HTML5 による 2D ゲームの開発: スプライト

Snail Bait に登場するキャラクターを実装する

David Geary
Author and speaker
Clarity Training, Inc.

2013年 1月 17日

この連載では、HTML5 のエキスパートである David Geary が、HTML5 で 2D テレビ・ゲームを実装する方法について順を追って説明します。スプライトというグラフィック・オブジェクトにはビヘイビアを割り当てることができるため、スプライトはテレビ・ゲームで最も基本的で最も重要な要素の 1 つです。今回は Snail Bait に登場する一連のキャラクターのスプライトを実装する方法について説明します。

このシリーズの他の記事を見る

映画、ドラマ、フィクションなど、他の形式の芸術と同様、ゲームにはキャラクターが登場し、それぞれのキャラクターは特定の役割を演じます。例えば Snail Bait には、ランナー (このゲームの主人公)、コイン、ルビー、サファイア、蜂、コウモリ、ボタン、そしてカタツムリのキャラクターがあり、図 1 にはその大部分が表示されています。この連載の第 1 回 (「スプライト: 登場するキャラクター」セクションを参照) では、このゲームでの各キャラクターと、それぞれの役割について説明しました。

図 1. Snail Bait のキャラクター

Snail Bait の各キャラクターはスプライトです。スプライトはグラフィック・オブジェクトであり、スプライトにはビヘイビアを持たせることができます。例えば、ランナーは走ったり、ジャンプしたり、落下したり、ゲーム上の別のスプライトとぶつかったりすることができ、ルビーとサファイアは、きらきら輝いたり上下に動いたりし、またランナーとぶつかって消えたりします。

スプライトという言葉の由来

アニメーション・キャラクターに対してスプライトという言葉を初めて使用したのは、Texas Instruments の 9918A という VDP (Video-Display Processor) を実装した人達の 1 人でした。(標準的な英語では、sprite という単語はラテン語の spiritus に由来し、小人や妖精を意味します。) スプライトはソフトウェアでもハードウェアでも実装され、1985年に発売された Commodore Amiga は最大 8 つのスプライトをハードウェアでサポートしていました。

スプライトはあらゆるゲームの最も基本的な要素の 1 つであり、またゲームには通常、多くのスプライトがあるため、再利用可能なオブジェクトの中にスプライトの基本機能をカプセル化するのが有効です。この記事では以下の方法について説明します。

- 任意のゲームで再利用できる `Sprite` オブジェクトを実装する方法
- スプライトを描画するオブジェクト (「スプライト・アーティスト」と呼ばれます) からスプライトを分離し、実行時の柔軟性を高める方法
- スプライト・シートを使用して起動時間とメモリー要件を減らす方法
- メタデータによってスプライトを作成する方法
- スプライトをゲーム・ループに組み込む方法

この記事で使用するサンプル・コード全体を入手するには「ダウンロード」を参照してください。

スプライト・オブジェクト

Snail Bait のスプライトは任意のゲームに使用できる JavaScript オブジェクトとして実装されているため、スプライトはそれぞれ独自のファイルの中にあります。ここではスプライトのファイルを `<script src='js/sprites.js'></script>` のようにして Snail Bait の HTML に含めました。

表 1 は `Sprite` の属性の一覧です。

表 1. `Sprite` の属性

属性	説明
`artist`	スプライトを描画するオブジェクト。
`behaviors`	ビヘイビアの配列であり、その要素である各ビヘイビアは対応するスプライトを何らかの形で操作します。
`left`	スプライトの左上隅の `X` 座標。
`top`	スプライトの左上隅の `Y` 座標。
`width`	スプライトの幅をピクセル数で表現した値。
`height`	スプライトの高さをピクセル数で表現した値。
`opacity`	スプライトの不透明度が、不透明なのか、透明なのか、あるいはその中間のどこかであるのかを表します。
`type`	スプライトのタイプ (`bat`、`bee`、`runner` など) を表すストリング。
`velocityX`	スプライトの水平方向の速度で、毎秒のピクセル数で指定されます。
`velocityY`	スプライトの垂直方向の速度で、毎秒のピクセル数で指定されます。
`visible`	スプライトの可視性を表します。値が false の場合、そのスプライトは描画されません。

スプライトは、位置と大きさ (スプライトの「境界ボックス」と呼ばれます)、速度、そして可視性を情報として持つ単純なオブジェクトです。また、スプライトが持つ情報には、タイプ (タイプを使用してスプライト同士を区別することができます)、そして不透明度 (つまりスプライトをある程度透明にすることができます) もあります。

