HTML5 2D game development: Collision detection and sprite animations

Detecting and reacting to collisions; making sprites explode

David Geary
Author and speaker
Clarity Training, Inc.

Skill Level: Intermediate

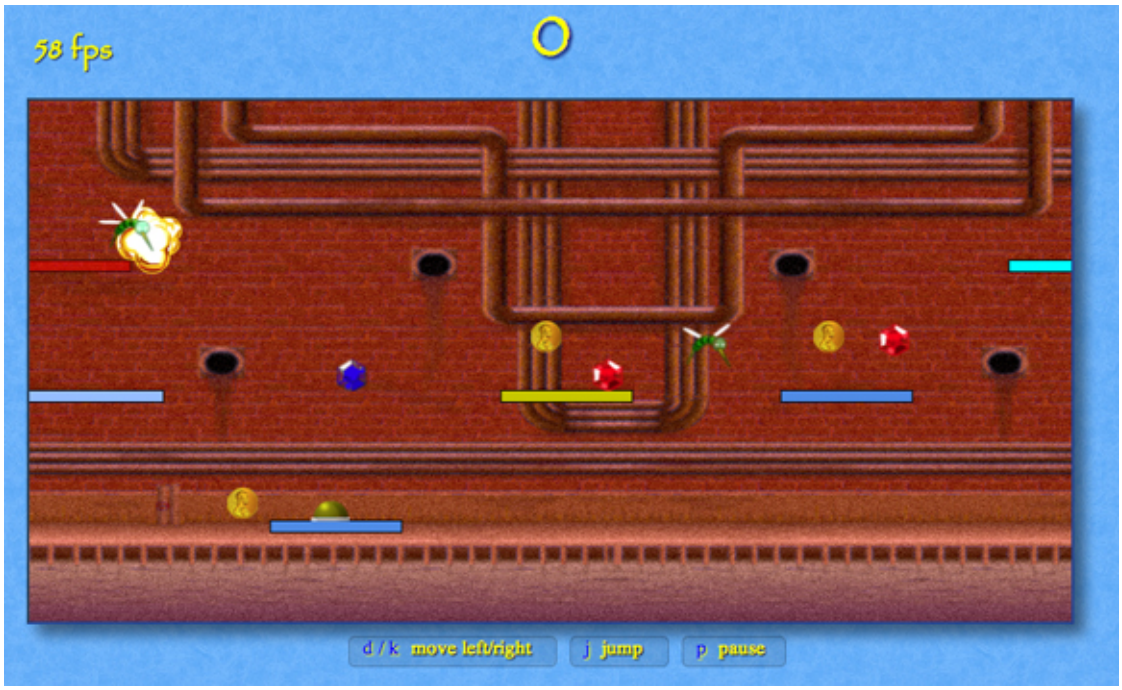
Date: 16 Apr 2013

In this series, HTML5 maven David Geary shows you how to implement an HTML5 2D video game one step at a time. In this installment, learn how Snail Bait implements collision detection and explosions.

View more content in this series

Collision detection and sprite animations are staples of all video games. Snail Bait, the game you're building in this series, is no exception. Figure 1 shows Snail Bait's runner exploding after colliding with the bee in the upper left corner.

Figure 1. Collision detection in action



In this article, learn how to:

- Detect collisions
- Use the HTML5 Canvas context for collision detection
- Implement collision detection as sprite behaviors
- Process collisions
- Implement sprite animations, such as explosions

The collision-detection process

Collision detection is a four-step process, one step being the actual detection of collisions:

1. Iterate over the game's sprites
2. Disqualify sprites that are not candidates for collision detection
3. Detect collisions between candidate sprites
4. Process collisions

Collision detection can be computationally expensive, so it's essential to avoid it for sprites that cannot possibly collide. For example, Snail Bait's runner runs through other sprites when they are exploding. Because it takes less time to check whether a sprite is exploding than it does to perform collision detection, Snail Bait disqualifies exploding sprites from collision detection.

Let's start with an overview of collision-detection techniques.

