HTML5 による 2D ゲームの開発: 時間を操作する、第 1 回

リニアな動きのジャンプを実装する

David Geary
Author and speaker
Clarity Training, Inc.

2013年 6月 06日

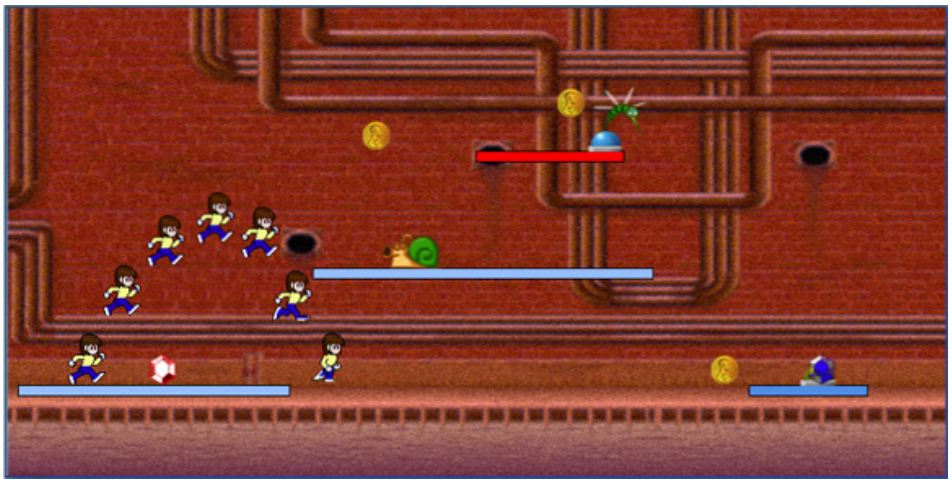
この連載では、HTML5 のエキスパートである David Geary が、HTML5 で 2D テレビ・ゲームを実装する方法について順を追って説明します。今回の記事では、時間を操作する方法について 2 回にわたって説明する第 1 回目として、ランナー・スプライトのジャンプ・ビヘイビアを実装します。

このシリーズの他の記事を見る

この連載の前回の記事では、スプライトが実行するアクション (走る、落下する、横にゆっくり動く、爆発する、など) をビヘイビアと呼ばれるプラガブル・オブジェクトの中にカプセル化する方法を説明しました。カプセル化することで、実行時にどのスプライトにも自分が望む複数のビヘイビアをセットにして簡単に持たせることができます。カプセル化にはさまざまなメリットがありますが、このような柔軟性がゲームにもたらすさまざまな側面は、そのまま眠らせておかずに探ってみる価値があります。

この記事では、スプライトのビヘイビアについての説明を続けますが、前回の記事とは次の 2 つの点が異なっています。第 1 に、今回と次回の記事では 2 回続けて、スプライトのビヘイビアの 1 つである、ランナーのジャンプ・ビヘイビアのみを取り上げます。この記事はその第 1 回目ですが、「時間を操作する、第 2 回」の最後になると、Snail Bait には図 1 に示す自然なジャンプ・シーケンスが実現されます。

図 1. 自然なジャンプ・シーケンス



衝突検出は後回し

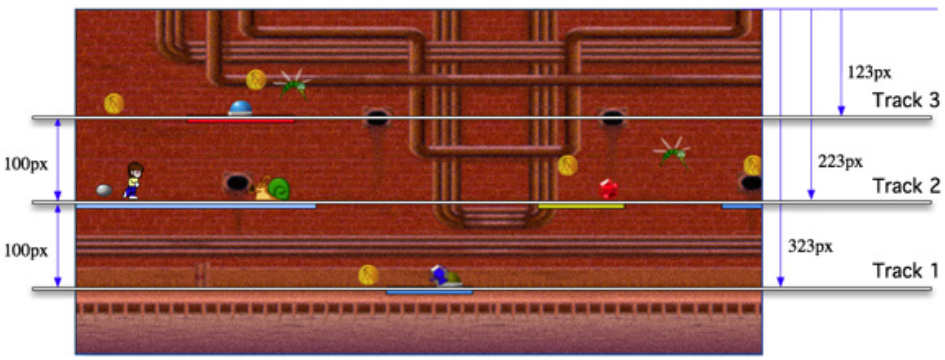
ここでは Snail Bait における衝突検出についての説明は後回しにし、ランナーの動きにフォーカスして説明しています。衝突検出を実装すると、ランナーがプラットフォームに着地した時点でジャンプの動作は打ち切られます。衝突検出がない場合は、ジャンプの動作の最後までジャンプは継続されます。ジャンプのエフェクト全体を確認するには、この記事のコードをダウンロードし、皆さん自身で試してみてください。

第 2 に、ジャンプ・ビヘイビアは、前回の記事で説明したビヘイビアとは異なり、無限に繰り返されるわけではありません。この単純な違いにより、Snail Bait はジャンプの進行に合わせて時間を追跡する必要があります。そのためにはストップウォッチのようなものが必要です。そこで JavaScript でストップウォッチを実装し、ランナーがジャンプする際の上昇時間や下降時間をそのストップウォッチを使用して測定することにします。