スプライトの描画とビヘイビアは、それぞれ `artist`、`behaviors` と呼ばれる他のオブジェクトによって規定されます。

リスト 1 はスプライトの属性を初期値に設定する `Sprite` コンストラクターを示しています。

リスト 1. `Sprite` コンストラクター

```
var Sprite = function (type, artist, behaviors) { // constructor
   this.type = type || '';
   this.artist = artist || undefined;
   this.behaviors = behaviors || [];

   this.left = 0;
   this.top = 0;
   this.width = 10;   // Something other than zero, which makes no sense
   this.height = 10;  // Something other than zero, which makes no sense
   this.velocityX = 0;
   this.velocityY = 0;
   this.opacity = 1.0;
   this.visible = true;

   return this;
};
```

表示とビヘイビア

Sprite メソッドのシグニチャーによって、表示とビヘイビアの間での関心の分離が行われます。つまり draw() は Canvas のコンテキストを使用してスプライトを描画する一方、update() は現在の時刻とアニメーションのフレーム・レートに基づいてスプライトの状態のみを更新するように設計されています。ビヘイビアは描画を行ってはならず、アーティストはスプライトの状態を操作してはなりません。

リスト 1 のコンストラクターの引数はすべてオプションです。behaviors を指定しない場合には、コンストラクターは空の配列を作成し、type を指定せずにスプライトを作成した場合には、そのスプライトのタイプは空のストリングになります。artist を指定しない場合には、単に artist が未定義となるにすぎません。

属性の他に、スプライトには表 2 に示す 2 つのメソッドがあります。

表 2. `Sprite` のメソッド

メソッド	説明
`draw(context)`	スプライトが表示されており、そのスプライトにアーティストがある場合には、そのスプライトのアーティストの `draw()` メソッドを呼び出します。
`update(time, fps)`	スプライトの各ビヘイビアの `update()` メソッドを呼び出します。

表 2 のメソッドの実装をリスト 2 に示します。

リスト 2. `Sprite` のメソッドの実装

```
Sprite.prototype = { // methods
   draw: function (context) {
     context.save();

     // Calls to save() and restore() make the globalAlpha setting temporary

     context.globalAlpha = this.opacity;

     if (this.artist && this.visible) {
        this.artist.draw(this, context);
     }
```

```
      context.restore();
   },

   update: function (time, fps) {
      for (var i=0; i < this.behaviors.length; ++i) {
         if (this.behaviors[i] === undefined) { // Modified while looping?
            return;
         }

         this.behaviors[i].execute(this, time, fps);
      }
   }
};
```

スプライトの速度は毎秒のピクセル数で指定

この連載の第 2 回の記事で説明したように (第 2 回の「タイム・ベースで動かす」セクションを参照)、ゲームのアニメーションのフレーム・レートとスプライトの動きは独立している必要があります。この要件から、スプライトの速度は毎秒のピクセル数で指定します。

リスト 1 とリスト 2 を見るとわかるように、スプライトは複雑なものではありません。スプライトに関する複雑さの大部分はスプライトのアーティストとビヘイビアの中にカプセル化されています。また、スプライトはアーティストやビヘイビアといったオブジェクトとは分離されているため、スプライトのアーティストとビヘイビアを実行時に変更可能であるということも理解することが重要です。実際、この連載の次回の記事で説明するように、いくつものスプライトに使用できる一般的なビヘイビアを実装することができ、そうした方が極めて望ましい場合がよくあるのです。

これでスプライトの実装方法を理解できたので、今度はスプライト・アーティストの実装について見ていきましょう。

スプライト・アーティストとスプライト・シート

スプライト・アーティストは以下の 3 つのいずれかとして実装することができます。

- ストローク・アーティストとフィル・アーティスト: 直線、弧、曲線などのグラフィックス・プリミティブを描画します。
- 画像アーティスト: 2D コンテキストの `drawImage()` メソッドを使用して画像を描画します。
- スプライト・シート・アーティスト: スプライト・シートから (また `drawImage()` を使用して) 画像を描画します。

リスト 2 からわかるように、アーティストのタイプにかかわらず、すべてのスプライト・アーティストが満たさなければならない要件は 1 つのみです。つまりスプライト・アーティストは、スプライトと Canvas の 2D コンテキストを引数にとる `draw()` メソッドを実装するオブジェクトでなければなりません。

次に、それぞれのタイプのアーティストについて、スプライト・シートの説明を間に交えて説明します。

ストローク・アーティストとフィル・アーティスト

ストローク・アーティストとフィル・アーティストには基準となる実装はなく、Canvas の 2D コンテキストのグラフィック機能を利用して状況に合わせて実装します。リスト 3 は Snail Bait のプ

ラットフォームのスプライトを描画するストローク・アーティストとフィル・アーティストの実装を示しています。

リスト 3. ストローク・アーティストとフィル・アーティスト

```
// Stroke and fill artists draw with Canvas 2D drawing primitives

var SnailBait =  function (canvasId) { // constructor
   ...

   this.platformArtist = {
      draw: function (sprite, context) {
         var top;

         context.save();

         top = snailBait.calculatePlatformTop(sprite.track);

         // Calls to save() and restore() make the following settings temporary

         context.lineWidth = snailBait.PLATFORM_STROKE_WIDTH;
         context.strokeStyle = snailBait.PLATFORM_STROKE_STYLE;
         context.fillStyle = sprite.fillStyle;

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

         context.restore();
      }
   },
};
```

