HTML5 2D game development: Introducing Snail Bait

Getting started with your first platform video game

David Geary

August 28, 2012

In this series, HTML5 maven David Geary shows you how to implement an HTML5 2D video game one step at a time. This initial installment shows you the finished game and then gets you started implementing it from scratch. If you've ever wanted to create an HTML5 game but didn't have time to master all the details, this is the series for you.

View more content in this series

The great thing about software development is that, within reason, you can make anything you can imagine come to life on screen. Unencumbered by physical constraints that hamper engineers in other disciplines, software developers have long used graphics APIs and UI toolkits to implement creative and compelling applications. Arguably the most creative genre of software development is game programming; few endeavors are more rewarding from a creative standpoint than making the vision you have for a game become a reality.

Platform video games

Donkey Kong, *Mario Bros.*, *Sonic the Hedgehog*, and *Braid* are all well-known, best-selling games, and they are all platformers. At one time platformers represented up to one third of all video game sales. Today, their market share is drastically lower, but there are still many successful platform games.

Rewarding, however, does not mean *easy*; in fact, it means the opposite. Implementing games, especially video games, requires a nontrivial understanding of programming, a good grasp of graphics and animation, and lots of math blended with substantial doses of art and creativity. And that's just the beginning. Successful game developers spend a lot of time polishing their games by refining gameplay and graphics, in addition to implementing many aspects of the game that have nothing to do with gameplay — such as scoreboards, instructions, animations between lives and levels, and endgame sequences.

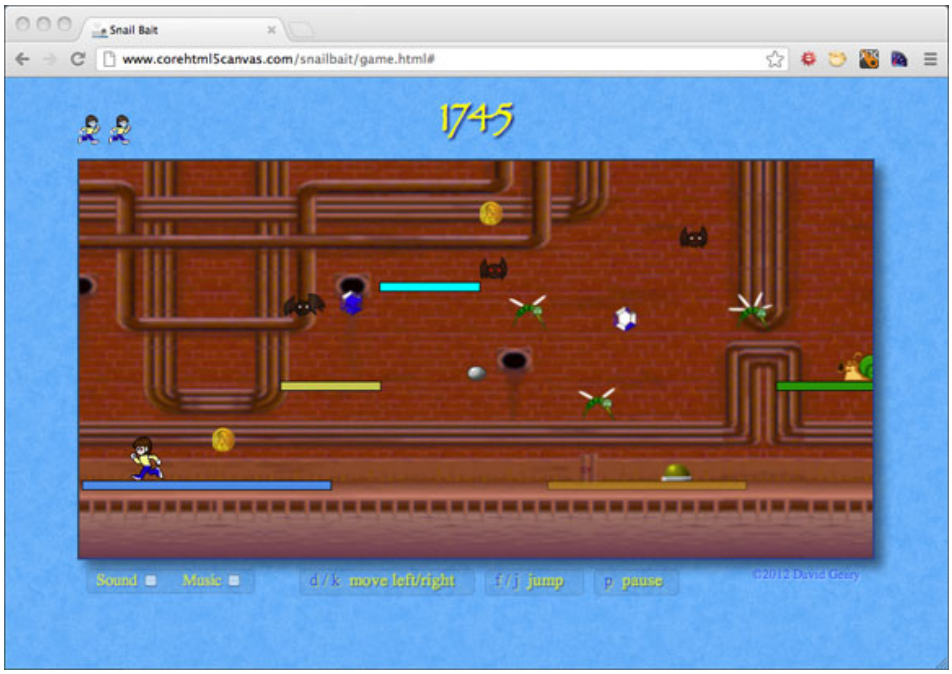
The goal of this series is to show you how to implement an HTML5 video game so you can start working on one of your own.

View the "About this series" video transcript here.

The game: Snail Bait

In this series, I will show you how to implement a platform video game primarily with the HTML5 Canvas API. That game is Snail Bait, shown in Figure 1. You can play it online; see Related topics for a link to the game. Make sure that your browser has hardware acceleration for Canvas (just recently implemented in most browsers, including Chrome since version 18); otherwise, Snail Bait performance will be extremely poor. (See the HTML5 Canvas performance sidebar for more information.)

Figure 1. Snail Bait running in Chrome



HTML5 technologies used in Snail Bait

- Canvas (2D API)
- Timing control for script-based animations
- Audio
- CSS3 (transitions and media queries)

Snail Bait is a classic platform game. The protagonist, whom I'll refer to simply as the *runner*, runs along and jumps between floating platforms that move horizontally. The runner's ultimate goal is to get to a pulsating platform with a gold button at the end of the level. The runner, pulsating platform, and gold button are all shown in Figure 1.

The player controls the runner with the keyboard: **d** moves the runner left, **k** moves her to the right, **j** or **f** make her jump, and **p** pauses the game.

When the game begins, you have three lives. Runner icons representing the number of lives that remain are displayed above and to the left of the game's canvas, as you can see in Figure 1. In the runner's quest to make it to the end of the level, she must avoid bad guys — bees, bats, and a snail — while trying to capture valuable items such as pennies, rubies, and sapphires. If the runner collides with bad guys, the runner blows up, you lose a life, and you must go back to the beginning

of the level. When the runner collides with good guys, your score increases and you are rewarded with a pleasant sound effect.

