HTML5 による 2D ゲームの開発: グラフィックスとアニメーション

キャンバスに描画し、要素に動きを与える

David Geary
Author and speaker
Clarity Training, Inc.

2012年 11月 01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。今回は Canvas グラフィックスと HTML5 によるアニメーションについて説明します。この記事を読むことで、ゲームのグラフィックスの描画方法、それらのグラフィックスに動きを与える方法を理解することができます。またこの記事では、HTML5 でアニメーションを作成するための最も適切な方法、背景をスクロールする方法、パララックスを実装して 3 次元をシミュレートする方法についても説明します。

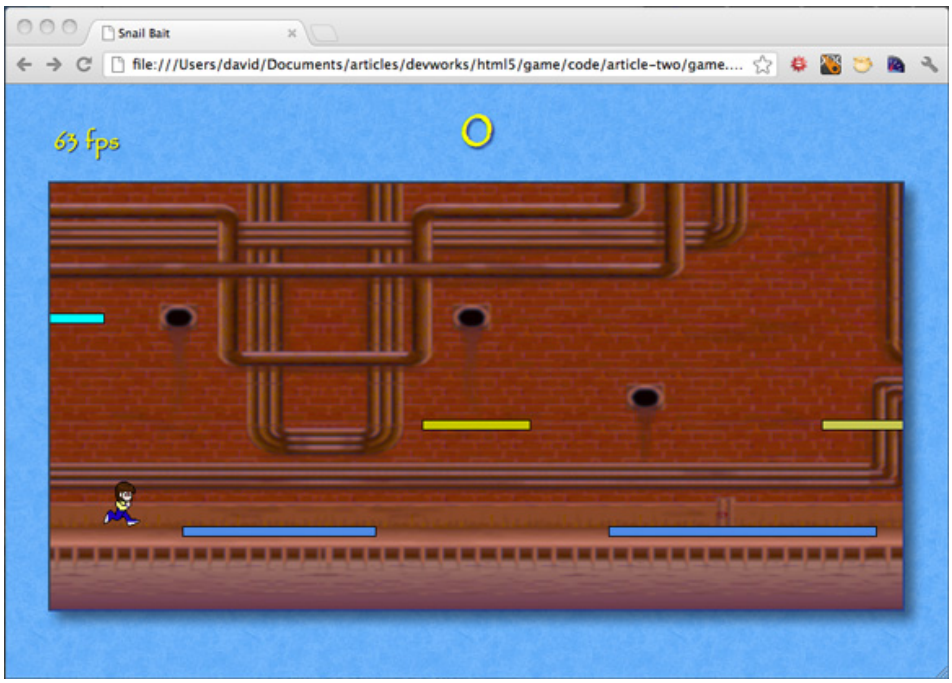
このシリーズの他の記事を見る

どのようなテレビ・ゲームでも、グラフィックスとアニメーションは最も基本的な要素です。そこでこの記事ではまず、Canvas 2D API の概要を簡単に説明し、次に Snail Bait の中心となるアニメーションの実装について説明します。この記事では以下の内容を説明します。

- 画像とグラフィックス・プリミティブをキャンバスに描画する方法
- 動きが滑らかでちらつきのないアニメーションを作成する方法
- ゲーム・ループを実装する方法
- アニメーションのフレーム・レート (フレーム/秒: fps) を監視する方法
- ゲームの背景をスクロールさせる方法
- パララックスを使用して 3 次元をシミュレートする方法
- タイム・ベースの動きを実装する方法

この記事で説明するコードを実行した最終的な結果を図 1 に示します。

図 1. 背景をスクロールさせ、フレーム・レートを監視する



背景とプラットフォームは水平方向にスクロールします。プラットフォームは前面にあるため、背景よりも明らかに速く移動させることで、穏やかなパララックス効果を生み出しています。ゲームが開始されると、背景は右から左へとスクロールを始めます。ステージの最後まで到達すると、背景とプラットフォームのスクロール方向は逆方向 (左から右) になります。

今の段階では、ランナーは動きません。また、このゲームにはまだ衝突検出がないため、ランナーの下にプラットフォームがない場合、ランナーは宙に浮いています。

ゲーム・キャンバスの左上には、最終的には (この連載第 1 回の記事の図 1 に示されているように) 残機数を示すアイコンが表示されるようになりますが、現段階ではアニメーションの現在のフレーム・レート (フレーム/秒: fps) が表示されるようになっています。

即時モードのグラフィックス

Canvas は「即時モード」のグラフィックス・システムです。つまり Canvas は指定されたものを即時に描画して、即時に忘れ去ります。SVG (Scalable Vector Graphics) など、その他のグラフィックス・システムは「保持モード」のグラフィックスを実装しており、描画対象のオブジェクトのリストを保持しています。Canvas は表示リストを保持するオーバーヘッドがないため、SVG よりも高速です。ただし、ユーザーが操作できるオブジェクトのリストを保持したい場合には、その機能を Canvas 内で独自に実装する必要があります。

