# 6

# RENDERING CANVAS SPRITES

Up until now, we've built *Bubble Shooter* with a DOM-based approach by using HTML elements for game objects that are styled and positioned by CSS and manipulated by JavaScript. In this chapter, we'll rework *Bubble Shooter* so most of the game area is rendered to a canvas instead of using the DOM. Our game's dialogs will remain in HTML and CSS.

Canvas rendering allows us to achieve graphical effects that are often impossible with DOM-based development, and it can often provide a faster rendering speed. To use canvas rendering for *Bubble Shooter*, we need to learn how to render entire scenes to the canvas, maintain state, and perform frame-by-frame animations.

Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

We'll keep the existing DOM-rendering code in place for devices where the canvas element isn't supported and provide progressive enhancement to the canvas for more modern browsers. We'll do this to demonstrate the principle involved in coding for both canvas- and DOM-based animation and to highlight the differences between the two approaches.

## Detecting Canvas Support

Modernizr can help us detect canvas features so we don't have to remember multiple cross-browser cases. We'll load in only a couple of extra JavaScript files for the canvas version and won't delete any files. To detect the canvas and load in the right files, we need an extra node in `Modernizr.load` in *index.html*, which will check for canvas support, and if present, load JavaScript files from an array. Add the following before *game.js* is loaded:

*index.html*

```
  },
  {
    test: Modernizr.canvas,
    yep: ["_js/renderer.js","_js/sprite.js"]
  },
  {
    load: "_js/game.js",
    complete: function(){
      $(function(){
        var game = new BubbleShoot.Game();
        game.init();
      })
    }
  }]);
```

The value of `Modernizr.canvas`, the parameter that `test` looks for, will be either `true` or `false`. If it's `true`, the two files listed in `yep` are loaded; if it's `false`, nothing new happens.

Create empty files for *renderer.js* and *sprite.js* in the *_js* folder. The `Renderer` object will draw the game state at each frame, and the `Sprite` class will perform many of the operations that we've been using jQuery for to date. We want `Renderer` to be responsible for drawing pixels onto the canvas and not mix up game logic with it; likewise, we'll try to keep state information inside the relevant objects. This approach makes it much easier to switch between rendering using the canvas or the DOM, depending on what we think is best for the game.

## Drawing to the Canvas

With HTML5's canvas feature, you can build games at a level of sophistication similar to that of Flash games or even native applications. You place canvas elements into documents in the same way as other elements, such as `<div>` or `<img>`, but it's the way you work with the element that makes it



Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

different. Inside the canvas, you have pixel-level control, and you can draw to individual pixels, read their values, and manipulate them. You can write JavaScript code to generate arcade shooters or even 3D games that are difficult to reproduce with a DOM-based approach.

### The DOM vs. the Canvas

HTML is primarily an information format; CSS was introduced as a way to format that information. Creating games using both technologies is really a misappropriation, and games like *Bubble Shooter* are feasible largely because browser vendors have made an effort to increase performance. Many of the processes that are invaluable in laying out documents, such as ensuring that text areas don't overlap or that text wraps around images, are practices that we don't need for laying out games. As game developers, we take on responsibility for ensuring the screen is laid out well, but, unfortunately for us, the browser still runs through all of these checks in the background.

For example, adding or removing elements in the DOM can be a relatively expensive operation in terms of processing power. The reason is that if we add or remove something, the browser needs to inspect it to ensure that the change doesn't have a domino effect on the rest of the document flow. If we were working with, say, an expanding menu on a website, we might want the browser to push a navigation area down if we add more elements to it. However, in a game it's more likely that we will be using `position: absolute`, and we definitely don't want the addition or removal of a new element to force everything surrounding it to be repositioned.

By contrast, when the browser sees a `canvas` element, it sees just an image. If we change the contents of the canvas, only the contents change. The browser doesn't need to consider whether this change will have a knock-on effect on the rest of the document.

Unlike CSS and HTML, the canvas doesn't let you rely on the browser to keep track of the positions of objects on the screen. Nothing automatically deals with layering or rendering backgrounds when a sprite moves over them because the canvas outputs a flat image for the browser to display. If sprite animation and movement with CSS is like moving papers around on a notice wall, canvas animation is more like working with a whiteboard: if you want to change something or move it, you'll have to erase an area and redraw it.

Canvas rendering also differs from CSS layout in that positioning of elements can't be offloaded to the browser. For example, with our existing DOM-based system, we can use a CSS transition to move the bubble visually from its firing position to wherever we want it to end up in the board layout. To do this takes only a couple of lines of code.

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

Canvas rendering, on the other hand, requires us to animate frame by frame in a way similar to the internal workings of jQuery. We must calculate how far a bubble is along its path and draw it at that position each time a frame update occurs.

On its own, animating on the canvas using JavaScript would be no more arduous than JavaScript animation using the DOM without jQuery or CSS transitions to fall back on, but the process is made more complex by the fact that if we want to change the contents of the canvas, we need to delete pixels and redraw them. Ways to optimize the redrawing process are available, but a basic approach is to draw the entire canvas afresh for each animation frame. This means that, if we want to move an object across the canvas, we have to render not just the object that we want to move but possibly every object in the scene.

We'll draw the game board and the current bubble using the canvas, but some components, such as dialogs, are better left as DOM elements. User interface components are generally easier to update as DOM elements, and the browser usually renders text more precisely with HTML than it would render text within a `canvas` element.

Now that we've decided to render the game with a canvas system, let's look at what that will involve. The key tasks are rendering the images and maintaining states for each bubble so that we know which bubbles are stationary, which are moving, and which are in the various stages of being popped.

## Image Rendering

Any image you want to draw to the canvas must be preloaded so it's available to be drawn; otherwise, nothing appears. To do this, we'll create an in-memory `Image` object in JavaScript, set the image source to the sprite sheet, and attach an `onload` event handler to it so we know when it's finished loading. Currently, the game is playable once the `init` function in *game.js* has run and the New Game button has the `startGame` function attached to its `click` event:

```
$(".but_start_game").bind("click",startGame);
```

We still want this to happen, but we don't want it to happen until after the sprite sheet image has loaded. This will be the first task we'll tackle.

### *canvas Elements*

Next, we need to know how to draw images onto the canvas. A `canvas` element is an HTML element just like any other: it can be inserted into the DOM, can have CSS styling applied, and behaves in much the same way as an image. For example, to create a `canvas` element, we add the following to *index.html*:

```
<canvas id="game_canvas " width="1000" height="620"></canvas>
```


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

This creates a canvas element with the dimensions of 1000 pixels wide by 620 pixels high. These dimensions are important because they establish the number of pixels that make up the canvas. However, we should also set these dimensions in CSS to establish the size of the canvas as it will appear on the page:

```
#game_canvas
{
  width: 1000px;
  height: 620px;
}
```

In the same way that an image can be rendered at scale, the canvas element can also be scaled. By setting the CSS dimensions to the same values as the HTML attributes, we ensure that we're drawing the canvas at a scale of 1:1. If we omitted the CSS, the canvas would be rendered at the width and height specified in the attributes, but it's good practice to specify layout dimensions within the style sheet. Not only does it help with code readability, but it also ensures that if the internal dimensions of the canvas are changed, the page layout won't break.