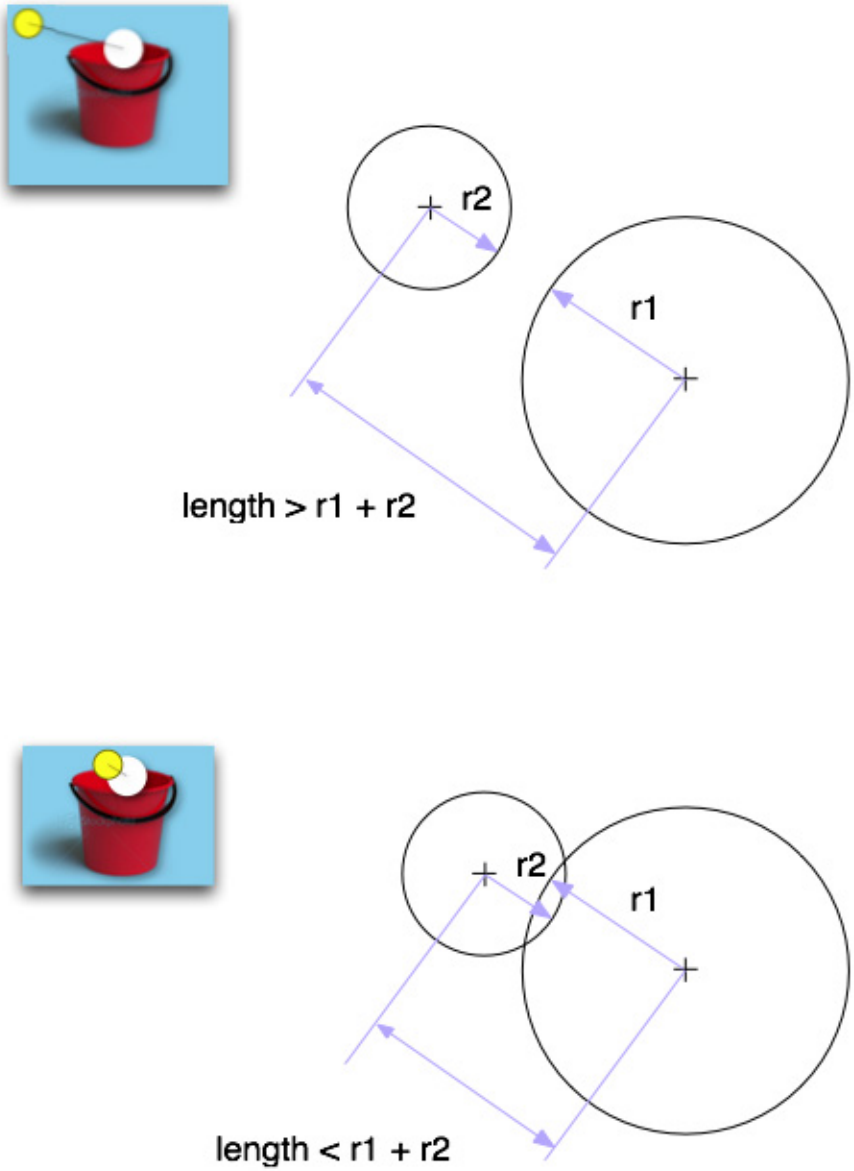
Collision-detection techniques

You can detect collisions between sprites in several ways. Three popular techniques, in order of increasing sophistication and complexity, are:

- Bounding areas (bounding volumes for 3D games)
- Ray casting
- The Separating Axis Theorem

Collision detection with bounding areas detects intersections between circles or polygons. In the example in Figure 2, the smaller circle is the bounding area that represents one sprite (a small ball), and the larger circle is the bounding area for a bucket sprite that's larger than the ball. When the two circular bounding areas intersect, the ball lands in the bucket.

Figure 2. Bounding areas: Collisions between circles



Detecting collisions between two circles is the simplest of all collision-detection techniques. If the distance between the circles' centers is less than the circles' combined radii, the circles intersect and the sprites have collided.

Bounding-area collision detection is simple, but it can fail when bounding areas are too small or are moving too fast. In either case, sprites can pass by each other in a single animation frame, thereby avoiding detection.

A more reliable technique for small, fast-moving sprites is ray casting, illustrated in Figure 3. Ray casting detects the intersection of two sprites' velocity vectors. In each of the five frames in Figure 3, the ball's velocity vector is the diagonal line drawn in blue, and the velocity vector for the bucket is the red horizontal line. (The bucket moves horizontally.) The ball lands in the bucket when the intersection of those vectors lies within the opening at the top of the bucket and the ball is below the opening, as illustrated in the right-most screenshot in Figure 3.

Figure 3. Ray casting

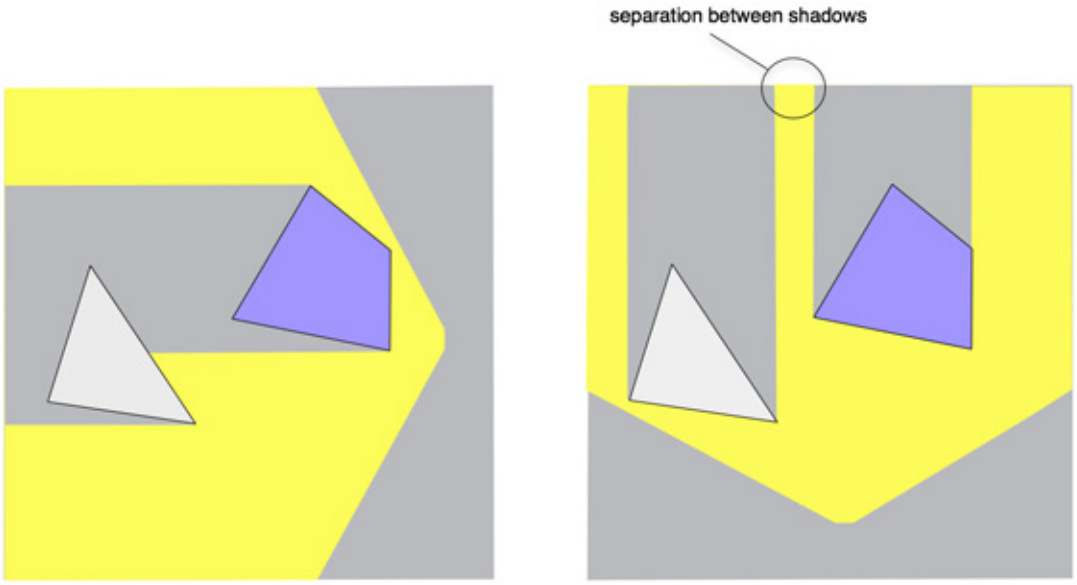
A priori or a posteriori collision detection

You can detect collisions before they occur (a priori) or after (a posteriori). If you detect collisions before they occur, you must predict where sprites will be in the future. If you detect collisions after they occur, you will typically need to separate the sprites that have collided. Neither approach is noticeably better or simpler than the other.

Ray casting works well for simple shapes under circumstances — such as the ball landing in the bucket in Figure 2 — where it's easy to determine if two shapes have collided, given the intersection of their velocity vectors.

For more complicated scenarios, such as collisions between polygons of arbitrary size and shape, the Separating Axis Theorem is one of the most reliable — and most complicated — collision-detection techniques. The Separating Axis Theorem is the mathematical equivalent of shining a light on two polygons from different angles, as shown in Figure 4. If the shadow on the wall behind the polygons reveals a gap, the polygons are not colliding.