先へ進む前に、 図 1 の状態のゲームを試してみるとよいかもしれません。そうすれば、コードを容易に理解できるはずです。(今回の記事の状態の Snail Bait の実装を入手するには「ダウンロード」セクションを参照してください。)

HTML5 の Canvas の概要

Canvas の 2D コンテキストには充実したグラフィックス API が用意されており、この API を使用すれば、テキスト・エディターからプラットフォーム・テレビ・ゲームに至るまで、あらゆるもの

を実装することができます。この記事の執筆時点で、この API には 30 を超えるメソッドが含まれていますが、Snail Bait が使用するメソッド (表 1) はそのごく一部にすぎません。

表 1. Snail Bait で使用している Canvas の 2D コンテキストのメソッド

メソッド	説明
`drawImage()`	画像の全体または一部をキャンバス上の特定の場所に描画します。`video` 要素から別のキャンバス (つまりフレーム) を描画することもできます。
`save()`	コンテキストの属性をスタックに保存します。
`restore()`	コンテキストの属性をスタックからポップし、コンテキストに適用します。
`strokeRect()`	塗りつぶされていない四角形を描画します。
`fillRect()`	四角形を塗りつぶします。
`translate()`	座標系を変換します。これは多くのさまざまなシナリオで役立つ強力なメソッドです。Snail Bait のスクロール動作はすべて、この 1 つのメソッドを呼び出すことで実装されています。

パス・ベースのグラフィックス

Apple の Cocoa や Adobe の Illustrator と同様、Canvas API はパス・ベースです。つまりキャンバスにグラフィックス・プリミティブを描画する場合には、パスを作成した後、そのパスに沿って線を描画したり、パスを塗りつぶしたりします。`strokeRect()` メソッドと `fillRect()` メソッドは、それぞれ四角形の枠線を描く、または四角形を塗りつぶすコンビニエンス・メソッドです。

Snail Bait では、プラットフォームを除き、すべてが画像です。背景、ランナー、すべてのアイテムと敵は画像であり、このゲームでは `drawImage()` メソッドを使用してこれらの画像を描画します。

最終的に、Snail Bait はスプライトシート (ゲームのグラフィックスをすべて含む 1 つの画像) を使用します。しかし今はとりあえず、背景とランナーに別々の画像を使用します。ここではランナーをリスト 1 の関数で描画します。

リスト 1. ランナーを描画する

```
function drawRunner() {
   context.drawImage(runnerImage,                                    // image
                     STARTING_RUNNER_LEFT,                           // canvas left
                     calculatePlatformTop(runnerTrack) - RUNNER_HEIGHT); // canvas top
}
```

`drawRunner()` 関数は `drawImage()` に 3 つの引数を渡します。3 つというのは、画像、そしてその画像をキャンバス内に描画するための左座標と上座標です。左座標は定数ですが、上座標はランナーが配置されるプラットフォームによって決まります。

背景も同じようにして描画します。これをリスト 2 に示します。

リスト 2. 背景を描画する

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

多機能な drawImage() メソッド

Canvas の 2D コンテキストの drawImage() メソッドを使用すると、キャンバス内の任意の場所で、1 つの画像全体を描画することも、1 つの画像内に任意の四角形領域を描画することもでき、しかもオプションとして、その画像を描画しながら大きさを変化させることもできます。画像の他に、別のキャンバスのコンテンツや video 要素の現在のフレームを drawImage() によって描画することもできます。drawImage() は単なる 1 つのメソッドですが、このメソッドを使用すると、興味深いながらも他の方法では実装が困難なアプリケーション (動画編集ソフトウェアなど) を簡単に実装することができます。

リスト 2 の drawBackground() 関数は背景の画像をキャンバスの (0,0) に描画します。この記事では後ほど、この関数を変更して背景をスクロールさせます。

プラットフォームは画像ではないため、プラットフォームを描画するには Canvas API をさらに別の方法で使用する必要があります (リスト 3)。

リスト 3. プラットフォームを描画する

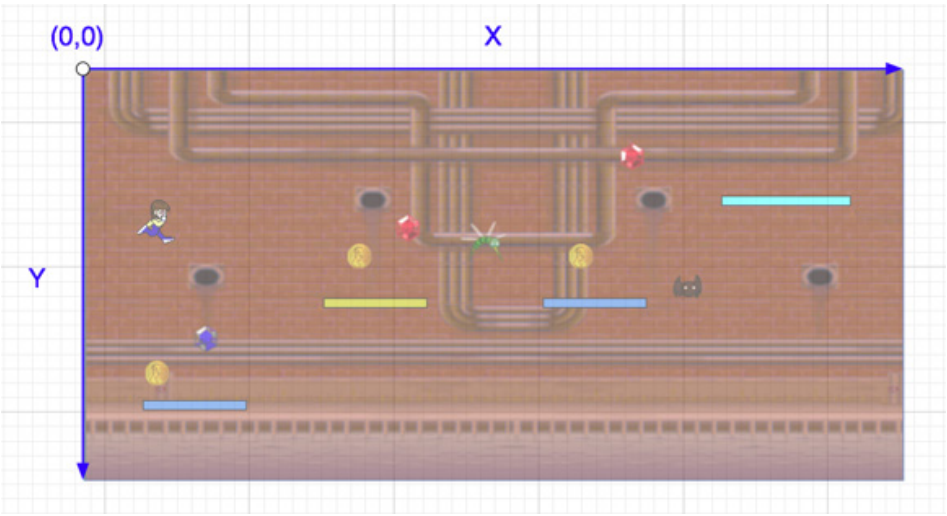
```
var platformData = [
    // Screen 1.....................................................
    {
       left:      10,
       width:     230,
       height:    PLATFORM_HEIGHT,
       fillStyle: 'rgb(150,190,255)',
       opacity:   1.0,
       track:     1,
       pulsate:   false,
    },
    ...
 ],
 ...

function drawPlatforms() {
   var data, top;

   context.save(); // Save the current state of the context

   context.translate(-platformOffset, 0); // Translate the coord system for all platforms

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

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

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

   context.restore(); // Restore context state saved above
}
```