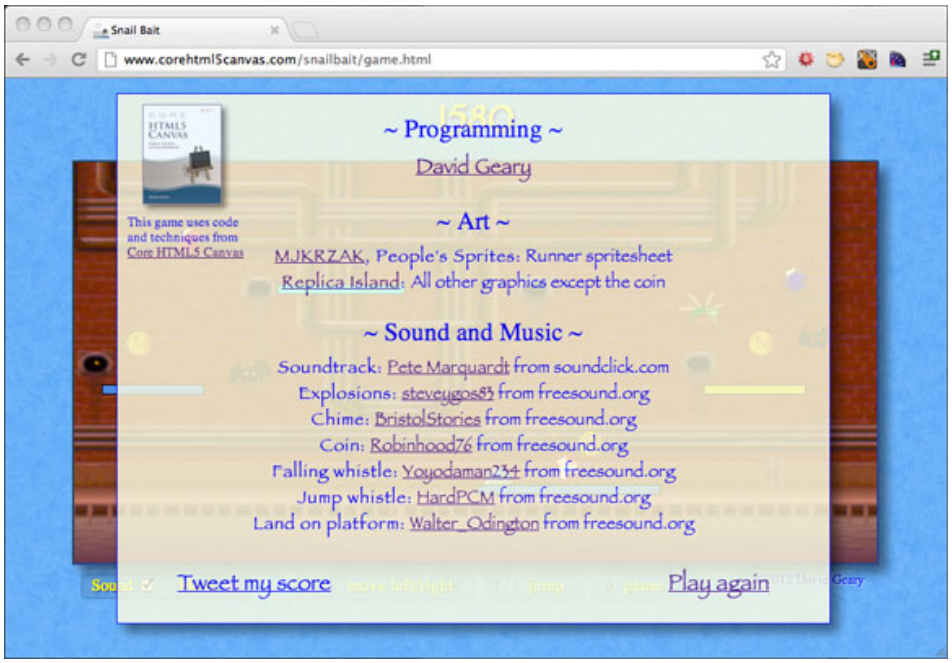
WASD?

By convention, computer games often use the w, a, s, and d keys to control play. That convention evolved primarily because it lets right-handed players use the mouse and keyboard simultaneously. It also leaves the right hand free to press the spacebar or modifier keys such as CTRL or ALT. Snail Bait doesn't use WASD because it doesn't receive input from the mouse or modifier keys. But you can easily modify the game's code to use any combination of keys.

Bad guys mostly just hang around waiting for the runner to collide with them. However, the snail periodically shoots snail bombs (the silver ball shown near the center of Figure 1). The bombs, like the other bad guys, blow up the runner when they hit her.

The game ends in one of two ways: you lose all three lives, or you make it to the pulsating platform (with bonus points for landing on the gold button). Either way, the game ends with the credits shown in Figure 2:

Figure 2. Game credits



What you cannot see in Figure 1 is that everything — with the exception of the runner, whose movement you control — continuously scrolls. That scrolling further categorizes Snail Bait as a *side-scroller* platform game. However, that's not the only motion in the game, which leads me to sprites and their behaviors.

Sprites: The cast of characters

HTML5 Canvas performance

It wasn't long ago that most browsers had implemented hardware acceleration for CSS transitions but had not yet done so for Canvas. Canvas has always been relatively fast,

especially compared to other graphics systems such as Scalable Vector Graphics (SVG), but Canvas without hardware acceleration is no match for hardware-accelerated anything.

All modern browsers now hardware-accelerate Canvas elements. So does iOS 5, which means that Canvas-based video games with smooth animation are now possible not only on the desktop, but also on Apple's mobile devices.

With the exception of the background, everything in Snail Bait is a *sprite*. A sprite is an object that you can paint on the game's canvas. Sprites are not a part of the Canvas API, but they are simple to implement. The game's sprites are:

- Platforms (inanimate objects)
- Runner (protagonist)
- Bees and bats (bad)
- Buttons (good)
- Rubies and sapphires (good)
- Coins (good)
- Snails (bad)
- Snail bombs (bad)

Besides scrolling from right to left, nearly all of the game's sprites have independent motion of their own. For example, the rubies and sapphires bob up and down at varying rates of speed, and the buttons and snail pace back and forth along the length of the platform on which they reside.

Replica Island

The idea for sprite behaviors — which are an example of the Strategy design pattern — comes from Replica Island, a popular open source Android platform game. Most of Snail Bait's graphics are from Replica Island (used with permission). See Related topics for links to the Strategy design pattern on Wikipedia and the home page for Replica Island.

That independent motion is one of many sprite *behaviors*. Sprites can have other behaviors that have nothing to do with motion; for example, besides bobbing up and down, the rubies and sapphires sparkle.

Each sprite has an *array* of behaviors. A behavior is simply an object with an `execute()` method. In every animation frame, the game invokes each behavior's `execute()` method. In that method, behaviors manipulate their associated sprites in some manner depending on game conditions. For example, when you press k to move the runner to the right, the runner's *move laterally* behavior subsequently moves the runner to the right in every animation frame until you change her direction. Another behavior, *run in place*, periodically changes the runner's image so it appears as though the runner is running in place. Those two behaviors combine to make it appear as though the runner is running either left or right.