Figure 4. The Separating Axis Theorem

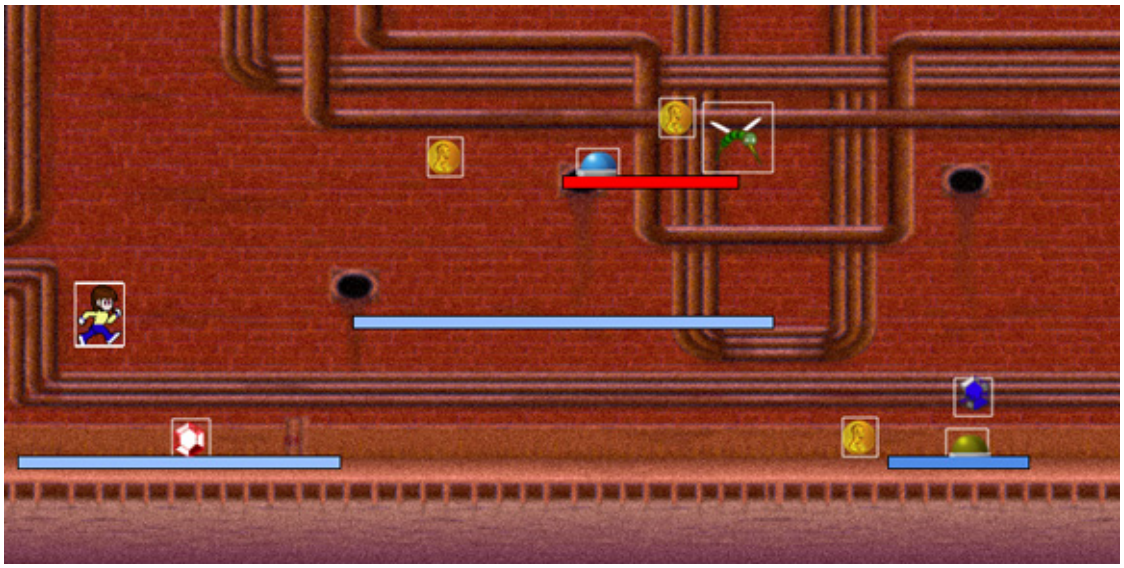


This article doesn't cover ray casting or the Separating Axis Theorem any further. You can read in-depth discussions of each approach in Core HTML5 Canvas (Prentice Hall, 2012). (See Resources for a link.)

Snail Bait's collision detection

Snail Bait's collision detection involves relatively large sprites moving at slow speeds, so the game detects collisions with bounding boxes. Those bounding boxes are shown in Figure 5.

Figure 5. Snail Bait's collision-detection bounding boxes



Snail Bait implements sprite activities, such as running, jumping, and exploding, as sprite behaviors — see the article "Implementing sprite behaviors" (developerWorks, January 2013) for more information. And the same is true for collision detection. At this point in Snail

Bait's development, the runner has three behaviors: She can run, jump, and collide with other sprites. Listing 1 shows the instantiation of the runner sprite with these three behaviors.

Listing 1. The runner's behaviors

```
Sprite = function () {
  ...
  this.runner = new Sprite('runner',           // type
                           this.runnerArtist,  // artist
                           [ this.runBehavior, // behaviors
                             this.jumpBehavior,
                             this.collideBehavior
                           ]);
};
```

Listing 2 shows the code for the runner's `collideBehavior`.

Listing 2. The runner's collide behavior

```
var SnailBait =  function () {
  ...

  // Runner's collide behavior.....................................................

  this.collideBehavior = {
    execute: function (sprite, time, fps, context) {  // sprite is the runner
      var otherSprite;

      for (var i=0; i < snailBait.sprites.length; ++i) {
        otherSprite = snailBait.sprites[i];

        if (this.isCandidateForCollision(sprite, otherSprite)) {
          if (this.didCollide(sprite, otherSprite, context)) {
            this.processCollision(sprite, otherSprite);
          }
        }
      }
    },
    ...

  };
};
```

Because the `collideBehavior` object is a sprite behavior, Snail Bait invokes its `execute()` method for every animation frame. And because the `collideBehavior` object is associated with the runner, the sprite that Snail Bait passes to the `execute()` method is always the runner. See the Behavior fundamentals section in "Implementing sprite behaviors" article for more information about sprite behaviors.

The `collideBehavior` object's `execute()` method encapsulates the four collision-detection steps listed earlier. The last three steps are represented by the following `collideBehavior` methods:

- `isCandidateForCollision(sprite, otherSprite)`
- `didCollide(sprite, otherSprite, context)`
- `processCollision(sprite, otherSprite)`

The implementations of each of these methods are discussed in the following sections.

Selecting candidates for collision detection

A Snail Bait sprite is eligible to collide with the runner sprite when:

- The sprite is not the runner
- Both that sprite and the runner are visible
- Neither that sprite nor the runner is exploding

The collideBehavior object's isCandidateForCollision() method, shown in Listing 3, implements this logic.

Listing 3. Selecting candidates for collision detection: isCandidateForCollision()

```
var SnailBait =  function () {
  ...

  isCandidateForCollision: function (sprite, otherSprite) {
    return sprite !== otherSprite &&                    // not same
           sprite.visible && otherSprite.visible &&     // visible
           !sprite.exploding && !otherSprite.exploding;    // not exploding
  },
  ...
};
```

Next, see how to detect collisions between qualifying sprites.

Detecting collisions between the runner and another sprite

The collideBehavior object's didCollide() method, which determines whether the runner has collided with another sprite, is shown in Listing 4.