リスト 3 の JavaScript は platformData という配列を定義しています。この配列の各オブジェクトは個々のプラットフォームを記述するメタデータを表します。

drawPlatforms() 関数は Canvas のコンテキストの strokeRect() メソッドと fillRect() メソッドを使用してプラットフォームの四角形を描画します。これらの四角形の特性 (platformData 配列のオブジェクトに保存されています) を使用してコンテキストの塗りつぶしスタイルを設定し、また globalAlpha 属性も設定します (globalAlpha 属性は今後キャンバス内に描画されるすべての要素の透明度を設定します)。

context.translate() を呼び出すと、キャンバスの座標系 (図 2) は指定されたピクセル数だけ水平方向に移動した状態に変換されます。この変換と属性設定は context.save() の呼び出しと context.restore() の呼び出しの間に行われるため、一時的なものです。

図 2. Canvas のデフォルトの座標系



デフォルトで、座標系の原点はキャンバスの左上隅にあります。context.translate() を使用すると座標系の原点を移動することができます。

context.translate() を使用して背景をスクロールする方法については「背景をスクロールする」で説明します。しかし現在の時点で、Snail Bait を実装するために HTML5 の Canvas について知っておかなければならないことは、ほとんどすべて理解できたことになります。この連載のこれから先では、HTML5 によるゲーム開発の他の側面に焦点を絞ることにします。まず始めにアニメーションを取り上げます。

HTML5 によるアニメーション

基本的に、アニメーションの実装は単純です。画像のシーケンスを繰り返し描画し、物体が何らかの動きをしているように見せればよいのです。つまり、画像を一定期間ごとに描画するループを実装する必要があるということです。

従来、アニメーションのループは JavaScript の setTimeout() を使用するか、リスト 4 に示す setInterval() を使用して実装されていました。

リスト 4 setInterval() を使用してアニメーションを実装する

```
setInterval( function (e) { // Don't do this for time-critical animations
   animate();               // A function that draws the current animation frame
}, 1000 / 60);              // Approximately 60 frames/second (fps)
```

ベスト・プラクティス

タイム・クリティカルなアニメーションには決して `setTimeout()` や `setInterval()` を使用してはなりません。

リスト 4 のコードは確かに、次のアニメーション・フレームを描画する `animate()` 関数を繰り返し呼び出すことによってアニメーションを作成します。ただし、`setInterval()` と `setTimeout()` はアニメーションに関して何も認識しているわけではないため、結果は満足できるものではない可能性があります。(注: `animate()` 関数を実装する必要がありますが、この関数は Canvas API には含まれていません。)

リスト 4 では間隔を 1000/60 ミリ秒に設定しており、これは毎秒約 60 フレームということです。この数値は最適フレーム・レートと思える値を私が推測した結果であり、それほど適切な数値ではないかもしれません。しかし `setInterval()` と `setTimeout()` はアニメーションに関して何も認識するわけではないため、私がフレーム・レートを指定することになります。本来は私ではなくブラウザーがフレーム・レートを指定した方が適切です。次のアニメーション・フレームをいつ描画すべきかについては、私よりもブラウザーの方が確実に認識しているからです。

`setInterval()` と `setTimeout()` を使用することには、さらに深刻な欠点があります。これらのメソッドに時間間隔を渡す場合はミリ秒で時間を指定しますが、これらのメソッドにはミリ秒単位の精度がありません。実際、HTML の仕様によれば、これらのメソッドは (リソースを節約するために)、指定された時間間隔を「気前よく水増し」する可能性があるのです。

こうした欠点を避けるために、タイム・クリティカルなアニメーションには `setTimeout()` と `setInterval()` を使用するべきではありません。これらの代わりに `requestAnimationFrame()` を使用する必要があります。

`requestAnimationFrame()`

W3C は「Timing control for script-based animations (スクリプト・ベースのアニメーションのタイミング制御)」仕様 (「参考文献」を参照) の中で、`window` オブジェクトの `requestAnimationFrame()` というメソッドを定義しています。`setTimeout()` や `setInterval()` とは異なり、`requestAnimationFrame()` はアニメーションの実装専用です。そのため `requestAnimationFrame()` は `setTimeout()` や `setInterval()` に付随する欠点が何もありません。またリスト 5 に示すように `requestAnimationFrame()` は使い方も簡単です。