Table 1 lists the game's sprites and their respective behaviors:

Table 1. Snail Bait's sprites and behaviors

Sprite	Behaviors

Platforms	• Move horizontally (all sprites except the runner and snail bombs move in concert with the platforms)
Runner	• Run in place • Move laterally • Jump • Fall • Collide with bad guys and explode • Collide with good guys and get points
Bees and bats	• Hover • Flap wings
Buttons	• Pace • Collapse • Varies: Make bad guys explode or End level
Coins, rubies, and sapphires	• Sparkle • Bob up and down • Pace
Snails	• Pace • Shoot bombs
Snail bombs	• Move right to left (faster than platforms) • Collide with runner and disappear

Subsequent articles in the series will give you an in-depth look at sprites and sprite behaviors. For now, to give you a high-level overview, Listing 1 shows how the game creates the `runner` sprite:

Listing 1. Creating sprites

```
var runInPlace = {  // Just an object with an execute method
   execute: function (sprite, time, fps) {
      // Update the sprite's attributes based on the time and frame rate
   }
};

var runner = new Sprite('runner', // name
                        runnerPainter, // painter
                        [ runInPlace,... ]); // behaviors
```

A `runInPlace` object is defined and passed, in an array with other behaviors, to the runner sprite's constructor. While it's running, the game invokes the `runInPlace` object's `execute()` method for every animation frame.

HTML5 game development best practices

Freely available assets

Most game developers need some help with their graphics, sound effects, and music. Fortunately, an abundance of assets are freely available under various licensing arrangements.

Snail Bait uses:

- Sound effects from freesound.org
- Soundtrack from soundclick.com
- Runner sprite from panelmonkey.org (site has been hacked)
- All other graphics from Replica Island

I'll discuss game development best practices throughout this series, starting here with five that are specific to HTML5:

- Pause the game when the window loses focus
- Implement a countdown when the window regains focus
- Use CSS3 transitions
- Detect and react to slowly running games
- Incorporate social features

I'll examine these five in detail in subsequent articles in this series; for now, here's a quick look at each of them.

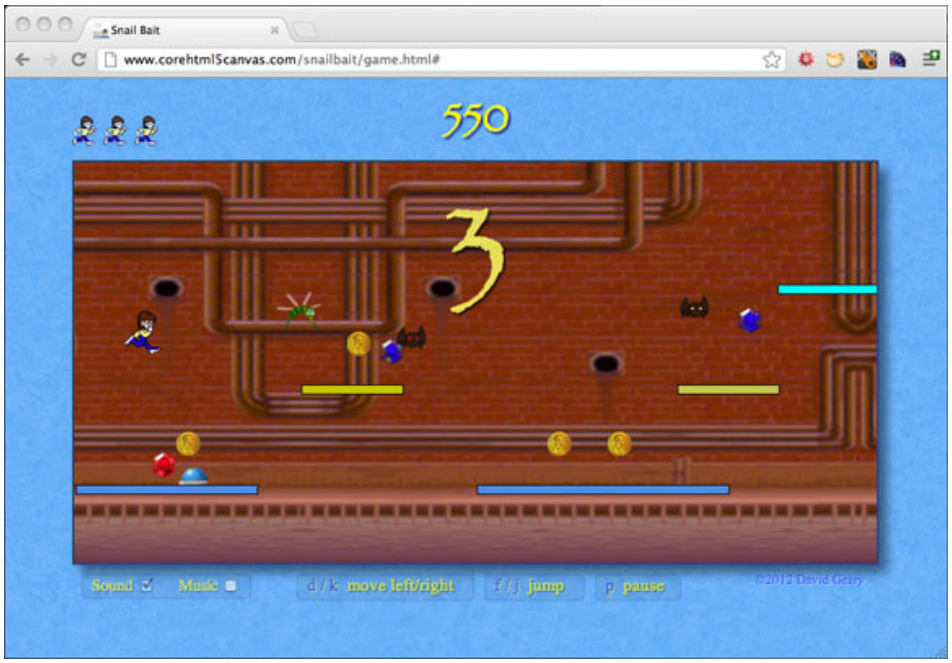
1. Pause the game when the window loses focus

If an HTML5 game is running in a browser and you change focus to another tab or browser window, most browsers severely clamp the frame rate at which the game's animation runs to save resources such as CPU and battery power. That frame-rate clamping, in turn, almost always wreaks havoc with collision-detection algorithms, which expect the game to be running at a minimum frame rate. To avoid frame-rate limitations and the ensuing collision-detection meltdown, you should automatically pause the game when the window loses focus.

2. Implement a countdown when the window regains focus

When your game's window regains focus, it's a good idea to give the user a few seconds to prepare for the game to restart. Snail Bait uses a three-second countdown when the window regains focus, as shown in Figure 3:

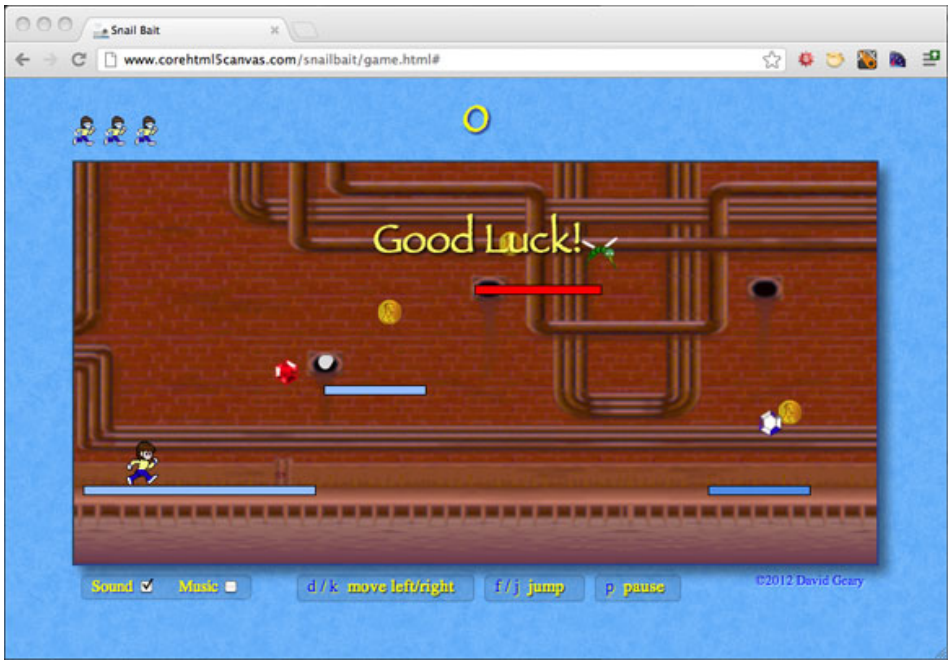
Figure 3. Snail Bait auto-pause



3. Use CSS3 transitions

Figure 4 is a screenshot taken after the game loads:

Figure 4. CSS3 effects



There are two things to notice about Figure 4. First, a *toast* — something that is briefly shown to the player — is visible that says *Good luck!*. That toast fades in when the game loads, and after five seconds, it fades out. Second, notice the check boxes (for sound and music) and instructions (telling which keystrokes perform which functions) below the game's canvas. When the game starts, the check boxes and instructions are fully opaque, as they are in Figure 4; after play begins, those elements slowly fade out until they are barely visible (as shown in Figure 3), so that they don't become a distraction.

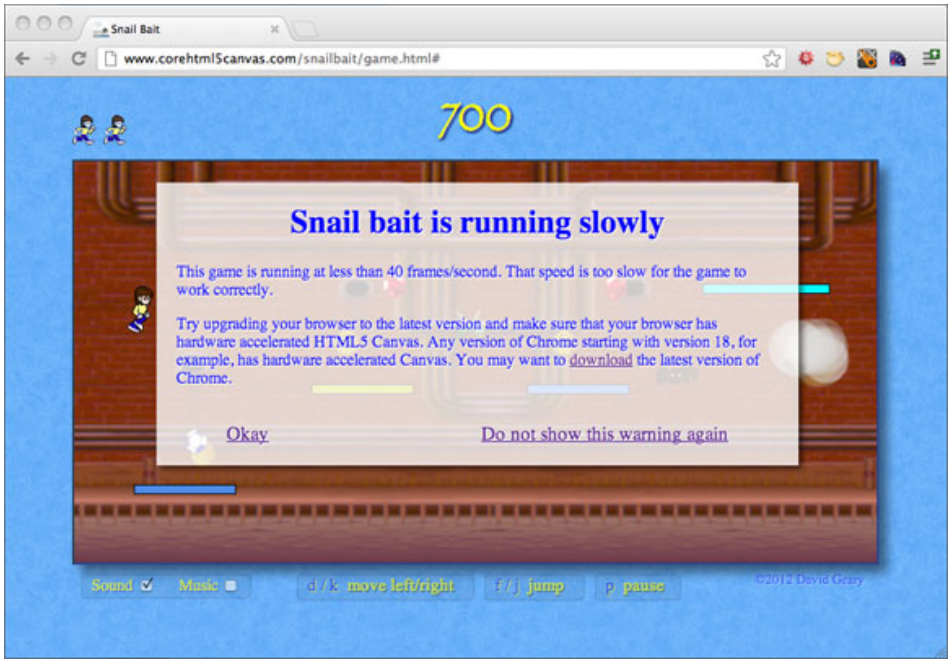
Snail Bait dims elements and makes toasts fade with CSS3 transitions.

4. Detect and react to slowly running games

Unlike console games, which run in a tightly controlled environment, HTML5 games run in a highly variable, unpredictable, and chaotic one. It won't be uncommon for your game to run unacceptably slowly when players are playing YouTube videos in another tab or are otherwise overworking the CPU or GPU. And there's always the possibility that your players will use a browser that can't keep up.