Listing 4. Checking for collisions: didCollide()

```
var SnailBait =  function () {
  ...

  didCollide: function (sprite,      // runner
                        otherSprite, // candidate for collision
                        context) {   // for context.isPointInPath()
    var left = sprite.left + sprite.offset,
        right = sprite.left + sprite.offset + sprite.width,
        top = sprite.top,
        bottom = sprite.top + sprite.height,
        centerX = left + sprite.width/2,
        centerY = sprite.top + sprite.height/2;

    // All the preceding variables -- left, right, etc. -- pertain to the runner sprite.

    if (otherSprite.type !== 'snail bomb') {
      return this.didRunnerCollideWithOtherSprite(left, top, right, bottom,
                                                  centerX, centerY,
                                                  otherSprite, context);
    }
    else {
      return this.didSnailBombCollideWithRunner(left, top, right, bottom,
                                                otherSprite, context);
    }
  },
```

The didCollide() method calculates the runner's bounding box and its center, both of which it subsequently passes to one of two methods, depending on the other sprite's identity.

If the other sprite is not the snail bomb, didCollide() invokes didRunnerCollideWithOtherSprite(), shown in Listing 5:.

Listing 5. didRunnerCollideWithOtherSprite()

```
didRunnerCollideWithOtherSprite: function (left, top, right, bottom,
                                  centerX, centerY,
                                  otherSprite, context) {
  // Determine if either of the runner's four corners or its
  // center lies within the other sprite's bounding box.

  context.beginPath();
  context.rect(otherSprite.left - otherSprite.offset, otherSprite.top,
               otherSprite.width, otherSprite.height);

  return context.isPointInPath(left,   top)    ||
         context.isPointInPath(right,  top)    ||

         context.isPointInPath(centerX, centerY) ||

         context.isPointInPath(left,   bottom) ||
         context.isPointInPath(right,  bottom);
},
```

Given the runner's bounding box and the coordinates of its center, didRunnerCollideWithOtherSprite() checks to see whether one of the bounding box's corners or its center lies within the other sprite's bounding box.

The Canvas context does more than graphics

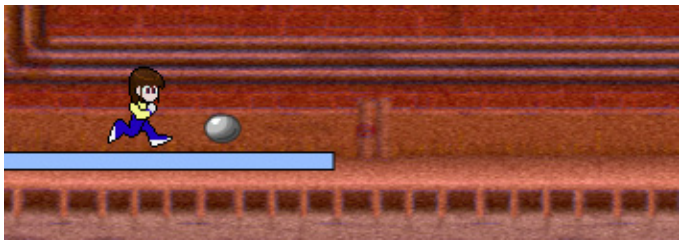
The Canvas context's isPointInPath() method detects whether a point is in the current path. Snail Bait uses it to determine whether a point lies within a rectangle. But isPointInPath() really shines when the path is an irregular shape. Determining whether a point lies within an irregular shape is difficult to calculate by hand.

Determining whether a point lies within a rectangle does not require a great deal of mathematical acuity; however, the HTML5 canvas element's 2D context makes it even easier with the isPointInPath() method, which returns true if a point lies within the canvas context's current path.

The didRunnerCollideWithOtherSprite() method creates a rectangular path representing the other sprite's bounding box with calls to beginPath() and rect(). The didRunnerCollideWithOtherSprite() method subsequently calls isPointInPath() to determine if one of the five points within the runner lies within the other sprite's bounding box.

The didRunnerCollideWithOtherSprite() method correctly identifies collisions between the runner and all other sprites, except for the snail bomb, as shown in Figure 6.

Figure 6. The runner and the snail bomb



It doesn't work correctly for the snail bomb because the snail bomb happens to be just small enough that it can pass through the runner without any of the corners of the runner's bounding box or its center coming in contact with the bomb. Because of that unfortunate ratio of runner size to bomb size, the `didCollide()` method in Listing 4 invokes `didSnailBombCollideWithRunner()`, shown in Listing 6, when the other sprite is the bomb.

Listing 6. The `didSnailBombCollideWithRunner()` method

```
didSnailBombCollideWithRunner: function (left, top, right, bottom,
                                         snailBomb, context) {
  // Determine if the center of the snail bomb lies within
  // the runner's bounding box

  context.beginPath();
  context.rect(left, top, right - left, bottom - top); // runner's bounding box

  return context.isPointInPath(
            snailBomb.left - snailBomb.offset + snailBomb.width/2,
            snailBomb.top + snailBomb.height/2);
},
```

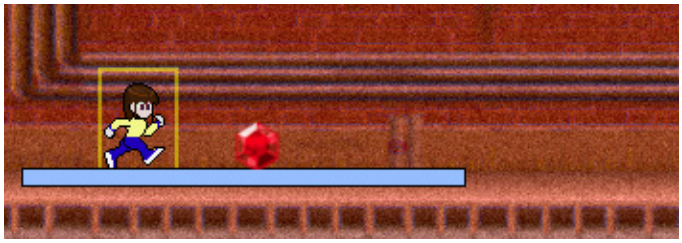
The `didSnailBombCollideWithRunner()` method is the inverse of `didRunnerCollideWithOtherSprite()`: `didRunnerCollideWithOtherSprite()` checks to see if points in the runner lie within another sprite, whereas `didSnailBombCollideWithRunner()` checks to see if the middle of the other sprite (the bomb) lies within the runner.

You've seen how to implement collision detection with bounding boxes, but the technique can be more accurate and perform better. In the sections that follow, see how to refine Snail Bait's collision detection by modifying the runner's bounding box and partitioning the game's space.