ランナーのトラックとプラットフォームの最上部

Snail Bait のプラットフォームは 3 つのトラックに沿って水平に移動します (図 2)。

図 2. プラットフォームのトラック



トラック間の間隔は 100 ピクセルです。ランナーの高さは 60 ピクセルなので、ランナーが動作するには十分なスペースです。

リスト 1 は、Snail Bait がランナーの高さとプラットフォームの垂直方向の位置をどのように設定しているかを示しています。さらにこのリストには、`calculatePlatformTop()` というコンビニエンス・メソッドも記載してあります。トラック (1、2、3 のいずれか) が指定されると、このメソッドは指定されたトラックのベースラインを返します。

リスト 1. トラックのベースラインからプラットフォームの最上部位置を計算する

```
var SnailBait = function () {
   // Height of the runner's animation cells:

   this.RUNNER_CELLS_HEIGHT = 60, // pixels

   // Track baselines:

   this.TRACK_1_BASELINE = 323, // pixels
   this.TRACK_2_BASELINE = 223,
   this.TRACK_3_BASELINE = 123,
   ...
};
...

SnailBait.prototype = {
   ...
calculatePlatformTop: function (track) {
      var top;

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

      return top;
   ...
};
```

Snail Bait はゲームに登場するほぼすべてのスプライトの位置を指定する際に、`calculatePlatformTop()` を使用しています。

ジャンプの最初の実装

前回の記事が終わった時点で実装されている、Snail Bait のジャンプのアルゴリズムは非常に単純なものです (リスト 2)。

リスト 2. ジャンプのキーボード操作の処理

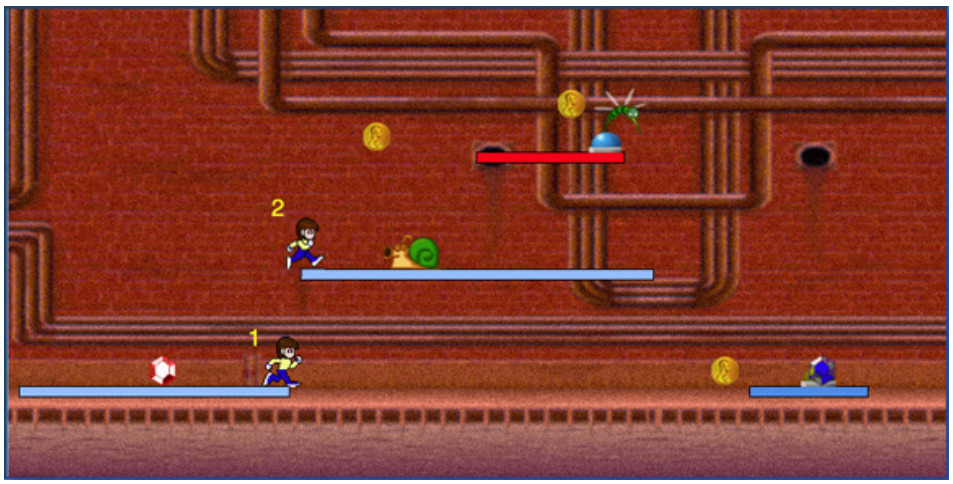
```
window.onkeydown = function (e) {
   var key = e.keyCode;
   ...

   if (key === 74) { // 'j'
      if (snailBait.runner.track === 3) { // At the top; nowhere to go
         return;
      }

      snailBait.runner.track++;
      snailBait.runner.top = snailBait.calculatePlatformTop(snailBait.runner.track) -
                             snailBait.RUNNER_CELLS_HEIGHT;
   }
};
...
```

プレイヤーが「j」キーを押すと、(ランナーが一番上のトラックにいる場合を除き) Snail Bait は即座に、ランナーの上部にあるトラックにランナーを配置します (図 3)。

図 3. ぎこちないジャンプ・シーケンス: 実装は簡単ですが、不自然です



リスト 2 に示したジャンプの実装には、深刻な欠陥が 2 つあります。第 1 に、ランナーが瞬時に上へ移動してしまうのは、とても望ましいエフェクトとは言えません。第 2 に、ジャンプの実装の抽象化レベルが不適切です。ウィンドウ・イベント・ハンドラーは、ランナーの属性を直接操作するためのものではないため、ランナー・オブジェクト自体がジャンプを実装する必要があります。

ジャンプの実装をランナー・オブジェクトへ移す

リスト 3 は、ウィンドウの `onkeydown` イベント・ハンドラーの実装をリファクタリングしたものです。この実装はリスト 2 の実装よりもはるかに単純であり、ジャンプの実装がイベント・ハンドラーからランナー・オブジェクトへ移されています。

リスト 3. ランナー・オブジェクトへ処理を委譲した、ウィンドウのキー・ハンドラー

```
window.onkeydown = function (e) {
   var key = e.keyCode;
   ...

   if (key === 74) { // 'j'
      runner.jump();
   }
};
```