As a game developer, you must anticipate this unfortunate confluence of events and react accordingly. Snail Bait constantly monitors frame rate, and when it detects that the frame rate has dipped below a particular threshold so many times in so many seconds, it shows the running-slowly toast shown in Figure 5:

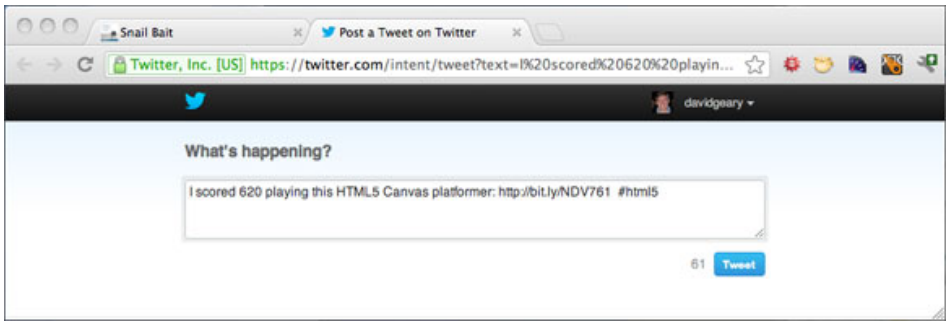
Figure 5. Slow frames-per-second detection



5. Incorporate social features

Nearly all successful games incorporate social aspects, such as posting scores on Twitter or Facebook. When a Snail Bait player clicks on the **Tweet my score** link that appears at game end (see Figure 2), Snail Bait goes to Twitter in a separate tab and automatically creates a tweet announcing the score, like the one shown in Figure 7:

Figure 7. Tweet text



Now that you have a high-level understanding of the game, it's time to take a look at some code.

Snail Bait's HTML and CSS

Snail Bait code statistics

Lines of code:

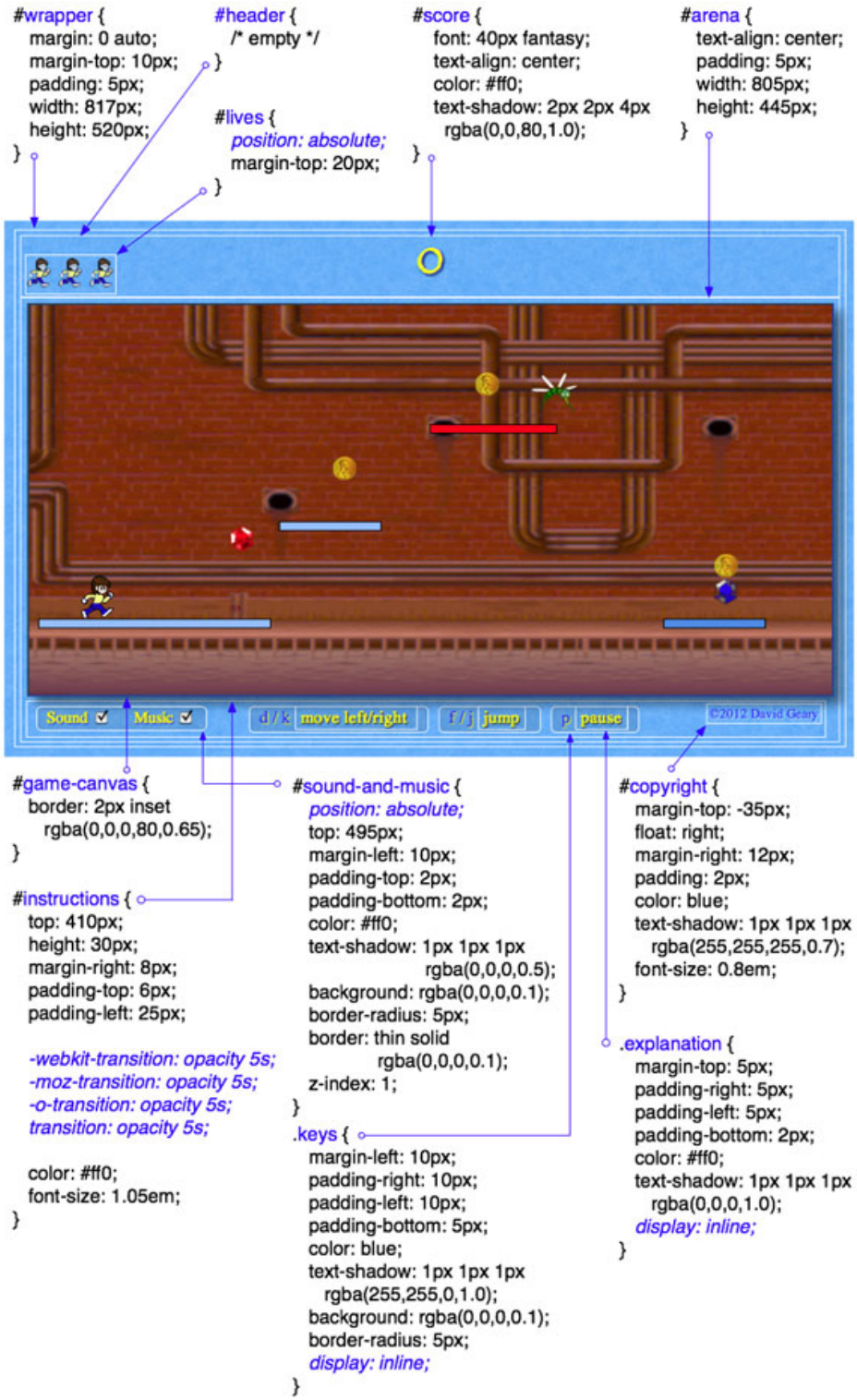
- HTML: 276

- CSS: 410
- JavaScript: 3,898

Snail Bait is implemented with HTML, CSS, and JavaScript; however, as you can see from the Snail Bait code statistics sidebar, most of the code is JavaScript. In fact, the rest of this series is primarily concerned with JavaScript, with only occasional forays into HTML and CSS3.