図 1 からわかるように、プラットフォームは単純な長方形です。リスト 3 のプラットフォーム・アーティストは Canvas の 2D コンテキストの `strokeRect()` メソッドと `fillRect()` メソッドを使用してこれらの長方形を描画します。これらのメソッドについては、この連載の第 2 回の記事 (第 2 回の「HTML5 の Canvas の概要」を参照) で説明しました。この後で描画される長方形の位置と大きさはプラットフォームのスプライトの境界ボックスによって決まります。

画像アーティスト

ストローク・アーティストやフィル・アーティストとは異なり、画像アーティストには基準となる実装があります。それをリスト 4 に示します。

リスト 4. 画像アーティスト

```
// ImageArtists draw an image

var ImageArtist = function (imageUrl) { // constructor
   this.image = new Image();
   this.image.src = imageUrl;
};

ImageArtist.prototype = { // methods
   draw: function (sprite, context) {
      context.drawImage(this.image, sprite.left, sprite.top);
   }
};
```

画像の URL を使用して画像アーティストを作成し、画像アーティストの `draw()` メソッドにより、そのスプライトの位置でその画像全体を描画します。

Snail Bait の場合はスプライト・シートから画像を描画した方が効率的なので、画像アーティストは使用しません。

スプライト・シート

Web サイトを素早くロードするために最も効果的な方法の 1 つは、送信する HTTP リクエストの数を必要最小限に減らすことです。ほとんどのゲームは大量の画像を使用するため、それらの各画像に対して HTTP リクエストを別々に実行すると起動時間が遅くなります。そこで HTML5 ゲームを開発する人達は、そのゲームの画像すべてを含む 1 つの大きな画像を作成します。その 1 つの画像をスプライト・シートと呼びます。図 2 は Snail Bait のスプライト・シートを示しています。

図 2. Snail Bait のスプライト・シート

スプライト・シートが指定された場合、そのスプライト・シートから特定の長方形の部分をキャンバスに描画するための方法が必要です。幸いなことに、それは Canvas の 2D コンテキストの `drawImage()` メソッドを使用すれば簡単です。この手法は、スプライト・シート・アーティストが使用します。

スプライト・シート・アーティスト

スプライト・シート・アーティストの実装をリスト 5 に示します。

リスト 5. スプライト・シート・アーティスト

```
// Sprite sheet artists draw an image from a sprite sheet

SpriteSheetArtist = function (spritesheet, cells) { // constructor
   this.cells = cells;
   this.spritesheet = spritesheet;
```

```
   this.cellIndex = 0;
};

SpriteSheetArtist.prototype = { // methods
   advance: function () {
      if (this.cellIndex == this.cells.length-1) {
         this.cellIndex = 0;
      }
      else {
         this.cellIndex++;
      }
   },

   draw: function (sprite, context) {
      var cell = this.cells[this.cellIndex];

      context.drawImage(this.spritesheet,
                cell.left,   cell.top,     // source x, source y
                cell.width,  cell.height,  // source width, source height
                sprite.left, sprite.top,   // destination x, destination y
                cell.width,  cell.height); // destination width, destination height
   }
};
```

スプライト・シート・アーティストをインスタンス化するには、スプライト・シートへの参照と、境界ボックスの配列を使用します。これらの境界ボックスはセルと呼ばれ、それぞれのセルがスプライト・シート内の長方形の領域を表しており、その領域に 1 つのスプライト画像が含まれます。

またスプライト・シート・アーティストは、セルの中にインデックスも保持します。スプライト・シートの `draw()` メソッドはそのインデックスを使用して現在のセルにアクセスし、Canvas の 2D コンテキストの `drawImage()` メソッドのなかでも 9 つの引数を取るバージョンのメソッドを使用して、そのセルの内容をキャンバス上のそのスプライトの位置に描画します。

スプライト・シート・アーティストの `advance()` メソッドは、セルのインデックスを次のセルへと進めます。インデックスが最後のセルを指している場合は、先頭のセルに戻ります。続けてスプライト・シート・アーティストの `draw()` メソッドを呼び出すと、そのインデックスに対応する画像が描画されます。繰り返しインデックスを進めながら描画することで、スプライト・シート・アーティストはスプライト・シートの一連の画像を連続的に描画することができます。

リスト 5 を見るとわかるように、スプライト・シート・アーティストの実装は容易です。またスプライト・シート・アーティストは使い方も容易です。単純にスプライト・シートといくつかのセルによってスプライト・シート・アーティストをインスタンス化し、必要に応じて `advance()` メソッドと `draw()` メソッドを呼び出せばよいのです。ただし、セルの定義には注意が必要です。