Refining bounding boxes

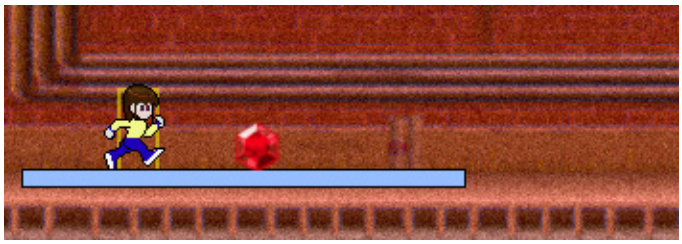
As you can see from Figure 5, a collision-detection bounding box encloses the sprite that it represents. Near the corners of those bounding boxes, however, the interior of the box is often transparent. This is the case for the runner sprite, as Figure 7 illustrates. Those transparent areas can result in false collisions, which is especially noticeable when two transparent areas collide.

Figure 7. The runner's original bounding box



One way to eliminate false collisions resulting from transparent corner areas is to reduce the size of the sprite's bounding box, as shown in Figure 8.

Figure 8. The runner's revised bounding box



Snail Bait reduces the size of the runner's bounding box with the revised `didCollide()` method shown in Listing 7.

Listing 7. Modifying the runner's bounding box

```
var SnailBait =  function () {
...

didCollide: function (sprite,      // runner
                      otherSprite, // candidate for collision
                      context) {   // for context.isPointInPath()
    var MARGIN_TOP = 10,
        MARGIN_LEFT = 10,
        MARGIN_RIGHT = 10,
        MARGIN_BOTTOM = 0,
        left = sprite.left + sprite.offset + MARGIN_LEFT,
        right = sprite.left + sprite.offset + sprite.width - MARGIN_RIGHT,
        top = sprite.top + MARGIN_TOP,
        bottom = sprite.top + sprite.height - MARGIN_BOTTOM,
        centerX = left + sprite.width/2,
        centerY = sprite.top + sprite.height/2;
      ...
  },
  ...
};
```

Reducing the runner's bounding box makes Snail Bait's collision detection more accurate because it eliminates false collisions. Next, see how to make collision detection perform better.

Spatial partitioning

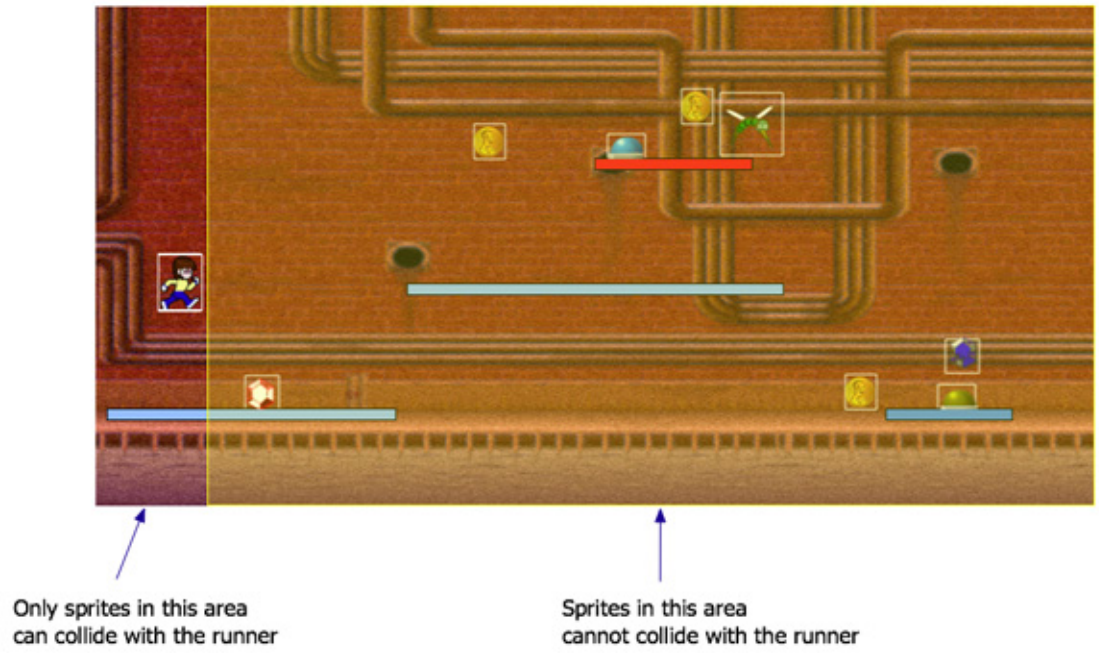
More on spatial partitioning

Snail Bait's spatial partitioning is as primitive as spatial partitioning gets. More complicated implementations of spatial partitioning are octrees and binary space

partitioning, which are appropriate when you have lots of collision-detection cells. See Resources for more information about spatial partitioning.

Spatial partitioning involves partitioning a game's space into cells, so that only sprites in the same cell can possibly collide. By eliminating collision detection for sprites that reside in different cells, spatial partitioning often results in substantial performance increases. Snail Bait gets just such a performance increase by partitioning space as shown in Figure 9.

Figure 9. Snail Bait's spatial partitioning



As Listing 8 shows, Snail Bait disqualifies all sprites in the right-hand region in Figure 9 from collision detection, which greatly reduces the amount of collision detection the game performs.

Listing 8. Refining the selection of sprites for collision detection

```
this.isCandidateForCollision: function (sprite, otherSprite) {
  return sprite !== otherSprite &&
        sprite.visible && otherSprite.visible &&
        !sprite.exploding && !otherSprite.exploding &&
        otherSprite.left - otherSprite.offset < sprite.left + sprite.width;


},
```

Now that you've seen how to efficiently detect collisions, let's take a look at how Snail Bait processes collisions.

Processing collisions

Once you detect a collision, you have to do something about it. Snail Bait's `processCollision()` processes collisions between the runner and other sprites, as you can see from Listing 9.