Figure 8 shows the HTML elements and their corresponding CSS for the game proper, omitting the HTML and CSS for other elements such as toasts and credits:

Figure 8. The game's HTML and CSS (box shadow declarations omitted)



The CSS is mostly unremarkable, with the exception of a few attributes of interest that I've highlighted in Figure 8. First, I've set the `wrapper` element's `margin` attribute to `0 auto`, which means the wrapper, and everything in it, is centered horizontally in the window. Second, the `lives` and `sound-and-music` elements have an `absolute` position. If they are left with the default position,

which is `relative`, those `DIV`s will expand to the width of the canvas, and their neighbors (score and instructions, respectively) will move beneath them. Finally, the `keys` and `explanation` CSS classes have a `display` attribute of `inline` to put the associated elements on the same row.

Listing 2 shows the CSS from Figure 8:

Listing 2. game.css (excerpt)

```
#arena {
   text-align: center;
   padding: 5px;
   width: 805px;
   height: 445px;
}

#copyright {
   margin-top: -35px;
   float: right;
   margin-right: 12px;
   padding: 2px;
   color: blue;
   text-shadow: 1px 1px 1px rgba(255,255,255,0.7);
   font-size: 0.8em;
}

.explanation {
   color: #ff0;
   text-shadow: 1px 1px 1px rgba(0,0,0,1.0);
   display: inline;
   margin-top: 5px;
   padding-right: 5px;
   padding-left: 5px;
   padding-bottom: 2px;
}

#game-canvas {
   border: 2px inset rgba(0,0,80,0.62);
   -webkit-box-shadow: rgba(0,0,0,0.5) 8px 8px 16px;
   -moz-box-shadow: rgba(0,0,0,0.5) 8px 8px 16px;
   -o-box-shadow: rgba(0,0,0,0.5) 8px 8px 16px;
   box-shadow: rgba(0,0,0,0.5) 8px 8px 16px;
}

#instructions {
   height: 30px;
   margin-right: 8px;
   padding-top: 6px;
   padding-left: 25px;

   -webkit-transition: opacity 2s;
   -moz-transition: opacity 2s;
   -o-transition: opacity 2s;
   transition: opacity 2s;

   color: #ff0;
   font-size: 1.05em;
   opacity: 1.0;
}

.keys {
   color: blue;
   text-shadow: 1px 1px 1px rgba(255,255,0,1.0);
   background: rgba(0,0,0,0.1);
   border: thin solid rgba(0,0,0,0.20);
   border-radius: 5px;
```

```
   margin-left: 10px;
   padding-right: 10px;
   padding-left: 10px;
   padding-bottom: 5px;
   display: inline;
}

#sound-and-music {
   position: absolute;
   top: 495px;
   margin-left: 10px;
   color: #ff0;
   text-shadow: 1px 1px 1px rgba(0,0,0,0.5);
   background: rgba(0,0,0,0.1);
   border-radius: 5px;
   border: thin solid rgba(0,0,0,0.20);
   padding-top: 2px;
   padding-bottom: 2px;
   z-index: 1;
}

#wrapper {
   margin: 0 auto;
   margin-top: 20px;
   padding: 5px;
   width: 817px;
   height: 520px;
}
```

As you can see from Listing 3, which lists the HTML shown in Figure 8, the game's HTML is a bunch of `DIV`s and a canvas, with a few images and a couple of check boxes:

Listing 3. game.html (excerpt)

```
<!DOCTYPE html>
<html>
   <!-- Head..................................................-->

   <head>
     <title>Snail Bait</title>
   </head>

   <!-- Body..................................................-->

   <body>
      <!-- Wrapper.............................................-->

      <div id='wrapper'>
         <!-- Header..........................................-->

         <div id='header'>
            <div id='lives'>
               <img id='life-icon-left'   src='images/runner-small.png'/>
               <img id='life-icon-middle' src='images/runner-small.png'/>
               <img id='life-icon-right'  src='images/runner-small.png'/>
            </div>

            <div id='score'>0</div>
            <div id='fps'></div>
         </div>

         <!-- Arena...........................................-->

         <div id='arena'>
            <!-- The game canvas.............................-->
```

```
            <canvas id='game-canvas' width='800' height='400'>
               Your browser does not support HTML5 Canvas.
            </canvas>

            <!-- Sound and music.....................................-->

            <div id='sound-and-music'>
               <div class='checkbox-div'>
                  Sound <input id='sound-checkbox'
                               type='checkbox' checked/>
               </div>

               <div class='checkbox-div'>
                  Music <input id='music-checkbox'
                               type='checkbox' checked/>
               </div>
            </div>

            <!-- Instructions.......................................-->

            <div id='instructions'>
               <div class='keys'>
                  d / k

                  <div class='explanation'>
                     move left/right
                  </div>
               </div>

               <div class='keys'>
                  f / j

                  <div class='explanation'>
                     jump
                  </div>
               </div>

               <div class='keys'>
                  p

                  <div class='explanation'>
                     pause

                  </div>
               </div>
            </div>

            <!-- Copyright..........................................-->

            <div id='copyright'> ©2012 David Geary</div>
         </div>
      </div>

      <!-- JavaScript.............................................-->

      <script src='js/stopwatch.js'></script>
      <script src='js/animationTimer.js'></script>
      <script src='js/sprites.js'></script>
      <script src='js/requestNextAnimationFrame.js'></script>
      <script src='js/behaviors/bounce.js'></script>
      <script src='js/behaviors/cycle.js'></script>
      <script src='js/behaviors/pulse.js'></script>
      <script src='game.js'></script>
  </body>
</html>
```

The `canvas` element is where all the action takes place. That canvas comes with a 2D context with a powerful API for implementing 2D games, among other things. The text inside the `canvas` element is fallback text that the browser displays only if it does not support HTML5 Canvas.