スプライト・シートのセルを定義する

リスト6 は Snail Bait のスプライト・シート内でのコウモリ、蜂、カタツムリのセルの定義を示しています。

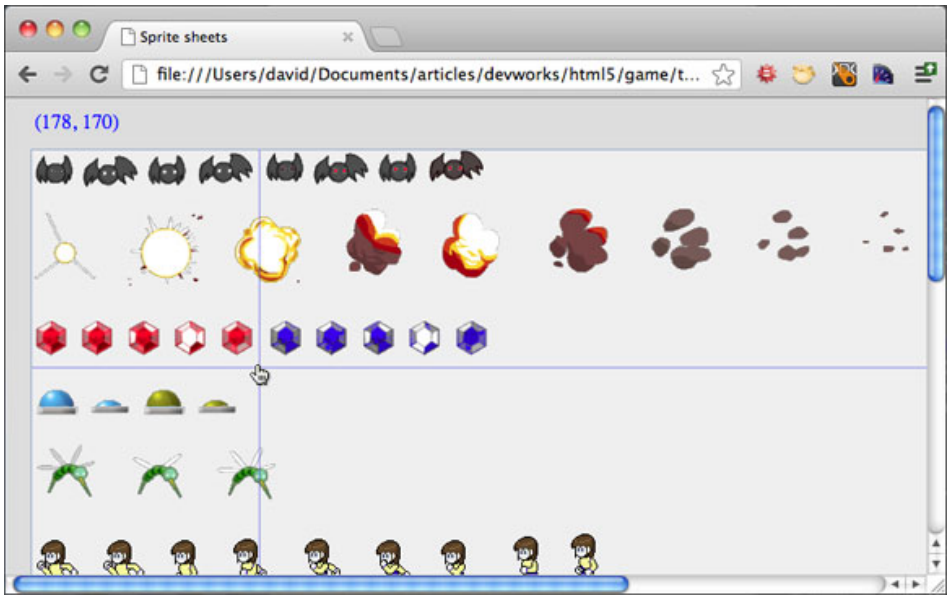
リスト 6. Snail Bait のスプライト・シートのセルを定義する

```
var BAT_CELLS_HEIGHT = 34,

    BEE_CELLS_WIDTH  = 50,
    BEE_CELLS_HEIGHT = 50,
```

```
...

SNAIL_CELLS_WIDTH = 64,
SNAIL_CELLS_HEIGHT = 34,

...

// Spritesheet cells.....................................

batCells = [
   { left: 1,   top: 0, width: 32, height: BAT_CELLS_HEIGHT },
   { left: 38,  top: 0, width: 46, height: BAT_CELLS_HEIGHT },
   { left: 90,  top: 0, width: 32, height: BAT_CELLS_HEIGHT },
   { left: 129, top: 0, width: 46, height: BAT_CELLS_HEIGHT },
],

beeCells = [
   { left: 5,   top: 234, width: BEE_CELLS_WIDTH, height: BEE_CELLS_HEIGHT },
   { left: 75,  top: 234, width: BEE_CELLS_WIDTH, height: BEE_CELLS_HEIGHT },
   { left: 145, top: 234, width: BEE_CELLS_WIDTH, height: BEE_CELLS_HEIGHT }
],
...

snailCells = [
   { left: 142, top: 466, width: SNAIL_CELLS_WIDTH, height: SNAIL_CELLS_HEIGHT },
   { left: 75,  top: 466, width: SNAIL_CELLS_WIDTH, height: SNAIL_CELLS_HEIGHT },
   { left: 2,   top: 466, width: SNAIL_CELLS_WIDTH, height: SNAIL_CELLS_HEIGHT },
];
```

セルの境界ボックスを判断するのは、単調で退屈な作業です。そこで少し時間をかけ、そのためのツールを実装するのが得策です。図 3 に示すのはまさにそうしたツールであり、このツールはオンライン (Core HTML5 Canvas サイト) で実行することができます (「参考文献」を参照)。

図 3. スプライト・シートを検証するための単純なプログラムの実行画面



ゲーム開発者のツールボックス

ゲーム開発者の仕事は楽しいことやゲームをすることばかりではありません。ゲーム開発者はスプライト・シートのセルの判断やゲームのステージの設計など、単調で退屈な作業に多

くの時間を費やします。そこでほとんどのゲーム開発者は多くの時間をかけ、そうした単調で退屈な作業の支援をする図 3 のようなツールを実装します。

図 3 に示すアプリケーションは 1 つの画像を表示し、その画像内でのマウスの動きを追跡します。マウスを動かすと、このアプリケーションはガイドとなる線を表示し、マウス・カーソルの現在の位置を表示する左上隅の読み取り値を更新します。このツールにより、各画像とスプライト・シートの境界ボックスを容易に判断することができます。