Listing 9. Processing collisions: processCollision()

```
var SnailBait =  function () {
  processCollision: function (sprite, otherSprite) {
    if ('coin'  === otherSprite.type    ||  // bad guys
        'sapphire' === otherSprite.type ||
        'ruby' === otherSprite.type     ||
        'button' === otherSprite.type   ||
        'snail bomb' === otherSprite.type) {
      otherSprite.visible = false;
    }

    if ('bat' === otherSprite.type   ||  // good guys
        'bee' === otherSprite.type   ||
        'snail' === otherSprite.type ||
        'snail bomb' === otherSprite.type) {
      snailBait.explode(sprite);
    }

    if (sprite.jumping && 'platform' === otherSprite.type) {
      this.processPlatformCollisionDuringJump(sprite, otherSprite);
    }
  },
  ...
};
```

When the runner collides with good guys — coins, sapphires, rubies, and buttons — or with the snail bomb, Snail Bait makes the other sprite invisible by setting its `visible` attribute to `false`.

When the runner collides with bad guys — bats, bees, snails, and the snail bomb — `processCollision()` explodes the runner with Snail Bait's `explode()` method. At present, the `explode()` method simply prints `BOOM` to the console. The next article in this series will discuss the `explode()` method's final implementation.

Finally, the `processPlatformCollisionDuringJump()` method, shown in Listing 10, processes platform collisions while the runner is jumping.

Listing 10. processPlatformCollisionDuringJump()

```
processPlatformCollisionDuringJump: function (sprite, platform) {
    var isDescending = sprite.descendAnimationTimer.isRunning();

    sprite.stopJumping();

    if (isDescending) { // Collided with platform while descending
      // land on platform

      sprite.track = platform.track;
      sprite.top = snailBait.calculatePlatformTop(sprite.track) - sprite.height;
    }
    else { // Collided with platform while ascending
      sprite.fall();
    }
  }
};
```

When the runner collides with a platform during a jump, if she was descending during the jump, she stops jumping and lands on the platform. If the runner was ascending, she has collided with the platform from underneath, so she falls. For now, the runner's `fall()` method is implemented as shown in Listing 11.

Listing 11. Stopgap method for falling

```
var SnailBait =  function () {

  ...


  this.runner.fall = function () {
    snailBait.runner.track = 1;
    snailBait.runner.top = snailBait.calculatePlatformTop(snailBait.runner.track) -
                           snailBait.runner.height;
  };
  ...
};
```

The runner's `fall()` method immediately places the runner on the bottom track, meaning the top of the lowest platform. In the next article in this series, you'll see how to reimplement that method with realistic falling by incorporating gravity over time.

Monitoring collision-detection performance

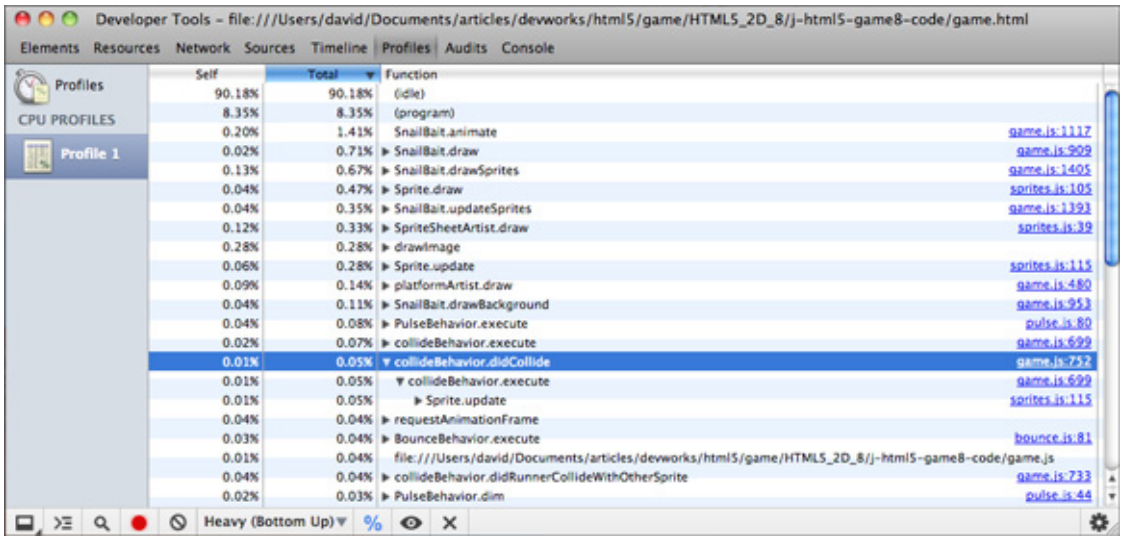
Ninety percent idle?

The top entry in the table displayed in Figure 10 means that Snail Bait is just waiting around for something to do 90 percent of the time. Snail Bait has such remarkable performance because the profile in Figure 10 was taken in the Chrome browser (version 26) — which like all modern browsers — hardware-accelerates the canvas element. Browser vendors typically implement acceleration for the canvas element by translating calls to the Canvas API into WebGL, so you get the convenience of the Canvas API with the performance of WebGL.

Collision detection can easily be a performance bottleneck, especially for more mathematically intense collision-detection algorithms such as the Separating Axis Theorem. This article has shown you some simple techniques that you can use to improve performance, such as refining bounding boxes and spatial partitioning. But it's also a good idea to constantly monitor your game's performance so you can catch and fix performance problems soon after they appear.

All modern browsers come with sophisticated development environments; for example, Chrome, Safari, Firefox, Internet Explorer, and Opera all let you profile your code as it runs. Figure 10 shows Chrome's profiler, which depicts how much time, relative to the total, you spend in individual methods.