To draw an image onto the canvas using JavaScript, we first need to get a *context*, the object that you use to manipulate canvas contents, using the method getContext. A context tells the browser whether we're working in two dimensions or three. You would write something like this to indicate you want to work in two-dimensional space rather than three-dimensional space:

```
document.getElementById("game_canvas").getContext("2d");
```

Or to write this using jQuery:

```
$("#game_canvas").get(0).getContext("2d");
```

Note that the context is a property of the DOM node, not the jQuery object, because we're retrieving the first object in jQuery's set with the get(0) call. We need the DOM node because the basic jQuery library doesn't contain any special functions for working with canvas elements.

Now, to draw the image onto the canvas, we use the drawImage method of the context object:

```
document.getElementById("game_canvas").getContext("2d").
drawImage(imageObject,x,y);
```

Or again, to write this using jQuery:

```
$("#game_canvas").get(0).getContext("2d").drawImage(imageObject,x,y);
```

Build an HTML5 Game
© 2015 Karl Bunyan

www.itbook.store/books/9781593275754

The parameters passed into `drawImage` are the `Image` object and then *x*- and *y*-coordinates at which to draw the image. These are pixels relative to the canvas context origin. By default, (0,0) is the top-left corner of the canvas.

We can also clear pixels from the canvas with the `clearRect` method:

```
$("#game_canvas").get(0).getContext("2d").clearRect(0, 0, 1000, 620);
```

The `clearRect` command removes all canvas pixels from the top-left corner (first two parameters) down to the bottom-right corner (last two parameters). Although you can just clear the canvas rectangle that you want to change, it's usually easier to clear the entire canvas and redraw it each frame. Again, the coordinates are relative to the context origin.

The context maintains a number of state properties about the canvas, such as the current line thickness, line colors, and font properties. Most important for drawing sprites, it also maintains the coordinates of the context origin and a rotation angle. In fact, you can draw an image at a set position on the canvas in two ways:

- Pass *x*- and *y*-coordinates into the `drawImage` function.
- Move the context origin and draw the image at the origin.

In practice, you'll see the same results with either method, but there is a reason it's often best to move—or *translate*—the context origin. If you want to draw an image onto the canvas at an angle, it's not the image that's rotated but the canvas context that's rotated prior to drawing the image.

### Rotating the Canvas

The canvas is always rotated around its origin. If you want to rotate an image around its own center, first translate the canvas origin to a new origin at the center of the image. Then rotate the canvas by the angle at which you want to rotate the image *but in the opposite direction to the rotation you wanted to apply to the object.* Then draw the image as usual, rotate the canvas back to zero degrees around its new origin, and finally translate the canvas back to its initial origin. Figure 6-1 shows how this works.

For example, to draw an image that's 100 pixels across at coordinates (100,100) and rotate it by 30 degrees around its center, you could write the following:

```
❶ var canvas = $("#game_canvas").get(0);
❷ var context = canvas.getContext("2d");
❸ context.clearRect(0, 0, canvas.width, canvas.height);
❹ context.translate(150, 150);
❺ context.rotate(Math.PI/6);
❻ context.drawImage(imageObject, -50, -50);
❼ context.rotate(-Math.PI/6);
❽ context.translate(-150, -150);
```


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

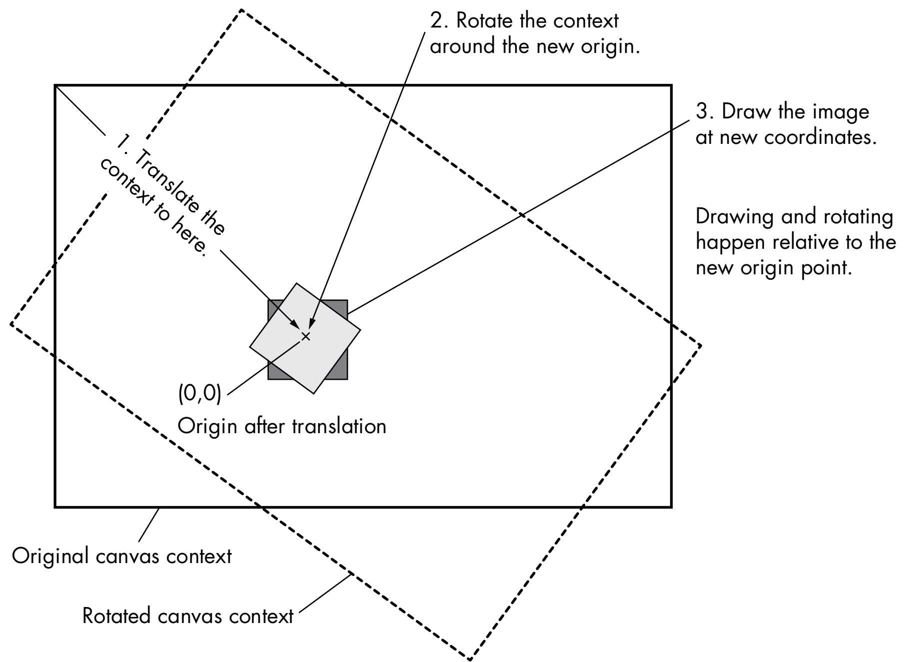


*Figure 6-1: Drawing a rotated image onto the canvas*

This code retrieves the canvas ❶ and the context ❷ and then clears the canvas so it's ready for drawing ❸. We next translate the origin to the coordinates at which we want to draw the image ❹, but we also need to add half the image's width and half of its height to the translation values, because we'll be drawing the center of the image at the new origin.

The next step is to add rotation ❺, but remember that we rotate the *context*, not the image. Angles are also specified in radians rather than degrees. The image is drawn at (-50,-50) ❻, which means that the center of the image is drawn at the context origin and then the context is rotated back ❼ and then translated back ❽. The last two steps are important because the context maintains state, so the next operation that's performed on the canvas would be on the rotated coordinates. By reversing the rotation and the translation, we have left the canvas in the same state in which we found it.

If you don't want to have to remember to rotate and translate the canvas back to its origin, you can simplify the whole process by storing the context before changing your image and resetting the context afterward:

```
  var canvas = $("#game_canvas").get(0);
  var context = canvas.getContext("2d");
  context.clearRect(0, 0, canvas.width, canvas.height);
❶ context.save();
  context.translate(150, 150);
  context.rotate(Math.PI/6);
  context.drawImage(imageObject, -50, -50);
❷ context.restore();
```

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

The call to `context.save` ❶ saves the current state of the context, although, importantly, it doesn't save the pixel data inside the canvas. Then `context.restore` ❷ sets it back to this saved state.

These principles are all we need to draw whole images onto the canvas and to remove them again, but to draw bubbles, we'll need to draw only a small section of the sprite sheet at a time.