これで、スプライトとスプライト・アーティストの実装方法を十分に理解できたので、Snail Bait がスプライトをどのように作成し、初期化するのかを調べてみましょう。

Snail Bait のスプライトの作成と初期化

Snail Bait はいくつかの配列を定義し、これらの配列は最終的にスプライトを格納します (リスト 7)。

リスト 7. ゲームのコンストラクターの中でスプライトの配列を定義する

```
var SnailBait = function (canvasId) { // constructor
   ...

   this.bats         = [],
   this.bees         = [],
   this.buttons      = [],
   this.coins        = [],
   this.platforms    = [],
   this.rubies       = [],
   this.sapphires    = [],
   this.snails       = [],

   this.runner = new Sprite('runner', this.runnerArtist);

   this.sprites = [ this.runner ]; // Add other sprites later
   ...
};
```

リスト 7 の各配列には同じタイプのスプライトが含まれています。例えば、`bats` 配列にはコウモリのスプライトが含まれ、`bees` 配列には蜂のスプライトが含まれるといった具合です。またこのゲームには、ゲームのすべてのスプライトを含む配列もあります。bees、bats などの個々の配列は必ずしも必要なわけではありません。実際、これらの配列は冗長なのですが、これらの配列があることによってパフォーマンスが向上します。例えば、このゲームでランナーがプラットフォームに着地したかどうかを判断する場合、`platforms` 配列に対して繰り返し処理を行う方が `sprites` 配列に対して繰り返し処理を行ってプラットフォームを検索するよりも効率的です。

またリスト 7 から、このゲームでランナーのスプライトがどのように作成され、そのスプライトがどのように `sprites` 配列に追加されるかもわかります。このゲームにはランナーが 1 人しか登場しないため、ランナーの配列はありません。このゲームは `runner` というタイプと 1 つのアーティストによってランナーをインスタンス化していますが、ランナーをインスタンス化するときに何もビヘイビアを規定していないことに注意してください。これらのビヘイビアについては次回の記事で説明しますが、ビヘイビアは後でコードに追加されます。

ゲームが開始されると、Snail Bait は (他のことをすると同時に) `createSprites()` メソッドを呼び出します (リスト 8)。

リスト 8. ゲームを開始する

```
SnailBait.prototype = { // methods
   ...
   start: function () {
      this.createSprites();
      this.initializeImages();
      this.equipRunner();
      this.splashToast('Good Luck!');
   },
};
```

`createSprites()` メソッドはランナーを除く Snail Bait のすべてのスプライトを作成します。このメソッドをリスト 9 に示します。

リスト 9. Snail Bait のスプライトを作成し、初期化する

```
SnailBait.prototype = { // methods
   ...
   createSprites: function() {
      this.createPlatformSprites();

      this.createBatSprites();
      this.createBeeSprites();
      this.createButtonSprites();
      this.createCoinSprites();
      this.createRubySprites();
      this.createSapphireSprites();
      this.createSnailSprites();

      this.initializeSprites();

      this.addSpritesToSpriteArray();
   },
```

`createSprites()` は、ヘルパー関数を呼び出してさまざまなタイプのスプライトを作成し、それに続いてスプライトを初期化するメソッドとスプライトを `sprites` 配列に追加するメソッドを呼び出します。ヘルパー関数の実装をリスト 10 に示します。

