any event
355          if (ballIdle)   // start the ball if its the first event
356          {
357              ballIdle = false;
358              // ballTimer calls hopAndBounce every "millis" milliseconds
359              ballTimer = setInterval(hopAndBounce, millis);
360          }
361      }
362  }
```

Touch events on tablets and smart phones are different from mouse events. Lines 317-329 handle the start of a touch event. If the touch involves a single finger, the coordinates of the touch are saved and the ***touchStarted*** variable is set to true. The actual response will be handled at the end of the touch event.

The ***endTouch*** function is called at the end of a swipe or the end of a touch. Lines 333-344 use the event object to obtain the new coordinates of the touch and compute the new angle for the frog's next jump. The angle will be the direction of the swipe.

Lines 346-351 allow the user to restart the game if it is over. If the touch is inside the flattened frog image, the ***init()*** method is called to initialize all variables and set the canvas back to its starting state.

Lines 352-361 will be executed when the game isn't over yet. Line 354 makes sure that the frog is in a hopping state after any user event and lines 355-360 will start the ball bouncing if this is the first user event in the game.

Notes on Animation

The animation technique used in this example is not a scalable solution.

An alternative technique (and the one used most often) rewrites the entire canvas on each timer event, using the new locations for each of the images involved. If this alternative is too slow or causes the screen to flicker, double-buffering should be used. With double-buffering the new scene is built on a second, off-screen canvas first and that canvas is then switched to become the viewable canvas. For details see the article from Ken's Blog at http://blog.bob.sh/2012/12/double-buffering-with-html-5-canvas.html

What's Next?

Computer games are never finished. They're based on a simulation of the real world that's never perfect. This game is based on such a simple plot, that we've probably beat it to death already, but we can see several things that could be modified to make it better, such as:

- Test it on more browsers, more devices and with more users.
- Rotate the frog, so it will always face the direction it's hopping.
- Improve the ball bouncing, so that it always bounces the same height.
- Add some levels of difficulty by putting obstacles in the court, such as walls and barricades.
- Add sound.
- Keep score and records.
- Compile the code to keep it from being copied and pirated.

Speaking of copies, Feel free to use this code to experiment with and to incorporate the techniques in your own apps. You can download the files from http://www.learnmorecode.com/ra/files.zip. The files include all HTML, CSS, JavaScript and images files for the app plus this tutorial paper and the PowerPoint slides. Please do not post the app, the tutorial or the PowerPoint slides on other Web sites. You may try the app at http://www.learnmorecode.com/ra/keepAway.html Questions may be directed to rrobinson@sf.coloradotech.edu or robrobinson47@gmail.com.