### CANVAS WIDTH AND HEIGHT

The canvas has its own settings for width and height, and it's important to specify these when you create a `canvas` element. You could use CSS to determine the dimensions of the canvas as displayed on the screen, but they may not match the number of pixels that the canvas internally is set to render. In our case, we'll make both the same, so drawing one pixel to the canvas will result in one pixel being displayed.

If we were to set the width and height of the `canvas` element to double what they are now, the DOM element would still take up the same amount of space on the page because of our CSS definition. The canvas interacts with CSS in the same way images do: the width and height are specified in the style sheet, but the canvas (or image) may be larger or smaller. The result is that the image we draw occupies only the top quarter of the canvas and appears to be a quarter of its original size. This happens because canvas pixels are scaled to screen pixels at render time. Try changing the `canvas` definition in *index.html* to the following and see what happens:

```
<canvas id="game_canvas" width="2000" height="1240"></canvas>
```

The `canvas` element won't appear any bigger on the screen because of the CSS rules. Instead, every pixel defined by CSS will be represented by 4 pixels on the canvas. In most desktop browsers, 1 CSS pixel is identical to 1 screen pixel, so there's little benefit to setting the canvas dimensions to values larger than those in the CSS. However, modern devices, especially mobile ones, have become sophisticated in their rendering and have what is called a higher pixel density. This allows the device to render much-higher-resolution images. You can read more about pixel density at *http://www.html5rocks.com/en/tutorials/canvas/hidpi/*.

When you're working with the canvas and CSS together, you need to remember which scale you're working at. If you're working within the canvas, it's the dimensions of the canvas, as specified by its HTML attributes, that are important. When working with CSS elements around—or possibly even on top of—the canvas, you'll be using CSS pixel dimensions. For example, to draw an image at the bottom-right of a canvas that is 2000 pixels wide and 1240 pixels high, you would use something like this:

```
$("#game_canvas").get(0).getContext("2d").drawImage(imageObject,2000,1240);
```


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

But to place a DOM element at the bottom-right corner, you would use the coordinates (1000,620), such as in the following CSS:

```
{
  left: 1000px;
  top: 620px;
}
```

If possible, it's generally easiest to keep your screen display canvas size (set in the CSS) and the width and height definitions for the canvas the same so the canvas renderer doesn't have to try to scale pixels. But if you're targeting devices with high pixel densities (such as Apple Retina displays), you can improve the quality of your graphics by experimenting with increasing the number of pixels in the canvas.

## Sprite Rendering

We can't use background images and position offsets to render bubble sprites, as we did with our DOM-based system. Instead, we need to draw the bubble sprites as images onto the canvas. Remember that the sprite image file contains all four bubble colors in both resting and popping states. For example, in the sprite image shown in Figure 6-2, if we want to draw a blue bubble onto the board, we are interested in only the section of the image surrounded by the dotted line. To select only this part of the image, we'll use the clip parameters that can be passed into the `drawImage` method of a canvas context.

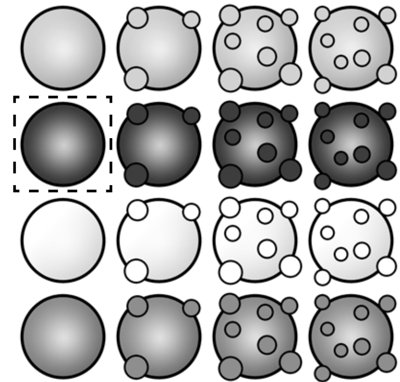

*Figure 6-2: Clip boundary required to draw a blue bubble onto the board*

If we want to draw the bubble in the first stage of being popped, we would move the clip area to the right. This is similar to the way we display bubbles in the DOM version except that, rather than letting the boundaries of a `div` element define the clip boundaries, we'll specify them in JavaScript.

To draw a clipped image to the canvas, add a couple more parameters to the `drawImage` method. Previously, we used `drawImage` with only three parameters (the `Image` object and *x*- and *y*-coordinates), but we can pass it a few more to clip the image. The full set of parameters that `drawImage` accepts are these:

```
context.drawImage(img,sx,sy,swidth,sheight,x,y,width,height);
```

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

The parameters are as follows:

**img** The Image object.

**sx and sy** The *x*- and *y*-coordinates at which to clip the image relative to the image's origin. For a blue bubble in its nonpopping state, these values would be 0 and 50, respectively.

**swidth and sheight** The width and height of the clip area. For our bubble sprite sheet, these values will both be 50.

**x and y** The coordinates to draw the image on the canvas relative to the canvas context origin.

**width and height** The width and height of the image to draw. We can use these parameters to scale an image, or we can omit them if we want the image to be drawn at 1:1.

For example, to draw the blue bubble highlighted in Figure 6-2 at the coordinates (200,150) on the canvas, we would use the following:

```
$("#canvas").get(0).getContext("2d").drawImage(spriteSheet,0,50,50,50,200,150,
50,50);
```

This line of code assumes the sprite `Image` object is named `spriteSheet` and the sprite is 50 pixels wide and 50 pixels high.

## Defining and Maintaining States

In the DOM-based version of the game code, we don't have to think about bubble state; we just queue up events with timeouts and animate/callback chains. Once a bubble is drawn to the screen at a fixed position, we leave it as is unless we need to change it. The bubble will be drawn in the same spot until we tell the browser to do something else with it.

But when we switch to canvas rendering, we need to render each bubble, with the correct sprite, on each frame redraw. Our code must track the state of all bubbles on the screen, whether they're moving, popping, falling, or just stationary. Each `bubble` object will track its current state and how long it's been in that state. We need that duration for when we draw the frames of the popping animation. The `Board` object currently keeps track of bubbles in the main layout, and we need to add to it so we can also keep track of those bubbles that are popping, falling, or firing.

### *Preparing the State Machine*

To maintain bubble state, we'll first create a set of constants that refer to a bubble's state. This is referred to as using a *state machine*, which you're likely to find increasingly useful as the complexity of your games increases. The basic principles of using a state machine, as related to this game, are as follows:

- A bubble can exist in a number of states, such as moving, popping, or falling.



Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

- The way a bubble reacts in the game will depend on the state it's in. For example, we don't want the bubble being fired to collide with a bubble being popped.
- The way a bubble is displayed may depend on its state, particularly if it's being popped.
- A bubble can be in only one state at a time; it can't be popped and popping at the same time, or popping and falling simultaneously.

Once we have the state machine set up, we'll know what we need to do to a bubble in any given situation. Some changes of state occur as a result of a user's actions, such as when they fire the bubble, but we'll also store the timestamp when a bubble enters a state. As a result, we can determine when the bubble should be moved from one state to another automatically, such as when we're in the process of popping it after a collision.