リスト 10. 個々のスプライトを作成する

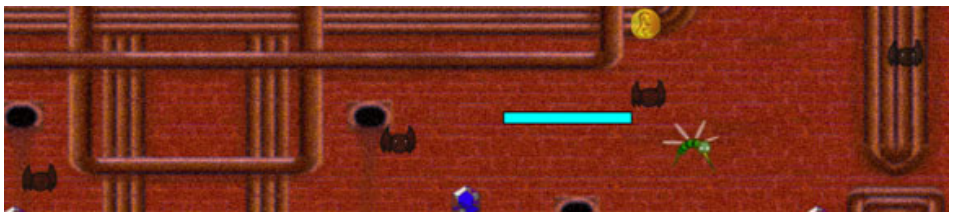
```
SnailBait.prototype = { // methods
  ...
  createBatSprites: function () {
    var bat,  batArtist = new SpriteSheetArtist(this.spritesheet, this.batCells),
   redEyeBatArtist = new SpriteSheetArtist(this.spritesheet, this.batRedEyeCells);

    for (var i = 0; i < this.batData.length; ++i) {
      if (i % 2 === 0) bat = new Sprite('bat', batArtist);
      else             bat = new Sprite('bat', redEyeBatArtist);

      bat.width  = this.BAT_CELLS_WIDTH;
      bat.height = this.BAT_CELLS_HEIGHT;

      this.bats.push(bat);
    }
  },

  createBeeSprites: function () {
    var bee, beeArtist = new SpriteSheetArtist(this.spritesheet, this.beeCells);

    for (var i = 0; i < this.beeData.length; ++i) {
      bee = new Sprite('bee', beeArtist);
```

```
    bee.width  = this.BEE_CELLS_WIDTH;
    bee.height = this.BEE_CELLS_HEIGHT;

    this.bees.push(bee);
  }
},

createButtonSprites: function () {
  var button, buttonArtist = new SpriteSheetArtist(this.spritesheet, this.buttonCells),

  goldButtonArtist = new SpriteSheetArtist(this.spritesheet, this.goldButtonCells);

  for (var i = 0; i < this.buttonData.length; ++i) {
    if (i === this.buttonData.length - 1) {
       button = new Sprite('button', goldButtonArtist);
    }
    else {
       button = new Sprite('button', buttonArtist);
    }

    button.width  = this.BUTTON_CELLS_WIDTH;
    button.height = this.BUTTON_CELLS_HEIGHT;

    button.velocityX = this.BUTTON_PACE_VELOCITY;
    button.direction = this.RIGHT;

    this.buttons.push(button);
  }
},

createCoinSprites: function () {
  var coin, coinArtist = new SpriteSheetArtist(this.spritesheet, this.coinCells);

  for (var i = 0; i < this.coinData.length; ++i) {
    coin        = new Sprite('coin', coinArtist);
    coin.width  = this.COIN_CELLS_WIDTH;
    coin.height = this.COIN_CELLS_HEIGHT;

    this.coins.push(coin);
  }
},

createPlatformSprites: function () {
  var sprite, pd;  // Sprite, Platform data

  for (var i=0; i < this.platformData.length; ++i) {
    pd = this.platformData[i];
    sprite           = new Sprite('platform-' + i, this.platformArtist);
    sprite.left      = pd.left;
    sprite.width     = pd.width;
    sprite.height    = pd.height;
    sprite.fillStyle = pd.fillStyle;
    sprite.opacity   = pd.opacity;
    sprite.track     = pd.track;
    sprite.button    = pd.button;
    sprite.pulsate   = pd.pulsate;
    sprite.power     = pd.power;
    sprite.top       = this.calculatePlatformTop(pd.track);

    this.platforms.push(sprite);
  }
},

createSapphireSprites: function () {
  // Listing omitted for brevity. Discussed in the next article in this series.
},
```

```
  createRubySprites: function () {
    // Listing omitted for brevity. Discussed in the next article in this series.
  },

  createSnailSprites: function () {
    // Listing omitted for brevity. Discussed in the next article in this series.
  },
};
```

リスト 10 のメソッドは 3 つの理由から注目に値します。第 1 に、作成されるメソッドはどれも非常に単純です。各メソッドはスプライトを作成し、そのスプライトの幅と高さを設定し、そのスプライトを個々のスプライト配列に追加します。第 2 に、`createBatSprites()` と `createButtonSprites()` は複数のアーティストを使用して同じタイプのスプライトを作成します。`createBatSprites()` メソッドはアーティストを切り換えることにより、半数のコウモリが赤い目を持ち、残りの半数が白い目を持つようにします (図 4)。`createButtonSprites()` メソッドは複数のアーティストを使用して青色のボタンまたは金色のボタンのいずれかを描画します。

図 4. 赤い目のコウモリと白い目のコウモリ



リスト 10 のメソッドで注目すべき 3 番目の点は、最も興味深いものであり、これらのメソッドがすべて、スプライトのメタデータの配列からスプライトを作成していることです。

メタデータによってスプライトを作成する

リスト 11 に、Snail Bait のスプライトのメタデータをいくつか示します。

リスト 11. スプライトのメタデータ

```
var SnailBait = function (canvasId) {
  // Bats.......................................................

   this.batData = [
      { left: 1150, top: this.TRACK_2_BASELINE - this.BAT_CELLS_HEIGHT },
      { left: 1720, top: this.TRACK_2_BASELINE - 2*this.BAT_CELLS_HEIGHT },
      { left: 2000, top: this.TRACK_3_BASELINE },
      { left: 2200, top: this.TRACK_3_BASELINE - this.BAT_CELLS_HEIGHT },
      { left: 2400, top: this.TRACK_3_BASELINE - 2*this.BAT_CELLS_HEIGHT },
   ],

   // Bees.......................................................

   this.beeData = [
      { left: 500,  top: 64 },
      { left: 944,  top: this.TRACK_2_BASELINE - this.BEE_CELLS_HEIGHT - 30 },
      { left: 1600, top: 125 },
      { left: 2225, top: 125 },
      { left: 2295, top: 275 },
      { left: 2450, top: 275 },
   ],

   // Buttons....................................................
```

```
   this.buttonData = [
      { platformIndex: 7 },
      { platformIndex: 12 },
   ],

   // Metadata for Snail Bait's other sprites is omitted for brevity
};
```