Draw into a small canvas and let CSS scale it?

Some games purposely draw into a small canvas and use CSS to scale the canvas to a playable size. That way, you're not manipulating as many canvas pixels, which increases performance. And typically, scaling a canvas with CSS is hardware-accelerated, so the cost of the scaling can be minimal. Today, however — because nearly all of the latest versions of modern browsers come equipped with hardware-accelerated Canvas — in most cases, it's just as fast to draw into a full-sized canvas.

One final note about the game's HTML and CSS: Notice that the width and height of the canvas is specified with the `canvas` element's `width` and `height` attributes. Those attributes pertain to both the size of the `canvas` element *and* the size of the drawing surface contained within that element.

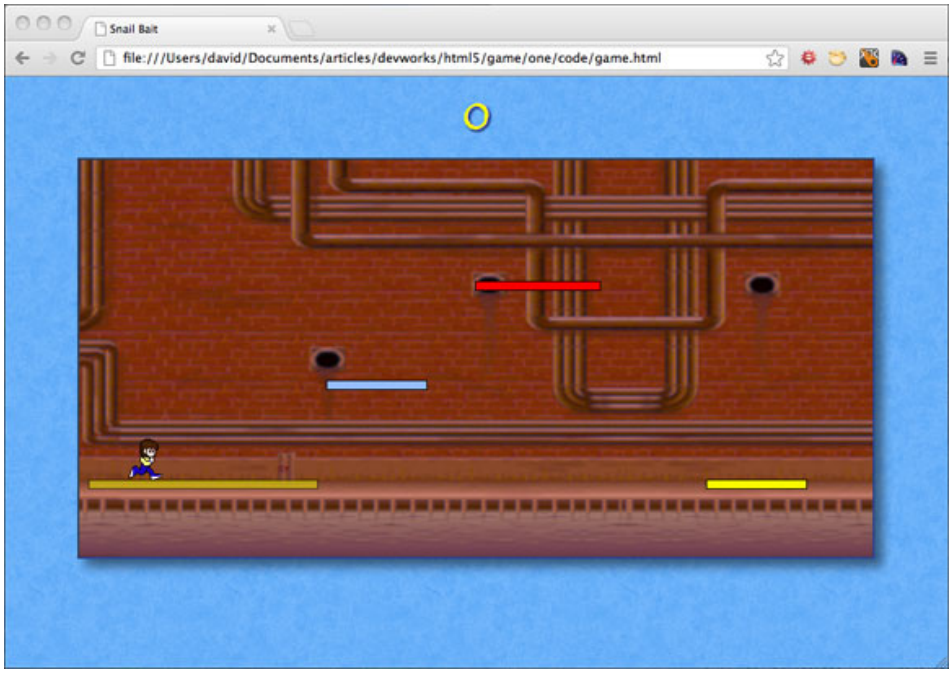
On the other hand, using CSS to set the width and height of the `canvas` element *only sets the size of the element*. The drawing surface remains at its default width and height of 300 and 150 pixels, respectively. That means you will most likely have a mismatch between the `canvas` element size and the size of its drawing surface, and when that happens the browser scales the drawing surface to fit the element. Most of the time that is an unwanted effect, so it's a good idea never to set the size of the `canvas` element with CSS.

As with fine movies such as *Pulp Fiction*, you've already seen the end of the story. Now I'll go back to the beginning.

Snail Bait's humble beginning

Figure 9 shows the starting point for the game, which simply draws the background, platforms, and the runner. To start, the platforms and runner are not sprites; instead, the game draws them directly. See Download to get the code that creates the background and runner.

Figure 9. Drawing the background and runner



Listing 3 lists the starting point for the game's HTML, which is just a scaled-down version of the HTML in Listing 2:

Listing 3. game.html (starter version)

```
<!DOCTYPE html>
<html>
   <!-- Head..................................................-->

   <head>
      <title>Snail Bait</title>
      <link rel='stylesheet' href='game.css'/>
   </head>

   <!-- Body..................................................-->

   <body>
      <!-- Wrapper..............................................-->

      <div id='wrapper'>
         <!-- Header..............................................-->

         <div id='header'>
            <div id='score'>0</div>
         </div>

         <!-- Arena...............................................-->

         <div id='arena'>
            <!-- The game canvas.....................................-->

            <canvas id='game-canvas' width='800' height='400'>
               Your browser does not support HTML5 Canvas.
            </canvas>
         </div>
      </div>

      <!-- JavaScript..........................................-->
```

```
      <script src='game.js'></script>
  </body>
</html>
```

Listing 4 shows the JavaScript:

Listing 4. game.js (starter version)

```
// --------------------------- DECLARATIONS ---------------------------

var canvas = document.getElementById('game-canvas'),
    context = canvas.getContext('2d'),

   // Constants.....................................................

   PLATFORM_HEIGHT = 8,
   PLATFORM_STROKE_WIDTH = 2,
   PLATFORM_STROKE_STYLE = 'rgb(0,0,0)',

   STARTING_RUNNER_LEFT = 50,
   STARTING_RUNNER_TRACK = 1,

   // Track baselines
   //
   // Platforms move along tracks. The constants that follow define
   // the Y coordinate (from the top of the canvas) for each track.

   TRACK_1_BASELINE = 323,
   TRACK_2_BASELINE = 223,
   TRACK_3_BASELINE = 123,

   // Images

   background  = new Image(),
   runnerImage = new Image(),

   // Platforms
   //
   // Each platform has its own fill style, but the stroke style is
   // the same for each platform.

   platformData = [  // One screen for now
      // Screen 1.....................................................
      {
         left:      10,
         width:     230,
         height:    PLATFORM_HEIGHT,
         fillStyle: 'rgb(255,255,0)',
         opacity:   0.5,
         track:     1,
         pulsate:   false,
      },

      {  left:      250,
         width:     100,
         height:    PLATFORM_HEIGHT,
         fillStyle: 'rgb(150,190,255)',
         opacity:   1.0,
         track:     2,
         pulsate:   false,
      },

      {  left:      400,
         width:     125,
         height:    PLATFORM_HEIGHT,
```

```
         fillStyle: 'rgb(250,0,0)',
         opacity:   1.0,
         track:     3,
         pulsate:   false
      },

      {  left:      633,
         width:     100,
         height:    PLATFORM_HEIGHT,
         fillStyle: 'rgb(255,255,0)',
         opacity:   1.0,
         track:     1,
         pulsate:   false,
      },
   ];

// ------------------------ INITIALIZATION ----------------------------

function initializeImages() {
   background.src = 'images/background_level_one_dark_red.png';
   runnerImage.src = 'images/runner.png';

   background.onload = function (e) {
      startGame();
   };
}

function drawBackground() {
   context.drawImage(background, 0, 0);
}

function calculatePlatformTop(track) {
   var top;

   if      (track === 1) { top = TRACK_1_BASELINE; }
   else if (track === 2) { top = TRACK_2_BASELINE; }
   else if (track === 3) { top = TRACK_3_BASELINE; }

   return top;
}

function drawPlatforms() {
   var pd, top;

   context.save(); // Save context attributes on a stack

   for (var i=0; i < platformData.length; ++i) {
      pd = platformData[i];
      top = calculatePlatformTop(pd.track);

      context.lineWidth = PLATFORM_STROKE_WIDTH;
      context.strokeStyle = PLATFORM_STROKE_STYLE;
      context.fillStyle = pd.fillStyle;
      context.globalAlpha = pd.opacity;

      // If you switch the order of the following two
      // calls, the stroke will appear thicker.

      context.strokeRect(pd.left, top, pd.width, pd.height);
      context.fillRect  (pd.left, top, pd.width, pd.height);
   }

   context.restore(); // Restore context attributes
}

function drawRunner() {
   context.drawImage(runnerImage,
```

```
      STARTING_RUNNER_LEFT,
      calculatePlatformTop(STARTING_RUNNER_TRACK) - runnerImage.height);
}

function draw(now) {
   drawBackground();
   drawPlatforms();
   drawRunner();
}

function startGame() {
   draw();
}

// Launch game

initializeImages();
```

The JavaScript accesses the `canvas` element and subsequently obtains a reference to the canvas's 2D context. The code then uses the context's `drawImage()` method to draw the background and runner images. In this case, I'm using the three-argument variant of `drawImage()` to draw images at a particular `(x,y)` destination in the canvas.

The `drawPlatforms()` function draws the platforms by stroking and filling rectangles after setting the context's line width, stroke style, fill style, and global alpha attribute. Notice the calls to `context.save()` and `context.restore()`: the attribute settings between those calls are temporary. I will discuss those methods in the next article in this series.

The game starts when the background image loads. For now, starting entails simply drawing the background, sprites, and runner. The next challenge is to bring those static images to life.

Next time

In the next article in this series, I'll start with an overview of the canvas context's 2D API, and then discuss animation and set things in motion by scrolling the background. You will see how to implement parallax to make the platforms appear closer than the background, and you'll see how to make sure that your sprites animate at a constant rate regardless of your animation's frame rate. See you next time.

Downloadable resources

Description	Name	Size
Code for Snail Bait's background and runner	j-html5-game1.zip	718KB

Related topics

- *Core HTML5 Canvas*: (David Geary, Prentice Hall, 2012): David Geary's book offers extensive coverage of the Canvas API and game development. Also check out the companion website and blog.
- Mind-blowing apps with HTML5 Canvas: Watch David Geary's presentation from Strange Loop 2011.
- HTML5 Game Development: Watch David Geary's presentation from the Norwegian Developer's Conference (NDC) 2011.
- Platform games: Read about platform games at Wikipedia.
- Side-scroller video games: Read about side-scroller video games at Wikipedia.
- Strategy pattern: Check out Wikipedia's article on the Strategy design pattern.
- *Pulp Fiction*: Read about the movie — a prime example of nonlinear narrative — on Wikipedia.
- Replica Island: You can download the source for this popular open source platform video game for Android.