**NOTE** *In general, even if you think your game will be relatively simple, it's worth using a state machine as a way to manage complexity that you may not have thought of yet.*

Add the following to *bubble.js*:

*bubble.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Bubble = (function($){
❶ BubbleShoot.BubbleState = {
    CURRENT : 1,
    ON_BOARD : 2,
    FIRING : 3,
    POPPING : 4,
    FALLING : 5,
    POPPED : 6,
    FIRED : 7,
    FALLEN : 8
  };
  var Bubble = function(row,col,type,sprite){
    var that = this;
❷   var state;
    var stateStart = Date.now();
    this.getState = function(){ return state;};
❸   this.setState = function(stateIn){
      state = stateIn;
❹     stateStart = Date.now();
    };
❺   this.getTimeInState = function(){
      return Date.now() - stateStart;
    };
    --snip--
  };
  Bubble.create = function(rowNum,colNum,type){
    --snip--
  };
  return Bubble;
})(jQuery);
```

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

These additions allow us to store and retrieve the bubble's current state ❷, which will be one of the eight states at the top of the class ❶. Whenever we change a bubble's state ❸, we also record the timestamp when it entered that state ❹. Once we determine how long the bubble has been in its current state ❺, we can work out what to draw. For example, the amount of time a bubble has spent in the `POPPING` state determines which frame of the popping sequence to display.

### Implementing States

Each bubble can have one of the following states, which we'll need to implement:

**CURRENT** Waiting to be fired.

**ON_BOARD** Already part of the board display.

**FIRING** Moving toward the board or off the screen.

**POPPING** Being popped. This will display one of the popping animation frames.

**FALLING** An orphaned bubble that's falling from the screen.

**POPPED** Done `POPPING`. A popped bubble doesn't need to be rendered.

**FIRED** Missed the board display after `FIRING`. A fired bubble doesn't need to be rendered.

**FALLEN** Done `FALLING` off the screen. A fallen bubble doesn't need to be rendered.

The bubbles displayed in the board at the beginning of a level start out in the `ON_BOARD` state, but all other bubbles will start in the `CURRENT` state and move into one of the other states, as shown in Figure 6-3.

We'll add a couple of arrays to `Game` to keep track of those. At the top of the class, add:

*game.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Game = (function($){
  var Game = function(){
    var curBubble;
    var board;
    var numBubbles;
❶   var bubbles = [];
    var MAX_BUBBLES = 70;
    this.init = function(){
      --snip--
    };
    var startGame = function(){
      $(".but_start_game").unbind("click");
      numBubbles = MAX_BUBBLES
      BubbleShoot.ui.hideDialog();
      board = new BubbleShoot.Board();
❷     bubbles = board.getBubbles();
      curBubble = getNextBubble();
```



Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

```
      BubbleShoot.ui.drawBoard(board);
      $("#game").bind("click",clickGameScreen);
    };
    var getNextBubble = function(){
      var bubble = BubbleShoot.Bubble.create();
❸     bubbles.push(bubble);
❹     bubble.setState(BubbleShoot.BubbleState.CURRENT);
      bubble.getSprite().addClass("cur_bubble");
      $("#board").append(bubble.getSprite());
      BubbleShoot.ui.drawBubblesRemaining(numBubbles);
      numBubbles--;
      return bubble;
    };
    --snip--
  };
  return Game;
})(jQuery);
```

This new array ❶ will contain all of the bubbles in the game, both on and off the board layout. Initially, every bubble is part of the board, so the board contents can be used to populate the array ❷. Each time we call `getNextBubble`, the bubble that's ready to fire needs to be added ❸ and have its state set to `CURRENT` ❹.

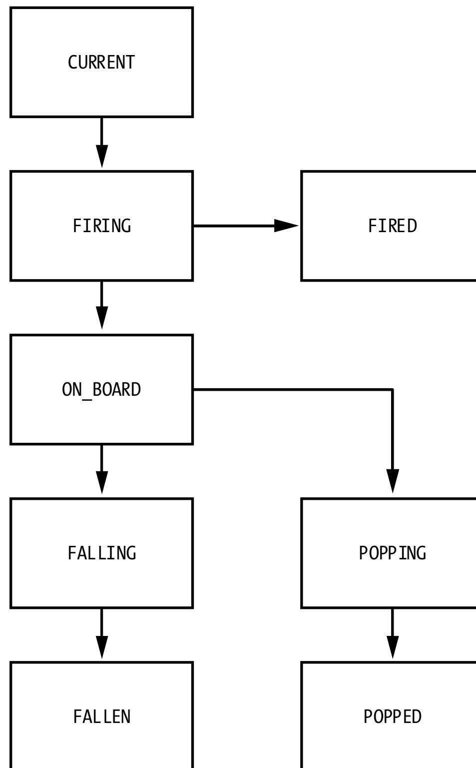


*Figure 6-3: Flowchart showing bubble states*

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

board.getBubbles is a new method that will return all of the bubbles in the rows and columns of the board as a single flat array, so add it to *board.js*:

*board.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Board = (function($){
  var NUM_ROWS = 9;
  var NUM_COLS = 32;
  var Board = function(){
    var that = this;
    --snip--
    this.getBubbles = function(){
      var bubbles = [];
      var rows = this.getRows();
      for(var i=0;i<rows.length;i++){
        var row = rows[i];
        for(var j=0;j<row.length;j++){
          var bubble = row[j];
          if(bubble){
            bubbles.push(bubble);
          };
        };
      };
      return bubbles;
    };
    return this;
  };
  --snip--
  return Board;
})(jQuery);
```

We also need to set the state of bubbles that are on the board to ON_BOARD, so make this change to the createLayout function in the same file:

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Board = (function($){
  var NUM_ROWS = 9;
  var NUM_COLS = 32;
  var Board = function(){
    --snip--
  };
  var createLayout = function(){
    var rows = [];
    for(var i=0;i<NUM_ROWS;i++){
      var row = [];
      var startCol = i%2 == 0 ? 1 : 0;
      for(var j=startCol;j<NUM_COLS;j+=2){
        var bubble = BubbleShoot.Bubble.create(i,j);
        bubble.setState(BubbleShoot.BubbleState.ON_BOARD);
        row[j] = bubble;
      };
      rows.push(row);
    };
    return rows;
  };
```


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

```
  return Board;
})(jQuery);
```

bubble.setState handles the setup, which contains the states of CURRENT and ON_BOARD, but we also need to be able to change the state of a bubble.