メタデータからスプライトを作成すると、以下のような利点があります。

- スプライトのメタデータは、コード内に分散されず、1 ヶ所にまとまって配置されます。
- スプライトを作成するメソッドは、メタデータと分離されていると、より単純になります。
- メタデータはどこにあっても構いません。

スプライトのメタデータはコード内の 1 ヶ所にあるため、それを見つけて修正するのは容易です。またスプライトを作成するメソッドの外部でメタデータが定義されるため、これらのメソッドは単純になり、従って理解するのも変更するのも容易になります。最後に、Snail Bait のメタデータはコードに直接埋め込まれていますが、スプライトのメタデータはどこにあっても構わず、例えばゲームの実行時にメタデータを作成するために使用されるレベル・エディターの中にあっても構いません。つまり簡単に言えば、メタデータは変更が容易であり、スプライトを作成するメソッドの中で直接スプライトのデータを指定するよりもメタデータを使用した方が柔軟になるのです。

リスト 9 で、Snail Bait の `createSprites()` メソッドがゲームのスプライトを作成した後で、`initializeSprites()` と `addSpritesToSpriteArray()` という 2 つのメソッドを呼び出していることを思い出してください。リスト 12 は `initializeSprites()` メソッドを示しています。

リスト 12. Snail Bait のスプライトを初期化する

```
SnailBait.prototype = { // methods
   ...

   initializeSprites: function() {
      this.positionSprites(this.bats,       this.batData);
      this.positionSprites(this.bees,       this.beeData);
      this.positionSprites(this.buttons,    this.buttonData);
      this.positionSprites(this.coins,      this.coinData);
      this.positionSprites(this.rubies,     this.rubyData);
      this.positionSprites(this.sapphires,  this.sapphireData);
      this.positionSprites(this.snails,     this.snailData);
   },

   positionSprites: function (sprites, spriteData) {
      var sprite;

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

         if (spriteData[i].platformIndex) { // put sprite on a platform
            this.putSpriteOnPlatform(sprite, this.platforms[spriteData[i].platformIndex]);
         }
         else {
            sprite.top  = spriteData[i].top;
            sprite.left = spriteData[i].left;
         }
      }
   },
```

```
};
```

initializeSprites() はゲームのスプライトの各配列に対して positionSprites() を呼び出します。そしてこの positionSprites() は、スプライトのメタデータによって指定された位置にスプライトを配置します。ボタンやカタツムリなど一部のスプライトがプラットフォームの上にあることに注意してください。putSpriteOnPlatform() メソッドをリスト 13 に示します。

リスト 13. スプライトをプラットフォームに配置する

```
SnailBait.prototype = { // methods
   ...

   putSpriteOnPlatform: function(sprite, platformSprite) {
      sprite.top  = platformSprite.top - sprite.height;
      sprite.left = platformSprite.left;
      sprite.platform = platformSprite;
   },
}
```

スプライトとプラットフォームが指定されると、putSpriteOnPlatform() メソッドはそのスプライトを指定のプラットフォームの上に配置し、後で参照できるようにそのプラットフォームへの参照をそのスプライトに保存します。

ご想像のとおり、リスト 14 を見るとわかるように、すべてのスプライトを含む sprites 配列に個々のスプライトを追加するのは簡単です。

リスト 14. Snail Bait のスプライトを作成し、初期化する

```
SnailBait.prototype = { // methods
   ...

   addSpritesToSpriteArray: function () {
      var i;

      for (i=0; i < this.bats.length; ++i) {
         this.sprites.push(this.bats[i]);
      }

      for (i=0; i < this.bees.length; ++i) {
         this.sprites.push(this.bees[i]);
      }

      for (i=0; i < this.buttons.length; ++i) {
         this.sprites.push(this.buttons[i]);
      }

      for (i=0; i < this.coins.length; ++i) {
         this.sprites.push(this.coins[i]);
      }

      for (i=0; i < this.rubies.length; ++i) {
         this.sprites.push(this.rubies[i]);
      }

      for (i=0; i < this.sapphires.length; ++i) {
         this.sprites.push(this.sapphires[i]);
      }

     for (i=0; i < this.snails.length; ++i) {
```

```
         this.sprites.push(this.snails[i]);
      }

      for (i=0; i < this.snailBombs.length; ++i) {
         this.sprites.push(this.snailBombs[i]);
      }
   },
};
```

これで、スプライトとスプライト・アーティストの実装方法、そして Snail Bait でスプライトを作成する方法と初期化する方法を理解できたので、今度はスプライトをどのように Snail Bait のゲーム・ループに組み込むのかを説明します。

スプライトをゲーム・ループに組み込む

この連載の第 2 回の記事で、Snail Bait の水平方向の動きのほとんどすべては、Canvas の 2D コンテキストの変換によって実現されていると説明したことを思い出してください (第 2 回の「背景をスクロールさせる」セクションを参照)。Snail Bait は常に大多数のスプライトを同じ水平位置で描画しており、それらのスプライトが明らかに水平方向に動いているように見えるのは、あくまでもその変換の結果にすぎません。Snail Bait のほとんどのスプライトは、ゲームのプラットフォームに合わせて水平方向に移動します。それをリスト 15 に示します。