リスト 5. `requestAnimationFrame()` を使用してアニメーションを実装する

```
function animate(time) {           // Animation loop
   draw(time);                     // A function that draws the current animation frame
   requestAnimationFrame(animate); // Keep the animation going
};

requestAnimationFrame(animate);    // Start the animation
```

コールバック関数への参照を `requestAnimationFrame()` に渡しておけば、次のアニメーション・フレームを描画するためのブラウザーの準備が整うと、`requestAnimationFrame()` はそのコールバック関数を呼び出します。アニメーションを継続するために、そのコールバック関数でも `requestAnimationFrame()` を呼び出します。

リスト 5 を見るとわかるように、ブラウザーはコールバック関数に引数として `time` を渡します。皆さんは、引数 time が具体的に何を意味するのか疑問に思うかもしれません。`time` は現在の時刻なのでしょうか？それともブラウザーが次のアニメーション・フレームを描画する時刻なのでしょうか？

驚くべきことに、この時刻には決まった定義がありません。確実なことは、どのようなブラウザーの場合にも、この時刻は常に同じものを表す、ということのみです。つまり、この time を使用してフレーム間の経過時間を計算することができます。これについては「アニメーションのフレーム・レート (フレーム/秒: fps) を計算する」で説明します。

`requestAnimationFrame()` のポリフィル

多くの点で、HTML5 はプログラマーにとっての理想郷です。HTML5 を使用することで、開発者は独自仕様の API から解放され、どのブラウザーでもプラットフォームに依存せずに実行されるアプリケーションを実装することができます。さまざまな仕様が急速に進化しており、新しい技術が常に取り入れられ、既存の機能が改善されています。

ポリフィル: 将来に向けてのプログラミング

従来、クロスプラットフォームのソフトウェアの大部分は、どのプラットフォームにも共通に備わっている最低限の機能に合わせて実装されていました。ポリフィルはその考え方を逆にし、高度な機能が利用できる場合にはその機能を利用し、それができない場合には低機能の実装を利用するようにします。

その一方で、ブラウザー固有の機能として存在している新しい技術が仕様化されることはよくあります。ブラウザー・ベンダーは多くの場合、それらの機能に接頭辞を付け、別のブラウザーの実装に影響を与えないようにします。例えば `requestAnimationFrame()` は当初、Mozilla で `mozRequestAnimationFrame()` として実装されました。その後、WebKit で実装されましたが、WebKit はその関数の名前を `webkitRequestAnimationFrame()` としました。そして最後に、W3C が `requestAnimationFrame()` として標準化しました。

ベンダーが実装に接頭辞を追加したり、標準実装のサポートがまちまちであったりすると、新しい機能を使用するのが難しくなります。そこで、HTML5 コミュニティーは「ポリフィル」と呼ばれるものを考え出しました。ポリフィルは特定の機能に対するブラウザーのサポート・レベルを判断し、その機能がブラウザーに実装されている場合にはその機能を直接利用できるようにし、ブラウザーに実装されていない場合には標準の機能を最大限模倣した間に合わせの実装を使用できるようにします。

ポリフィルを使用するのは簡単ですが、実装は複雑な場合があります。リスト 6 は `requestAnimationFrame()` に対してポリフィルを実装した場合を示しています。

リスト 6. `requestNextAnimationFrame()` のポリフィル

```
// Reprinted from Core HTML5 Canvas

window.requestNextAnimationFrame =
   (function () {
      var originalWebkitRequestAnimationFrame = undefined,
          wrapper = undefined,
          callback = undefined,
          geckoVersion = 0,
          userAgent = navigator.userAgent,
```

```
         index = 0,
         self = this;

      // Workaround for Chrome 10 bug where Chrome
      // does not pass the time to the animation function

      if (window.webkitRequestAnimationFrame) {
         // Define the wrapper

         wrapper = function (time) {
           if (time === undefined) {
              time = +new Date();
           }
           self.callback(time);
         };

         // Make the switch

         originalWebkitRequestAnimationFrame = window.webkitRequestAnimationFrame;

         window.webkitRequestAnimationFrame = function (callback, element) {
            self.callback = callback;

            // Browser calls the wrapper and wrapper calls the callback

            originalWebkitRequestAnimationFrame(wrapper, element);
         }
      }

      // Workaround for Gecko 2.0, which has a bug in
      // mozRequestAnimationFrame() that restricts animations
      // to 30-40 fps.

      if (window.mozRequestAnimationFrame) {
         // Check the Gecko version. Gecko is used by browsers
         // other than Firefox. Gecko 2.0 corresponds to
         // Firefox 4.0.

         index = userAgent.indexOf('rv:');

         if (userAgent.indexOf('Gecko') != -1) {
            geckoVersion = userAgent.substr(index + 3, 3);

            if (geckoVersion === '2.0') {
               // Forces the return statement to fall through
               // to the setTimeout() function.

               window.mozRequestAnimationFrame = undefined;
            }
         }
      }

      return window.requestAnimationFrame   ||
         window.webkitRequestAnimationFrame ||
         window.mozRequestAnimationFrame    ||
         window.oRequestAnimationFrame      ||
         window.msRequestAnimationFrame     ||

         function (callback, element) {
            var start,
                finish;


            window.setTimeout( function () {
               start = +new Date();
               callback(start);
               finish = +new Date();
```

```
                self.timeout = 1000 / 60 - (finish - start);

            }, self.timeout);
        };
    }
)
();
```