The two states of FIRING and FIRED will be set inside fireBubble in *ui.js*. Amend the function as follows:

*ui.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.ui = (function($){
  var ui = {
    --snip--
    fireBubble : function(bubble,coords,duration){
❶     bubble.setState(BubbleShoot.BubbleState.FIRING);
      var complete = function(){
        if(typeof(bubble.getRow()) !== undefined){
          bubble.getSprite().css(Modernizr.prefixed("transition"),"");
          bubble.getSprite().css({
            left : bubble.getCoords().left - ui.BUBBLE_DIMS/2,
            top : bubble.getCoords().top - ui.BUBBLE_DIMS/2
          });
❷         bubble.setState(BubbleShoot.BubbleState.ON_BOARD);
        }else{
❸         bubble.setState(BubbleShoot.BubbleState.FIRED);
        };
      --snip--
    },
    --snip--
  };
  return ui;
} )(jQuery);
```

When the bubble is initially fired, we set the state to FIRING ❶. If the bubble reaches the board, we set it to ON_BOARD ❷, but if it hasn't settled into a row and column, that means it missed the board, in which case it becomes FIRED ❸.

The other states will be set in *game.js*:

*game.js*

```
var Game = function(){
  --snip--
  var popBubbles = function(bubbles,delay){
    $.each(bubbles,function(){
      var bubble = this;
      setTimeout(function(){
❶       bubble.setState(BubbleShoot.BubbleState.POPPING);
        bubble.animatePop();
❷       setTimeout(function(){
          bubble.setState(BubbleShoot.BubbleState.POPPED);
        },200);
      },delay);
      board.popBubbleAt(bubble.getRow(),bubble.getCol());
      delay += 60;
    });
  };
```

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

```
  var dropBubbles = function(bubbles,delay){
    $.each(bubbles,function(){
      var bubble = this;
      board.popBubbleAt(bubble.getRow(),bubble.getCol());
      setTimeout(function(){
❸       bubble.setState(BubbleShoot.BubbleState.FALLING);
        bubble.getSprite().kaboom({
          callback : function(){
            bubble.getSprite().remove();
❹           bubble.setState(BubbleShoot.BubbleState.FALLEN);
          }
        })
      },delay);
    });
  };
};
```

In `popBubbles`, we set every bubble to `POPPING` ❶, and then after 200 milliseconds, when the popping animation has finished, we set them to `POPPED` ❷. In `dropBubbles`, we set them to `FALLING` ❸, and then when they've finished falling at the end of the `kaboom` process, they become `FALLEN` ❹.

Now that bubbles know which state they're in at any point in the game, we can start to render them onto a canvas.

## Sprite Sheets and the Canvas

We can use the existing sprite sheet PNG (*bubble_sprite_sheet.png*) from the CSS version of the game when we draw to the canvas, although we need to work with it in a different way. Rather than shifting the sprite sheet around like a background image, we'll draw part of the image that shows the correct bubble in the correct animation state. Our loading sequence will also change because we need to make sure that the sprite image is loaded before starting the game.

We'll make a new object called `Renderer` to handle drawing to the canvas, and we'll give it its own `init` method, which will preload the sprite sheet, and call that method within `game.init`.

Change the `init` method in *game.js* to the following:

*game.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Game = (function($){
  var Game = function(){
    --snip--
    this.init = function(){
❶     if(BubbleShoot.Renderer){
❷       BubbleShoot.Renderer.init(function(){
❸           $(".but_start_game").click("click",startGame);
        });
      }else{
          $(".but_start_game").click("click",startGame);
      };
    --snip--
```



Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

```
  };
  return Game;
})(jQuery);
```

First, we check if `BubbleShoot.Renderer` exists ❶. If the `Modernizr.canvas` test passes when we load in scripts, the object will exist; if canvas isn't supported, the object won't exist.

Then we call a `Renderer.init` method and pass it a function as its only parameter ❷. This is the function that attaches `startGame` to the New Game button ❸.

Now we need to write the `Renderer` object. In the blank *renderer.js* file, add the following code:

*renderer.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Renderer = (function($){
❶ var canvas;
  var context;
  var Renderer = {
❷   init : function(callback){
❸     canvas = document.createElement("canvas");
      $(canvas).addClass("game_canvas");
❹     $("#game").prepend(canvas);
❺     $(canvas).attr("width",$(canvas).width());
      $(canvas).attr("height",$(canvas).height());
      context = canvas.getContext("2d");
      callback();
    }
  };
  return Renderer;
})(jQuery);
```

We first create variables to hold the canvas that we'll use to render the game area ❶ and a reference to its rendering context, so we don't have to call `canvas.getContext("2d")` constantly.

In the `init` method, we accept the callback function as a parameter ❷, create the `canvas` DOM element ❸, and then prepend it in the game `div` ❹. We also explicitly set the width and height attributes of the `canvas` ❺. Remember that these attributes define the number of pixels and the boundaries of the canvas internally, so for simplicity, we set them to the same dimensions as those rendered to the screen.

That will create the `canvas` element for us and prime a context ready to be drawn into. We need to set the width and height of `game_canvas`, so add the following into *main.css*:

*main.css*

```
.game_canvas
{
  width: 1000px;
  height: 620px;
}
```

Build an HTML5 Game
© 2015 Karl Bunyan

www.itbook.store/books/9781593275754

The DOM-rendered version uses jQuery to move objects around the screen, but we won't have DOM elements to manipulate inside a canvas, so there's nothing for jQuery to work with. Hence, we'll have to keep track of the position of every bubble on the screen with new code. Much of this will happen inside the new *sprite.js* file we've created.

### MULTIPLE RENDERING METHODS: TWO APPROACHES

If you need to support different rendering methods, as we are here, you can take two approaches. First, you can create a class for each rendering method and provide identical sets of methods and properties so they can be used interchangeably. This is what we're doing with *Bubble Shooter*.

Second, you can create a single class for both rendering methods and then have code inside that branches depending on which rendering method is supported. The new class may act as just a wrapper for a different class for each method. For example, for *Bubble Shooter*, we could create something like the following pseudocode:

```
BubbleShoot.SpriteWrapper = (function($){
❶  var SpriteWrapper = function(id){
     var wrappedObject;
❷    if(BubbleShoot.Renderer){
❸      wrappedObject = getSpriteObject(id);
     }else{
❹      wrappedObject = getJQueryObject(id);
     }
❺    this.position = function(){
       return wrappedObject.position();
     };
   };
   return SpriteWrapper;
})(jQuery);
```

Here, we would pass in some kind of identifier to an object constructor ❶ and then branch the code depending on how we'll render the game ❷. We would need new functions to return either a `Sprite` ❸ or a `jQuery` ❹ object, which would be stored inside the class in `wrappedObject`.

From then on, if we wanted to find the position of the object, we would call the `position` method ❺ and know we would get correct data whether the object was being rendered in the DOM or on the canvas.

The main reason we're not taking this approach with *Bubble Shooter* is that we have only one type of sprite—the bubbles on the screen. These are represented well enough by the `Bubble` class, which acts as a wrapper anyway. However, if we were dealing with many different kinds of sprites, we might want to split the structure more explicitly.


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

We'll write *sprite.js* so that canvas sprites can be called with the same methods that we're using on jQuery sprites. The main methods we've been calling are `position`, `width`, `height`, and `css`, and if we create implementations of these in *sprite.js*, the `Sprite` class will look like a jQuery object as far as the rest of our code is concerned.

Add the following to *sprite.js*:

*sprite.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Sprite = (function($){
  var Sprite = function(){
    var that = this;
❶   var left;
    var top;
❷   this.position = function(){
      return {
        left : left,
        top : top
      };
    };
❸   this.setPosition = function(args){
      if(arguments.length > 1){
        return;
      };
      if(args.left !== null)
        left = args.left;
      if(args.top !== null)
        top = args.top;
    };
❹   this.css = this.setPosition;
    return this;
  };
❺ Sprite.prototype.width = function(){
    return BubbleShoot.ui.BUBBLE_DIMS;
  };
❻ Sprite.prototype.height = function(){
    return BubbleShoot.ui.BUBBLE_DIMS;
  };
❼ Sprite.prototype.removeClass = function(){};
  Sprite.prototype.addClass = function(){};
  Sprite.prototype.remove = function(){};
  Sprite.prototype.kaboom = function(){
    jQuery.fn.kaboom.apply(this);
  };
  return Sprite;
})(jQuery);
```

Here, we've created an object that implements many of the methods that we access for jQuery objects. We have left and top coordinates ❶ and a `position` method ❷ that returns those coordinates in the same way that a call to jQuery's `position` method would. The `setPosition` method can set the top and left coordinates ❸ or do nothing if other values are passed.