Figure 10. Collision-detection performance



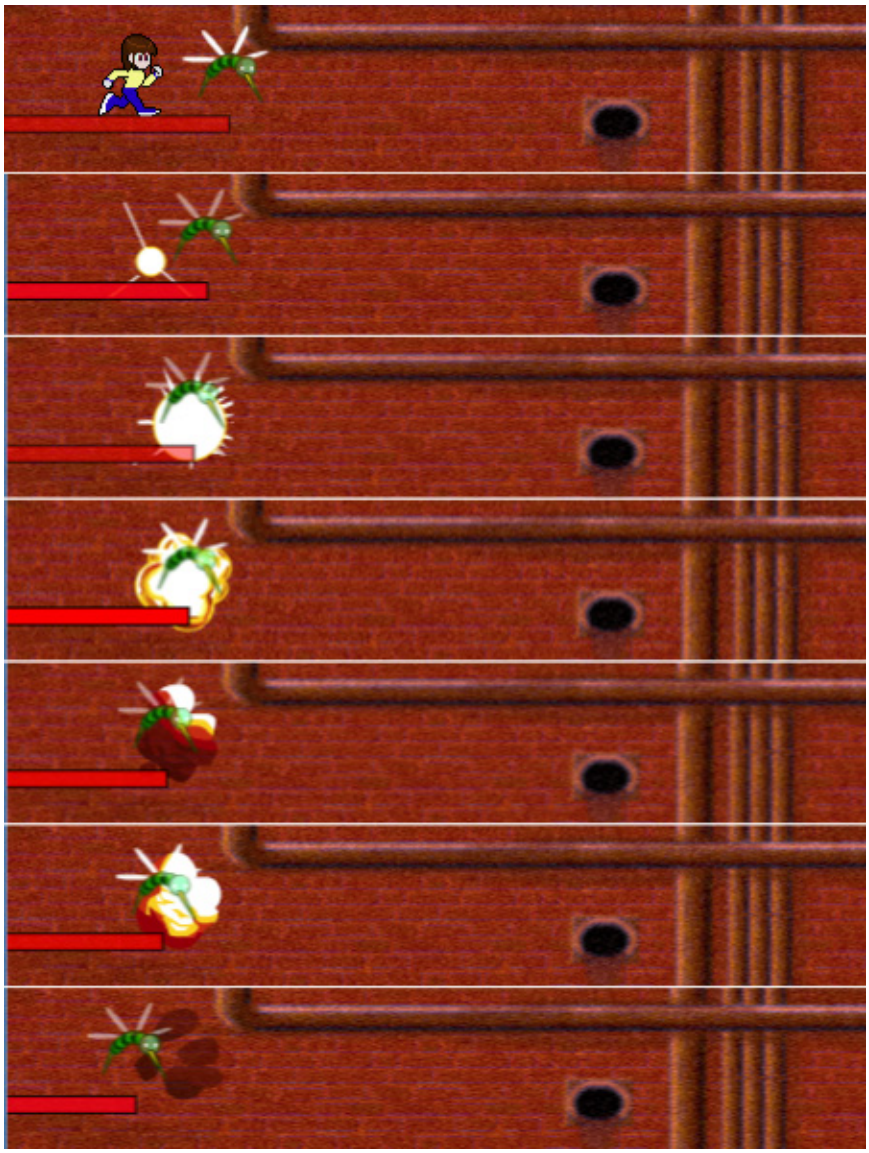
You can see in Figure 10 that Snail Bait's `didCollide()` method takes only 0.05 percent of the game's time. (The Self column, whose value for `didCollide()` is 0.01 percent, represents only the time spent directly in a method, not including time spent in methods that the method calls.)

When the runner collides with bad guys, she explodes. Let's take a look at how to implement that explosion next.

Sprite animations

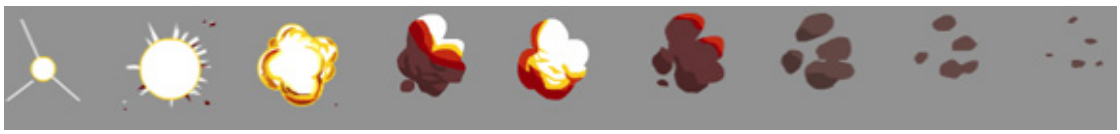
Figure 11 illustrates, from top to bottom, the explosion animation that Snail Bait displays when the runner runs into a bad guy, such as a bee.

Figure 11. The runner exploding after a collision



Snail Bait implements sprite animations, such as the one shown in Figure 11, with sprite animators. A sprite animator temporarily changes the cells that a sprite's artist draws for a specified duration. For example, the explosion sprite animator changes the runner's animation cells to those shown in Figure 12 for 500 milliseconds.

Figure 12. Explosion cells from Snail Bait's spritesheet



The constructor for sprite animator objects is shown in Listing 12.

Listing 12. Sprite animator constructor

```
// Sprite Animators.....................................................

var SpriteAnimator = function (cells, duration, callback) {
  this.cells = cells;
  this.duration = duration || 1000;
  this.callback = callback;
};
```

The SpriteAnimator constructor takes three arguments. The first argument is an array of bounding boxes in Snail Bait's spritesheet; those are the temporary animation cells, and the argument is mandatory. The second and third arguments are optional. The second is the duration of the animation, the third is a callback function that is invoked by the sprite animator when the animation duration has been exceeded.

SpriteAnimator methods are defined in the object's prototype, as shown in Listing 13.