ポリフィルの定義

ポリフィル (polyfill) という言葉はポリモーフィズム (polymorphism: 多態性) とバックフィル (backfill: 埋め戻し) の混成語です。ポリモーフィズムと同様、ポリフィルは要求されている機能にふさわしいコードを実行時に選択し、そうしたコードがない場合は元々用意されていた間に合わせの機能を使用します。

リスト 6 で実装されているポリフィルは、`requestNextAnimationFrame()` という関数を `window` オブジェクトに追加しています。関数名に「`Next`」を含めることにより、その基になる `requestAnimationFrame()` 関数と区別しています。

ポリフィルが `requestNextAnimationFrame()` に割り当てる関数は、ブラウザーが `requestAnimationFrame()` をサポートしている場合は `requestAnimationFrame()` であり、そうでない場合はベンダーが接頭辞を付けた実装です。そのどちらもブラウザーがサポートしていない場合、その関数は `setTimeout()` を使用して可能な限り `requestAnimationFrame()` を模倣した間に合わせの実装になります。

ポリフィルの複雑な部分のほぼすべてが 2 つのバグへの対処に関する部分であり、`return` 文の前にあるコードはこのバグ対策のコードで構成されています。1 つ目のバグは、時刻に対して未定義の値 (`undefined`) を渡すという Chrome 10 のバグです。2 つ目のバグは、フレーム・レートが毎秒 35 から 40 フレームに制限される Firefox 4.0 のバグです。

`requestNextAnimationFrame()` のポリフィルの実装は興味深いですが、この実装を理解する必要はなく、必要なことはポリフィルの使い方を理解することのみです。それを次のセクションで説明します。

ゲーム・ループ

グラフィックスとアニメーションの前提となる事項は説明したので、今度は Snail Bait に動きを与えます。まず、このゲームの HTML の中に `requestNextAnimationFrame()` のための JavaScript を読み込みます (リスト 7)。

リスト 7. HTML

```
<html>
   ...

   <body>
      ...

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

リスト 8 はゲームのアニメーションのループを示しています。このループは一般に「ゲーム・ループ」と呼ばれます。

リスト 8. ゲーム・ループ

```
var fps;

function animate(now) {
   fps = calculateFps(now);
   draw();
   requestNextAnimationFrame(animate);
}

function startGame() {
   requestNextAnimationFrame(animate);
}
```

`startGame()` 関数は背景画像の `onload` イベント・ハンドラーによって呼び出され、`requestNextAnimationFrame()` のポリフィルを呼び出すことでゲームを開始します。ゲームの最初のアニメーション・フレームを描画する時になると、ブラウザーが `animate()` 関数を呼び出します。

animate() 関数は現在の時刻を基に、アニメーションのフレーム・レートを計算します (time の値の詳細は「`requestAnimationFrame()`」を参照)。フレーム・レートを計算した後、animate() は `draw()` 関数を呼び出し、`draw()` 関数が次のアニメーション・フレームを描画し、さらに `requestNextAnimationFrame()` を呼び出してアニメーションを継続させます。

アニメーションのフレーム・レート (フレーム/秒: fps) を計算する

リスト 9 は、Snail Bait がフレーム・レートを計算する方法と、フレーム・レートの読み取り値 (図 1) を更新する方法を示しています。

リスト 9. フレーム・レート (フレーム/秒: fps) を計算し、`fps` 要素を更新する

```
var lastAnimationFrameTime = 0,
    lastFpsUpdateTime = 0,
    fpsElement = document.getElementById('fps');

function calculateFps(now) {
   var fps = 1000 / (now - lastAnimationFrameTime);
   lastAnimationFrameTime = now;

   if (now - lastFpsUpdateTime > 1000) {
      lastFpsUpdateTime = now;
      fpsElement.innerHTML = fps.toFixed(0) + ' fps';
   }