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

In our DOM-based version of the game, we call the css method to set the screen coordinates of an object. setPosition has been constructed to accept the same arguments as the css method, and to spare us from having to rewrite code anywhere that the css method is called and using setPosition for the canvas version, we can create a css method of Sprite and alias it to setPosition ❹.

The width ❺ and height ❻ methods return the values defined for a bubble's dimensions in *ui.js*. Finally, we define empty methods for removeClass, addClass, and remove, which maintain compatibility with a lot of our existing code ❼. Anywhere these last methods are called will not affect the display but will also not throw an error.

When a bubble is created, we need to decide whether to create a jQuery object or an instance of Sprite, depending on whether we're rendering using the DOM or canvas. We'll do this inside the bubble creation process in *bubble.js*:

*bubble.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Bubble = (function($){
  --snip--
  var Bubble = function(row,col,type,sprite){
    --snip--
  };
  Bubble.create = function(rowNum,colNum,type){
    if(!type){
      type = Math.floor(Math.random() * 4);
    };
❶   if(!BubbleShoot.Renderer){
      var sprite = $(document.createElement("div"));
      sprite.addClass("bubble");
      sprite.addClass("bubble_" + type);
    }else{
❷     var sprite = new BubbleShoot.Sprite();
    }
    var bubble = new Bubble(rowNum,colNum,type,sprite);
    return bubble;
  };
  return Bubble;
})(jQuery);
```

This code checks again that the Renderer object is loaded ❶ (which happens if canvas is enabled) and, if not, continues the DOM-based path. Otherwise, we make a new Sprite object ❷. With this in place, a call to curBubble.getSprite will return a valid object no matter whether we're using jQuery with CSS or a pure canvas route.

The last part of initializing the Sprite objects is to make sure they have the correct onscreen coordinates. In the DOM version of the game, we set


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

these in the CSS, but with the canvas, we have to set them in JavaScript code. These will be set in the `createLayout` function in *board.js*:

*board.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Board = (function($){
  var NUM_ROWS = 9;
  var NUM_COLS = 32;
  var Board = function(){
    --snip--
    return this;
  };
  var createLayout = function(){
    var rows = [];
    for(var i=0;i<NUM_ROWS;i++){
      var row = [];
      var startCol = i%2 == 0 ? 1 : 0;
      for(var j=startCol;j<NUM_COLS;j+=2){
        var bubble = BubbleShoot.Bubble.create(i,j);
        bubble.setState(BubbleShoot.BubbleState.ON_BOARD);
❶       if(BubbleShoot.Renderer){
❷         var left = j * BubbleShoot.ui.BUBBLE_DIMS/2;
          var top = i * BubbleShoot.ui.ROW_HEIGHT;
❸         bubble.getSprite().setPosition({
            left : left,
            top : top
          });
        };
        row[j] = bubble;
      };
      rows.push(row);
    };
    return rows;
  };
  return Board;
})(jQuery);
```

If the renderer exists ❶, we calculate the left and top coordinates of where the bubble should be displayed ❷ and then set the sprite's properties to those values ❸.

The current bubble also needs its position set, so this will happen inside `getNextBubble` in *game.js*:

*game.js*

```
var BubbleShoot = window.BubbleShoot || {};
  BubbleShoot.Game = (function($){
  var Game = function(){
    --snip--
    var getNextBubble = function(){
      var bubble = BubbleShoot.Bubble.create();
```

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

```
      bubbles.push(bubble);
      bubble.setState(BubbleShoot.BubbleState.CURRENT);
      bubble.getSprite().addClass("cur_bubble");
      var top = 470;
      var left = ($("#board").width() - BubbleShoot.ui.BUBBLE_DIMS)/2;
      bubble.getSprite().css({
        top : top,
        left : left
      });
      $("#board").append(bubble.getSprite());
      BubbleShoot.ui.drawBubblesRemaining(numBubbles);
      numBubbles--;
      return bubble;
    };
    --snip--
  };
  return Game;
})(jQuery);
```

We now have all bubble positions tracked and know their state at all times. We can also manipulate a sprite representation, but nothing will appear on the screen just yet. In the next section, we'll render our sprites to the canvas.

## The Canvas Renderer

To animate anything on the canvas, we need to clear pixels before each redraw. To render the game, we'll use `setTimeout` with a timer to redraw the position and state of every bubble on a frame-by-frame basis. This process will be the same for just about any game you build and, certainly, for anything where the display is constantly being updated. In theory, we only need to redraw the canvas when information on the screen has changed; in practice, working out when there's new information to show can be difficult. Fortunately, canvas rendering is so fast that there's generally no reason not to just update the display as often as possible.

We'll store the value of the timeout ID returned by `setTimeout` so we know whether or not the frame counter is running. This will happen at the top of *game.js* in a new variable called `requestAnimationID`, where we'll also store a timestamp for when the last animation occurred:

*game.js*

```
var BubbleShoot = window.BubbleShoot || {};
  var Game = function(){
    var curBubble;
    var board;
    var numBubbles;
    var bubbles = [];
    var MAX_BUBBLES = 70;
❶   var requestAnimationID;
    this.init = function(){
    };
    --snip--
```



Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

```
    var startGame = function(){
      $(".but_start_game").unbind("click");
      $("#board .bubble").remove();
      numBubbles = MAX_BUBBLES;
      BubbleShoot.ui.hideDialog();
      board = new BubbleShoot.Board();
      bubbles = board.getBubbles();
❷     if(BubbleShoot.Renderer)
      {
        if(!requestAnimationID)
❸         requestAnimationID = setTimeout(renderFrame,40);
      }else{
        BubbleShoot.ui.drawBoard(board);
      };
      curBubble = getNextBubble(board);
      $("#game").bind("click",clickGameScreen);
    };
  };
  return Game;
})(jQuery);
```

We add the two variables ❶, and if the `Renderer` object exists ❷, we start the timeout running to draw the first animation frame ❸.

We haven't written `renderFrame` yet, but before we do, we'll write a method in *renderer.js* to draw all of the bubbles. The method will accept an array of `bubble` objects as an input.