Listing 13. Sprite animator methods

```
SpriteAnimator.prototype = {
  start: function (sprite, reappear) {
    var originalCells = sprite.artist.cells,
        originalIndex = sprite.artist.cellIndex,
        self = this;

    sprite.artist.cells = this.cells;
    sprite.artist.cellIndex = 0;

    setTimeout(function() {
      sprite.artist.cells = originalCells;
      sprite.artist.cellIndex = originalIndex;

      sprite.visible = reappear;

      if (self.callback) {
        self.callback(sprite, self);
      }
    }, self.duration);
  },
};
```

The SpriteAnimator's start() method starts the animation by saving the original animation cells and the index pointing to the current cell, and replacing them with the temporary cells and zero, respectively. Subsequently, after the animation's duration is over, the start() method reverts the sprite's animation cells and the original index to what they were before the animation.

Listing 14 shows how Snail Bait uses a sprite animator to make the runner explode.

Listing 14. Creating the explosion animator

```
var SnailBait =  function () {
  this.canvas = document.getElementById('game-canvas'),
  this.context = this.canvas.getContext('2d'),
  ...

  this.RUN_ANIMATION_RATE = 17,     // frames/second
  this.EXPLOSION_CELLS_HEIGHT = 62, // pixels
```

```
  this.EXPLOSION_DURATION = 500,    // milliseconds

  this.explosionCells = [
    { left: 1,   top: 48, width: 50, height: this.EXPLOSION_CELLS_HEIGHT },
    { left: 60,  top: 48, width: 68, height: this.EXPLOSION_CELLS_HEIGHT },
    { left: 143, top: 48, width: 68, height: this.EXPLOSION_CELLS_HEIGHT },
    { left: 230, top: 48, width: 68, height: this.EXPLOSION_CELLS_HEIGHT },
    { left: 305, top: 48, width: 68, height: this.EXPLOSION_CELLS_HEIGHT },
    { left: 389, top: 48, width: 68, height: this.EXPLOSION_CELLS_HEIGHT },
    { left: 470, top: 48, width: 68, height: this.EXPLOSION_CELLS_HEIGHT }
  ],
  ...

  this.explosionAnimator = new SpriteAnimator(
    this.explosionCells,          // Animation cells
    this.EXPLOSION_DURATION,      // Duration of the explosion

    function (sprite, animator) { // Callback after animation
      sprite.exploding = false;

      if (sprite.jumping) {
        sprite.stopJumping();
      }
      else if (sprite.falling) {
        sprite.stopFalling();
      }

      sprite.visible = true;
      sprite.track = 1;
      sprite.top = snailBait.calculatePlatformTop(sprite.track) - sprite.height;
      sprite.artist.cellIndex = 0;
      sprite.runAnimationRate = snailBait.RUN_ANIMATION_RATE;
    }
  );
};
```

Snail Bait creates a sprite animator with the animation cells shown in Figure 12. The animation's duration is 500ms, and when it's over, the sprite animator invokes the explosion animator's callback function, which places the runner on the lowest platform track. Ultimately, in a subsequent article, we'll reimplement that callback function to lose a life and restart the current level.

Listing 15 shows the runner's (somewhat anticlimactic) `explode()` method.

Listing 15. Snail Bait's `explode()` method

```
SnailBait.prototype = {
  ...

  explode: function (sprite, silent) {
    if (sprite.runAnimationRate === 0) {
      sprite.runAnimationRate = this.RUN_ANIMATION_RATE;
    }

    sprite.exploding = true;

    this.explosionAnimator.start(sprite, true);  // true means sprite reappears
  },
};
```

When the runner is jumping, she doesn't animate because her animation rate is zero. The `explode()` method sets her animation rate to its normal value so that she animates through the

explosion cells. The `explode()` method then sets the runner's `exploding()` attribute to `true` and starts the explosion animator.

Next time

In the next article in this series, see how to implement realistic falling by incorporating gravity, and how to add sound and music to Snail Bait.

Downloads

Description	Name	Size	Download method
Sample code	wa-html5-game8-code.zip	1.2MB	HTTP

Information about download methods

Resources

Learn

- 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.
- Space partitioning (Wikipedia): Read about spatial partitioning.
- The Making of an HTML5 Platform Game: Watch David Geary's February 20, 2013 presentation to the Atlanta HTML 5 User Group.
- Snail Bait: Play Snail Bait online in any HTML5-enabled browser (best in Chrome version 18+).
- 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 (Wikipedia): Read about platform games.
- Side-scroller video games (Wikipedia): Read about side-scroller video games.
- HTML5 fundamentals: Learn HTML5 basics with this developerWorks knowledge path.
- developerWorks Web development zone: Find articles covering various web-based solutions. See the Web development technical library for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.
- developerWorks technical events and webcasts: Stay current with technology in these sessions.
- developerWorks Live! briefings: Get up to speed quickly on IBM products and tools as well as IT industry trends.
- developerWorks on-demand demos: Watch demos ranging from product installation and setup for beginners, to advanced functionality for experienced developers.
- developerWorks on Twitter: Join today to follow developerWorks tweets.

Get products and technologies

- Replica Island: You can download the source for this popular open source platform video game for Android. Most of Snail Bait's sprites are from Replica Island (used with permission).
- IBM product evaluation versions: Download or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2, Lotus, Rational, Tivoli, and WebSphere.

Discuss

- developerWorks community: Connect with other developerWorks users while exploring the developer-driven blogs, forums, groups, and wikis.
- Find other developerWorks members interested in web development.

About the author

David Geary

The author of Core HTML5 Canvas, David Geary is also the co-founder of the HTML5 Denver User's Group and the author of eight Java books, including the best-selling books on Swing and JavaServer Faces. David is a frequent speaker at conferences, including JavaOne, Devoxx, Strange Loop, NDC, and OSCON, and is a three-time JavaOne Rock Star. He is the author of the HTML5 2D game development, JSF 2 fu, and GWT fu article series for developerWorks.