   return fps;
}
```

このフレーム・レートは、最後のアニメーション・フレームからの経過時間にすぎません。そのため、読者のなかには「このフレーム・レートは、複数のフレームから計算した値ではなく、1 つのフレームに対する値でしかないため、とてもレートと呼べるような代物ではない」と思う人もいるはずです。もっと厳密な方法を使用し、数フレームにわたる平均フレーム・レートにすることもできますが、それが必要とも思えませんでした。実際、この後にある「タイム・ベースで

動かす」セクションでは、まさに最後のアニメーション・フレームからの経過時間が必要となります。

リスト 9 には、重要なアニメーション手法も示してあります。つまりアニメーションでは、アニメーションの速度とは異なる速度でタスクを実行する必要があります。フレーム・レートの読み取り値をアニメーションのフレームごとに更新すると、読み取り値はいつも流動的になってしまうため、表示された値は読むことができなくなります。ここではフレームごとではなく、1 秒に 1 度、読み取り値を更新しています。

ゲーム・ループの作成が終わり、フレーム・レートを計算できたので、今度は背景のスクロールに移ります。

背景をスクロールさせる

Snail Bait の背景 (図 3) はゆっくりと水平方向にスクロールします。

図 3. 背景画像

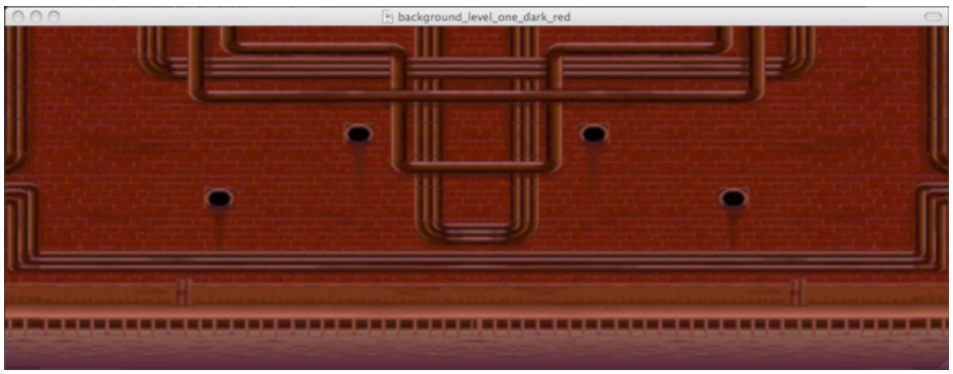
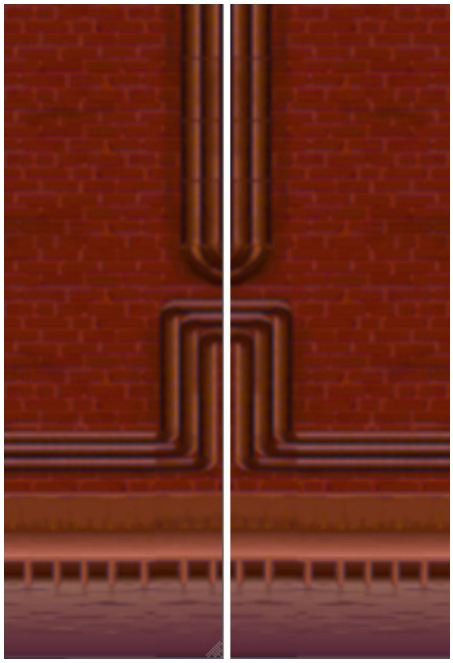


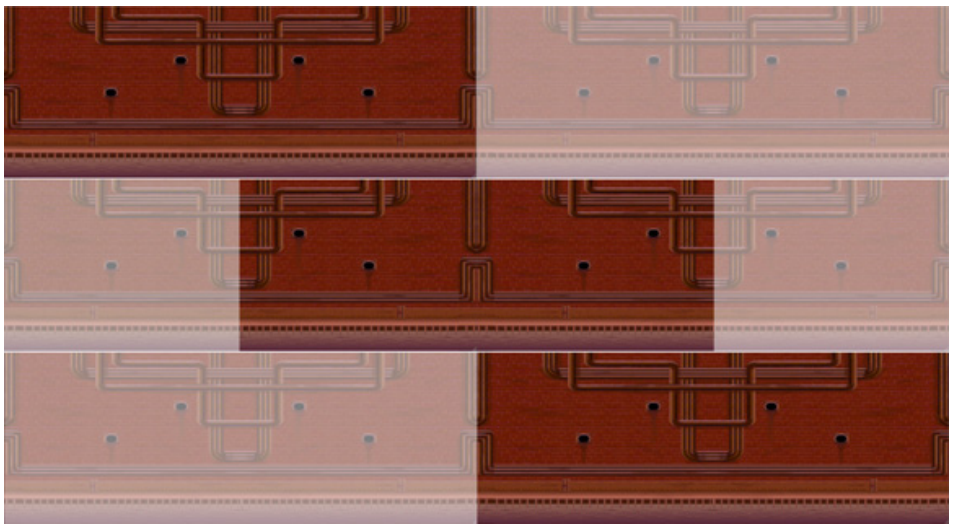
図 4 に示すように背景画像は左端と右端がうまくつながるようになっているため、背景のスクロールはシームレスに行われます。

図 4. 左端と右端がうまくつながるようになっているためスムーズに遷移する背景画像 (左側が背景画像の右端で、右側が左端)



Snail Bait では、図 5 に示すように背景を 2 度描画することによって、背景が切れ目なくスクロールするようにしています。最初は図 5 の上段のスクリーンショットに示すように、左側の背景画像が画面全体に表示され、右側の背景画像は画面にはまったく表示されていません。時間の経過と共に背景はスクロールしていき、図 5 の中段、そして下段のスクリーンショットのように画面に表示される部分が変わっていきます。

図 5. 右から左にスクロールする (半透明の領域は、画面に表示されない画像部分を示す)



リスト 10 に図 5 と関係するコードを示します。`drawBackground()` 関数は背景画像を 2 度描画し、そのそれぞれの位置は常に同じです。スクロールして見えるのは、キャンバスの座標系を

常に左へと変換している結果であり、これによって背景は右にスクロールしているように見えます。

(左に変換しているのに右にスクロールする、と言うと明らかに矛盾しているように思えますが、これは以下のように考えると理解することができます。キャンバスは、横長の紙の上に置かれた額縁であると考えてみてください。この横長の紙が座標系であり、座標系を左に変換することは額縁 (キャンバス) の下で座標系を左にスライドさせるようなものです。そのため、キャンバスは右に移動しているように見えます。)

リスト 10. 背景をスクロールさせる