First we need to load the bubble images into *renderer.js*:

*renderer.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Renderer = (function($){
  var canvas;
  var context;
❶ var spriteSheet;
❷ var BUBBLE_IMAGE_DIM = 50;
  var Renderer = {
    init : function(callback){
      canvas = document.createElement("canvas");
      $(canvas).addClass("game_canvas");
      $("#game").prepend(canvas);
      $(canvas).attr("width",$(canvas).width());
      $(canvas).attr("height",$(canvas).height());
      context = canvas.getContext("2d");
      spriteSheet = new Image();
❸     spriteSheet.src = "_img/bubble_sprite_sheet.png";
❹     spriteSheet.onload = function() {
        callback();
      };
    }
  };
  return Renderer;
})(jQuery);
```

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

We create a variable to hold the image data ❶ and define another variable for the width and height of each bubble image ❷. The dimensions will tell us where to crop each image within the sprite sheet. We then load in the image file ❸, and the callback function that's passed into `init` is triggered after the image has loaded ❹.

Next we'll create the function to draw the sprites onto the canvas.

*renderer.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Renderer = (function($){
  --snip--
  var Renderer = {
    init : function(callback){
      --snip--
    },
❶    render : function(bubbles){
      context.clearRect(0,0,canvas.width,canvas.height);
      context.translate(120,0);
❷      $.each(bubbles,function(){
        var bubble = this;
❸        var clip = {
          top : bubble.getType() * BUBBLE_IMAGE_DIM,
          left : 0
        };
❹        Renderer.drawSprite(bubble.getSprite(),clip);
      });
      context.translate(-120,0);
    },
    drawSprite : function(sprite,clip){
❺      context.translate(sprite.position().left + sprite.width()/2,sprite.
        position().top + sprite.height()/2);
❻      context.drawImage(spriteSheet,clip.left,clip.top,BUBBLE_IMAGE_DIM,
        BUBBLE_IMAGE_DIM,-sprite.width()/2,-sprite.height()/2,BUBBLE_IMAGE_
        DIM,BUBBLE_IMAGE_DIM);
❼      context.translate(-sprite.position().left - sprite.width()/2,
        -sprite.position().top - sprite.height()/2);
    }
  };
  return Renderer;
})(jQuery);
```

First, we create a render method that accepts an array of `Bubble` objects ❶. We then clear the canvas and offset the context by 120 pixels so the board display is drawn in the center of the screen. The code then loops over each bubble in the array ❷ and defines an (*x*,*y*) coordinate from which to extract the bubble's sprite from the image ❸. The *x*-coordinate always starts at zero until we add frames for the popping animation, and the *y*-coordinate is the bubble type (0 to 3) multiplied by the height of a bubble image (50 pixels). We pass this information along with the bubble's `Sprite` object to another new method called `drawSprite` ❹ before resetting the context position.



Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

Inside `drawSprite`, we translate the context ❺ by the coordinates of the sprite, remembering to offset the (top,left) coordinates by half of (width,height) to get the center of the image, and then draw the image ❻. In general, it's best to translate the canvas context so its origin is at the center of any image being drawn, because the `rotate` method of the context performs rotations around the context origin. This means that if we want to rotate an image around its center, we already have the context set up correctly to do so.

Finally, after calling `drawImage`, we translate the context back to the origin ❼. To see the board being rendered to the canvas, we just need to put `renderFrame` into *game.js*:

*game.js*

```
var BubbleShoot = window.BubbleShoot || {};
  var Game = function(){
    --snip--
    var renderFrame = function(){
      BubbleShoot.Renderer.render(bubbles);
      requestAnimationID = setTimeout(renderFrame,40);
    };
  };
  return Game;
})(jQuery);
```

Reload the page in your browser to start the game again. After clicking New Game, you should see the board render in its initial state. However, firing a bubble produces no animation, and neither does popping, falling, or anything else. In the next section, we'll get bubble firing working again and also animate the bubble popping. If you open the game in a browser that doesn't support canvas, then the game will still work as before because we have left the DOM version intact. Next, we'll add animation to the canvas version.

## Moving Sprites on the Canvas

With the CSS version of the game, we used jQuery to move objects around on the screen with one call to the `animate` method. For canvas animation, we need to calculate and update movements manually.

The process of animating on the canvas is the same as jQuery's internal processes, and we'll give `Sprite` an `animate` method so we can continue to use our existing code. The `animate` method will do the following:

1. Accept destination coordinates for a bubble and the duration of the movement.
2. Move the object a small distance toward those coordinates by a value proportional to the time elapsed since the last frame.
3. Repeat step 2 until the bubble reaches its destination.

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

This process is identical to the one that happens when we use jQuery's `animate` method and is one you'll use just about any time you want to move an object around the screen.

The `renderFrame` method, which is already called during each frame, will run the entire animation process. After the bubble sprites calculate their own coordinates, `renderFrame` will trigger the drawing process. We'll add an `animate` method to the `Sprite` object so our existing game logic will work without us having to rewrite our code. Remember that when we call `animate` in *ui.js*, we pass in two parameters:

- An object specifying `left` and `top` position coordinates
- An object specifying `duration`, `callback` function, and `easing`

By constructing the `animate` method of `Sprite` to take the same parameters, we can avoid making any changes to the call in *ui.js*. Add the following to *sprite.js*:

*sprite.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Sprite = (function($){
  var Sprite = function(){
    --snip--
    this.css = function(args){
      --snip--
    };
❶   this.animate = function(destination,config){
❷     var duration = config.duration;
❸     var animationStart = Date.now();
❹     var startPosition = that.position();
❺     that.updateFrame = function(){
        var elapsed = Date.now() - animationStart;
        var proportion = elapsed/duration;
        if(proportion > 1)
          proportion = 1;
❻       var posLeft = startPosition.left + (destination.left - startPosition.
          left) * proportion;
        var posTop = startPosition.top + (destination.top - startPosition.top)
          * proportion;
❼       that.css({
          left : posLeft,
          top : posTop
        });
      };
❽     setTimeout(function(){
❾       that.updateFrame = null;
❿       if(config.complete)
          config.complete();
      },duration);
    };
    return this;
  };
  --snip--
  return Sprite;
})(jQuery);
```


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

The `destination` parameter passed into `animate` ❶ represents the sprite's destination coordinates, which are contained in an object that looks like this:

```
{top: 100,left: 100}
```

We also pass a configuration object, which will have a `duration` property ❷, plus an optional post-animation callback function to run when the animation is over.

Next, we set a start time for the animation ❸ and store the starting position ❹. These will both be used to calculate a bubble's position at any time.

We dynamically add the `updateFrame` method onto the `Sprite` object ❺ so we can call it each frame to recalculate a bubble's position. Inside `updateFrame`, we calculate how much of the animation is completed. In case the last timeout is called after the animation has completed, we ensure that the proportion is never greater than 1 so that a bubble never moves past its target destination. The new coordinates are calculated ❻ with the following equations:

$$\text{current } x = \text{start } x + (\text{final } x - \text{start } x) \times \text{proportion elapsed}$$

$$\text{current } y = \text{start } y + (\text{final } y - \text{start } y) \times \text{proportion elapsed}$$

Once we have the new top and left coordinates, the position of the sprite is updated with a call to its `css` method ❼. We don't need `updateFrame` to run when the object has finished moving, so a timeout call is set ❽ to remove the method after `duration` ❾ passes, which is when the animation will be complete. This also calls any post-animation function that was passed in as the `callback` property of the `config` variable ❿.

Now that we can calculate a bubble's new coordinates, add a call to `updateFrame` in *game.js*:

*game.js*