リスト 15. スプライトのオフセットを更新する

```
SnailBait.prototype = {
   draw: function (now) {
      this.setPlatformVelocity();
      this.setTranslationOffsets();

      this.drawBackground();

      this.updateSprites(now);
      this.drawSprites();
   },

   setPlatformVelocity: function () {
      // Setting platform velocity was discussed in the second article in this series

      this.platformVelocity = this.bgVelocity * this.PLATFORM_VELOCITY_MULTIPLIER;
   },

   setTranslationOffsets: function () {
      // Setting the background translation offset was discussed
      // in the second article in this series

      this.setBackgroundTranslationOffset();
      this.setSpriteTranslationOffsets();
   },

   setSpriteTranslationOffsets: function () {
      var i, sprite;

      this.spriteOffset += this.platformVelocity / this.fps; // In step with platforms

      for (i=0; i < this.sprites.length; ++i) {
         sprite = this.sprites[i];

         if ('runner' !== sprite.type) {
            sprite.offset = this.platformOffset; // In step with platforms
         }
```

```
      }
   },
   ...
};
```

`draw()` メソッドは、まずプラットフォームの速度を設定し、それに続いてランナー以外のすべてのスプライトの変換のオフセットも設定します。(ランナーの水平位置は固定されており、プラットフォームに合わせて移動することはありません。)

オフセットの変換を設定して背景を描画すると、`draw()` メソッドは `updateSprites()` と `drawSprites()` を使用してゲームのスプライトの更新と描画を行います。これらのメソッドをリスト 16 に示します。

リスト 16. スプライトを更新し、描画する

```
SnailBait.prototype = {
   ...
   updateSprites: function (now) {
      var sprite;

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

         if (sprite.visible && this.spriteInView(sprite)) {
            sprite.update(now, this.fps);
         }
      }
   },

   drawSprites: function() {
      var sprite;

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

         if (sprite.visible && this.spriteInView(sprite)) {
            this.context.translate(-sprite.offset, 0);

            sprite.draw(this.context);

            this.context.translate(sprite.offset, 0);
         }
      }
   },

   spriteInView: function(sprite) {
      return sprite === this.runner || // runner is always visible
         (sprite.left + sprite.width > this.platformOffset &&
          sprite.left < this.platformOffset + this.canvas.width);
   },
```

スプライトが表示されていない場合

Snail Bait の最終版のプレイ・フィールドの幅はゲームのキャンバスの 4 倍あります (幅は任意であり、もっと広くすることもできます)。そのため、どの時点においても、Snail Bait のゲーム画面の 4 分の 3 は表示されません。その 4 分の 3 の部分にあるスプライトの更新や描画は必要ないため、Snail Bait はその部分のスプライトの更新や描画はしません。厳密に言えば、これらのスプライトは結局 Canvas のコンテキストによって除外されるため、描画の際にこれらのスプライトを除外する必要はありません。

`updateSprites()` と `drawSprites()` はどちらも、ゲームのすべてのスプライトに対して繰り返し処理を行い、それぞれスプライトを更新し、描画しますが、それを行うのはスプライトが表示されている場合であって、しかもキャンバス内の現在表示されているセクションにスプライトがある場合のみです。

スプライトを描画する前に、`drawSprites()` メソッドは `setTranslationOffsets()` で計算されるスプライトのオフセットによってコンテキストを変換し、その後でそのコンテキストを逆変換して元の位置に戻します。その結果、スプライトは明らかに水平方向に動いているように見えます。

次回は

この記事では、スプライトとスプライト・アーティストの実装方法、そしてスプライトをゲーム・ループに組み込む方法について説明しました。この連載「HTML5 による 2D ゲームの開発」の次回の記事では、スプライトのビヘイビアを実装する方法と、それらのビヘイビアを特定のスプライトに追加する方法について説明します。ではまた次回お会いしましょう。

ダウンロード

内容	ファイル名	サイズ
Sample code	j-html5-game4.zip	3.9MB

著者について

David Geary

『Core HTML5 Canvas』の著者、David Geary は HTML5 Denver User's Group の共同設立者でもあり、Swing と JavaServer Faces に関するベストセラーの本を含め、Java に関する 8 冊の本の著者でもあります。また彼は、JavaOne、Devoxx、Strange Loop、NDC、OSCON などのカンファレンスで頻繁に講演を行っており、JavaOne Rock Star にも 3 度選ばれています。彼は連載記事、「JSF 2 の魅力」と「GWT の魅力」を developerWorks に寄稿しました。Twitter の @davidgeary で彼をフォローしてください。