```
var backgroundOffset; // This is set before calling drawBackground()

function drawBackground() {
   context.translate(-backgroundOffset, 0);

   // Initially onscreen:
   context.drawImage(background, 0, 0);

   // Initially offscreen:
   context.drawImage(background, background.width, 0);

   context.translate(backgroundOffset, 0);
}
```

`setBackground()` 関数はキャンバスのコンテキストを `-backgroundOffset` ピクセル分、水平方向に変換します。`backgroundOffset` の値が正の場合には、背景が右にスクロールし、負の場合には、背景が左にスクロールします。

背景を変換した後、`drawBackground()` は背景画像を 2 度描画します。そしてコンテキストを逆変換し、`drawBackground()` が呼び出される前の場所にコンテキストを戻します。

一見、簡単な計算 (つまり `backgroundOffset` の計算) が残っています。この計算により、各アニメーション・フレームに対してキャンバスの座標系をどの程度変換するのかを決定します。この計算自体は実際に簡単ですが、非常に重要な意味を持っています。そこで、この計算について次に説明します。

タイム・ベースで動かす

アニメーションのフレーム・レートは変化しますが、その変化するフレーム・レートによってアニメーションの進行速度が影響を受けるようであってはなりません。例えば、Snail Bait はアニメーションの基本フレーム・レートに関わらず、毎秒 42 ピクセルの速さで背景をスクロールさせます。アニメーションはタイム・ベースでなければなりません。つまり 1 秒あたりのピクセル数で速度を指定する必要があり、速度がフレーム・レートに依存するようであってはなりません。

タイム・ベースの動きを使用して、指定フレームに対する物体の移動ピクセル数を計算するのは簡単です。それには、速度を現在のフレーム・レートで割ればよいのです。速度 (ピクセル数/秒) をフレーム・レート (フレーム数/秒) で割ると、(ピクセル数/フレーム) が求まります。つまり現在のフレームに対して物体を移動しなければならないピクセル数が得られます。

ベスト・プラクティス

アニメーションの速度はフレーム・レートに依存してはなりません。

リスト 11 に、Snail Bait がタイム・ベースの動きを使用して背景のオフセットをどのように計算しているかを示します。

リスト 11. 背景のオフセットを設定する

```
var BACKGROUND_VELOCITY = 42, // pixels / second
    bgVelocity = BACKGROUND_VELOCITY;

function setBackgroundOffset() {
   var offset = backgroundOffset + bgVelocity/fps; // Time-based motion