```
var BubbleShoot = window.BubbleShoot || {};
  var Game = function(){
    --snip--
    var renderFrame = function(){
❶      $.each(bubbles,function(){
❷        if(this.getSprite().updateFrame)
❸          this.getSprite().updateFrame();
      });
      BubbleShoot.Renderer.render(bubbles);
      requestAnimationID = setTimeout(renderFrame,40);
    };
  };
  return Game;
})(jQuery);
```

Each time `renderFrame` is called on a bubble ❶, if the method `updateFrame` is defined ❷, we call that method ❸.

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

We also need to call animate in fireBubble in *ui.js* by checking for the existence of BubbleShoot.Renderer again. We know that BubbleShoot.Renderer will exist only if canvas is supported, and we want to use the canvas for rendering if that is the case. The outcome is that CSS transitions will animate the bubbles only if CSS transitions are supported *and* canvas rendering isn't supported.

*ui.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.ui = (function($){
  var ui = {
    --snip--
    fireBubble : function(bubble,coords,duration){
      --snip--
      if(Modernizr.csstransitions && !BubbleShoot.Renderer){
        --snip--
      }else{
        --snip--
      }
    },
    --snip--
  };
  return ui;
} )(jQuery);
```

Reload the game and fire away! You should now have a working game again, but this time all the images are rendered onto the canvas. But now there's no popping animation because we're not handling changes in bubble state in the display. The game state is internally correct, but the screen isn't entirely in sync because we never see a bubble popping. Rendering the bubbles in their correct state is the focus of the next section.

## Animating Canvas Sprite Frames

Currently, every bubble is rendered in the same visual state regardless of whether it's sitting in the board, popping, newly fired, and so on. Bubbles remain on the screen after they've been popped, and we're missing out on the popping animation! This happens because bubbles are never deleted from the bubbles array in Game, so they're rendered even after they've been deleted from the Board object.

We already know which state a bubble is in, and we have the sprite sheet image loaded into memory to access all of the animation states. Drawing the correct state involves making sure that the drawSprite method of Renderer is either called with the correct state for a visible bubble or skipped entirely for any bubbles that have been popped or dropped off the screen. The changes in a bubble's appearance that we need to implement are listed by state in Table 6-1.


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

**Table 6-1:** Visual Changes Based on Bubble State

| Bubble's state in code | Visual displayed to the player |
|---|---|
| CURRENT_BUBBLE | No change |
| ON_BOARD | No change |
| FIRING | No change |
| POPPING | Render one of four bubble frames, depending on how long the bubble has been POPPING |
| FALLING | No change |
| POPPED | Skip rendering |
| FALLEN | Skip rendering |
| FIRED | Skip rendering |

Those changes will happen inside Renderer.render. We'll loop over the entire bubble array and either skip the rendering stage or adjust the coordinates to clip the sprite sheet for the correct stage in the popping animation. Make the following change to *renderer.js*:

*renderer.js*

```
var BubbleShoot = window.BubbleShoot || {};
BubbleShoot.Renderer = (function($){
  --snip--
  var Renderer = {
    init : function(callback){
      --snip--
    },
    render : function(bubbles){
      bubbles.each(function(){
        var bubble = this;
        var clip = {
          top : bubble.getType() * BUBBLE_IMAGE_DIM,
          left : 0
        };
❶       switch(bubble.getState()){
          case BubbleShoot.BubbleState.POPPING:
❷           var timeInState = bubble.getTimeInState();
❸           if(timeInState < 80){
              clip.left = BUBBLE_IMAGE_DIM;
❹           }else if(timeInState < 140){
              clip.left = BUBBLE_IMAGE_DIM*2;
❺           }else{
              clip.left = BUBBLE_IMAGE_DIM*3;
            };
            break;
❻         case BubbleShoot.BubbleState.POPPED:
            return;
❼         case BubbleShoot.BubbleState.FIRED:
            return;
❽         case BubbleShoot.BubbleState.FALLEN:
            return;
        }
```

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754

```
❾         Renderer.drawSprite(bubble.getSprite(),clip);
        });
      },
      drawSprite : function(sprite,clip){
        --snip--
      }
    };
    return Renderer;
})(jQuery);
```

First, we want to see which state the bubble is in ❶. To do this, we'll use a `switch` statement. State machines are often written using `switch`/`case` statements rather than multiple `if`/`else` statements. Using this structure not only makes it easier to add any future states but also provides a clue to others reading the code in the future that they're looking at a state machine.

If the bubble is popping, we want to know how long it's been in that state ❷. That time determines which animation frame to fetch. We use the unpopped state for the first 80 milliseconds ❸, the first frame for the next 60 milliseconds ❹, and the final popping frame from that point until the `POPPING` state is cleared ❺.

If the bubble is in the `POPPED` ❻, `FIRED` ❼, or `FALLEN` ❽ states, we return and skip rendering altogether. Otherwise, we call `drawSprite` as before ❾.

Now if you reload the game, it should completely work again. Without making drastic changes, we've refactored our entire game area to use either canvas- or DOM-based rendering, depending on browser compatibility. The browser you use to load the game and the features that browser supports will determine how *Bubble Shooter* is presented to you:

- If your browser supports the `canvas` element, you'll see that version.
- If your browser supports CSS transitions but *not* the `canvas` element, you'll see the CSS transition version.
- If neither of the above is supported, you'll see the DOM version animated with jQuery.

## Summary

That covers most of the core of drawing the graphics elements of an HTML5 game, whether you're using HTML and CSS or an entirely canvas-based approach. But that doesn't mean we've finished the game! We have no sound, only one level of play exists, and a scoring system would be nice. In the next chapter, we'll implement these elements and explore a few more features of HTML5, including local storage for saving game state, `requestAnimationFrame` for smoother animations, and how to make sound work reliably.


Build an HTML5 Game
© 2015 Karl Bunyan
www.itbook.store/books/9781593275754

## Further Practice

1. When bubbles pop, the animation plays identically for every bubble. Experiment with changing the timing so that some bubbles play the animation faster and some slower. Also, try adding some rotation to the bubbles as they're drawn onto the canvas. This should give the popping animation a much richer feel for very little effort.
2. When orphaned bubbles fall, they remain as the default sprite. Change *renderer.js* so that bubbles pop as they're falling.

Build an HTML5 Game
© 2015 Karl Bunyan



www.itbook.store/books/9781593275754