   if (offset > 0 && offset < background.width) {
      backgroundOffset = offset;
   }
   else {
      backgroundOffset = 0;
   }
}
```

`setBackgroundOffset()` 関数は、背景の速度を現在のフレーム・レートで割ることで、現在のフレームに対する背景の移動ピクセル数を計算しています。そして、その値を現在の背景のオフセットに足しています。

連続的に背景をスクロールさせるために、`setBackgroundOffset()` では、背景のオフセットが 0 未満になるか、背景の幅よりも大きくなると、背景のオフセットを `0` にリセットします。

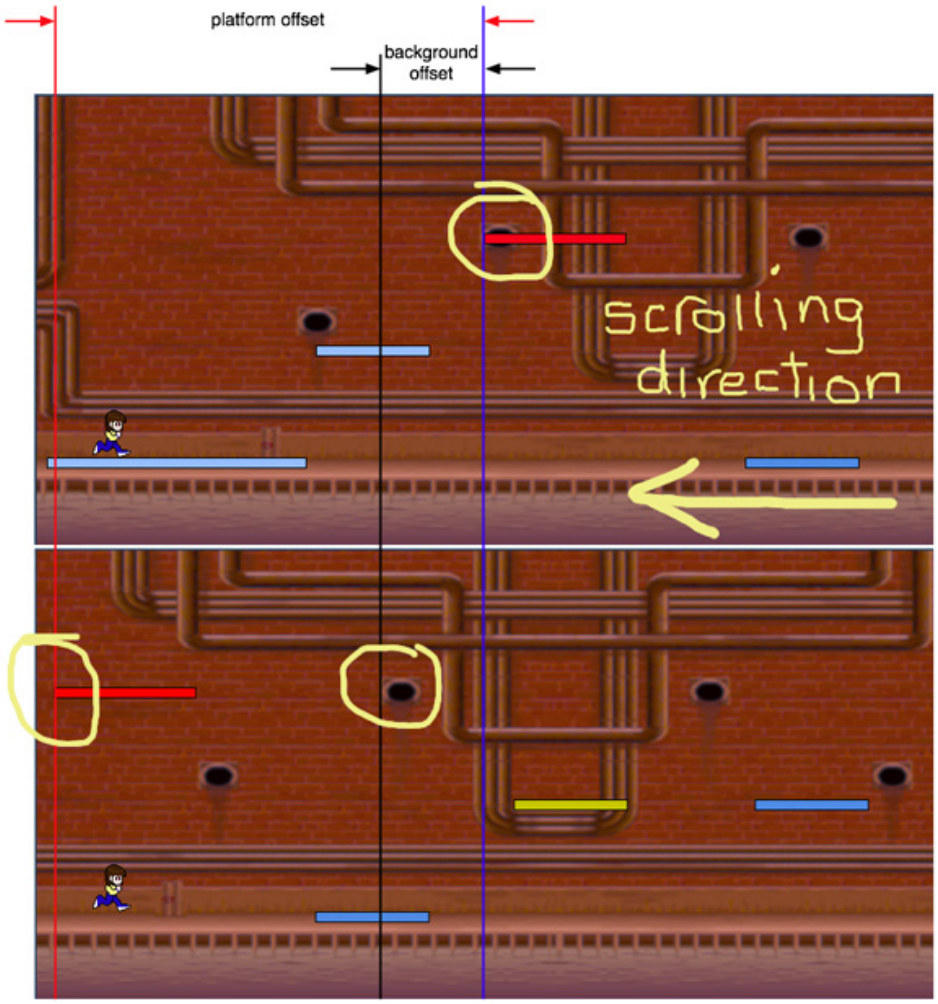
パララックス

移動している車のシートに座って、高速で迫りくる電柱を手刀でカットしたことのある人は、近くにあるものは遠くにあるものよりも速く動くことを知っているはずです。これは「視差」(パララックス) として知られています。

Snail Bait は 2D のプラットフォーム・ゲームですが、穏やかなパララックス効果を使用することで、あたかもプラットフォームが背景よりも近くにあるかのように見せています。このゲームでは、背景よりもプラットフォームを明らかに速くスクロールさせることで、そのパララックスを実装しています。

図 6 は Snail Bait でのパララックスの実装方法を示しています。上段のスクリーンショットはある特定の時点の背景を示しており、それから数フレーム後のアニメーションの背景を示したものが下段のスクリーンショットです。この 2 つのスクリーンショットから、同じ時間内にプラットフォームの方が背景よりもはるかに長い距離を移動したことがわかります。

図 6. パララックス: (近くの) プラットフォームの方が (遠くの) 背景よりも速くスクロールする様子



リスト 12 に、プラットフォームの速度およびオフセットを設定する関数を示します。

リスト 12. プラットフォームの速度およびオフセットを設定する

```
var PLATFORM_VELOCITY_MULTIPLIER = 4.35;

function setPlatformVelocity() {
   // Platforms move 4.35 times as fast as the background
   platformVelocity = bgVelocity * PLATFORM_VELOCITY_MULTIPLIER;
}

function setPlatformOffset() {
   platformOffset += platformVelocity/fps; // Time-based motion
}
```

Snail Bait のゲーム・ループを記述したリスト 8 を思い出してください。このループは `animate()` 関数で構成されており、ブラウザーはゲームの次のアニメーション・フレームを描画する時になると、この関数を呼び出します。その `animate()` 関数が今度は `draw()` 関数を呼び出し、その `draw()` 関数が次のアニメーション・フレームを描画します。この段階での `draw()` 関数のコードをリスト 13 に示します。

リスト 13. `draw()` 関数

```
function setOffsets() {
   setBackgroundOffset();
   setPlatformOffset();
}

function draw() {
   setPlatformVelocity();
   setOffsets();

   drawBackground();

   drawRunner();
   drawPlatforms();
}
```

`draw()` 関数はプラットフォームの速度と、背景とプラットフォーム両方のオフセットを設定します。そして背景、ランナー、プラットフォームを描画します。

次回は

次回の記事では、Snail Bait のコードを JavaScript オブジェクトの中にカプセル化し、名前空間の衝突を回避する方法について説明します。また、ゲームを中断する方法について、ウィンドウがフォーカスを失った場合にゲームを自動的に中断する方法と、ウィンドウが再度フォーカスを得た場合にカウントダウンを使用してゲームを再開する方法を含めて説明します。さらに、ゲームのランナーをキーボードで制御する方法についても説明します。それらの説明の中で、CSS のトランジションの使い方や、ゲーム・ループの途中に機能を追加する方法についても学びます。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-game2.zip	737KB

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。