ゲームが開始されると、Snail Bait は `equipRunner()` というメソッドを呼び出します (リスト 4)。

リスト 4. ゲームの開始時にランナー・オブジェクトを準備する

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

`equipRunner()` メソッド (リスト 5) では、ランナー・オブジェクトに属性と `jump()` メソッドを追加します。

リスト 5. ランナー・オブジェクトを準備する: ランナー・オブジェクトの `jump()` メソッド

```
SnailBait.prototype = {
equipRunner: function () {
      // This function sets runner attributes:

      this.runner.jumping = false; // 'this' is snailBait
      this.runner.track = this.INITIAL_RUNNER_TRACK;

      ... // More runner attributes omitted for brevity

      // This function also implements the runner's jump() method:

      this.runner.jump = function () {
         if ( ! this.jumping) {    // 'this' is the runner.
this.jumping = true; // Start the jump
         }
      };
   },
},
```

ビューとコントローラー

ランナーのジャンプ・ビヘイビアとそれに対応する jump() メソッドは、ビューとコントローラーのペアに似ています。Snail Bait では、ジャンプ中のランナーを描画する手段はビヘイビアの中で実装されます。一方、ランナー・オブジェクトの jump() メソッドはランナーが現在ジャンプ中かどうかを制御するための単純なコントローラーとして動作します。

ランナー・オブジェクトには、ランナーの現在のトラックを表す属性や、ランナーが現在ジャンプ中かどうかを表す属性など、さまざまな属性があります。

ランナーが現在ジャンプ中ではない場合、`runner.jump()` メソッドは単にランナー・オブジェクトの `jumping` 属性を `true` に設定します。Snail Bait では、ジャンプという動作を個別のビヘイビア・オブジェクトの中に実装します。この方法はランナーのすべてのビヘイビア (走る、落下する、など) で同じであり、さらにはすべてのスプライトのビヘイビアで同じです。Snail Bait ではランナーを作成するときに、ランナーのビヘイビア配列にジャンプのビヘイビア・オブジェクトを追加します (リスト 6)。

リスト 6. ランナーをそのビヘイビアとともに作成する

```
var SnailBait = function () {
   ...
   this.jumpBehavior = {
      execute: function(sprite, time, fps) {

         // Implement jumping here

      },
      ...
   };
   ...

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

これでジャンプを開始するための土台が準備できたので、ジャンプ・ビヘイビアのみにフォーカスすることができます。

ジャンプ・ビヘイビア

リスト 7 は、ランナーのジャンプ・ビヘイビアを最初に実装したコードを示しており、この実装はリスト 2 のコードと機能は同じです。ランナー・オブジェクトの `jumping` 属性 (ランナー・オブジェクトの `jump()` メソッドによって設定されます (リスト 5 を参照)) が `false` の場合、ジャンプ・ビヘイビアは何もしません。またランナーが一番上のトラックにいる場合も、ジャンプ・ビヘイビアは何もしません。

リスト 7. 現実的ではないジャンプ・ビヘイビアの実装

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

   this.jumpBehavior = {
      ...

execute: function(sprite, time, fps) {
         if ( ! sprite.jumping || sprite.track === 3) {
            return;
         }

         sprite.track++;

         sprite.top = snailBait.calculatePlatformTop(sprite.track) -
                      snailBait.RUNNER_CELLS_HEIGHT;

         sprite.jumping = false;
      }
   },
   ...
};
```

無限ループ

Snail Bait は基本的に無限ループであり、表示されているすべてのスプライトのすべてのビヘイビアを常に実行している、ということを思い出してください。ランナー・オブジェクトの

`jump()` メソッドは、単にランナー・オブジェクトの `jumping` 属性を `true` に設定することによって、ジャンプを開始します。次回 Snail Bait がランナーのジャンプ・ビヘイビアを実行する際には、この設定によってジャンプ・ビヘイビアが実行されます。

ランナーがジャンプ中で、一番上のトラックにはいない場合、リスト 7 で実装したジャンプ・ビヘイビアはランナーを次のトラックへ移動し、ランナー・オブジェクトの `jumping` 属性を `false` に設定してジャンプを完了させます。

リスト 2 で実装したジャンプとまったく同じように、リスト 7 の実装は、現在ランナーがいるトラックから別のトラックへとランナーを瞬時に移動します。ジャンプの動きをリアルにするには、一定の時間をかけて少しずつ、現在ランナーがいるトラックから別のトラックへとランナーを移動する必要があります。

アニメーションの時間を測定する: ストップウォッチ

ここまで Snail Bait に実装した動きは、すべて一定でした。例えば、ゲームのスプライトは、ランナー以外はすべて常に水平方向に移動しており、ボタンとカタツムリは常にそれぞれのプラットフォーム上をゆっくりと行ったり来たりしています (この動きがどのように実装されているかについては、この連載第2 回の記事の「背景をスクロールさせる」セクションを参照)。コイン、サファイア、ルビーも、停止することなく、ゆっくりと上下に動いています。

ただしジャンプの動きは一定ではなく、明確な開始と終了があります。つまりジャンプを実装するには、ジャンプを開始してからの経過時間を常にモニターする手段が必要です。そこで、ストップウォッチが必要になります。

リスト 8 に示すのは、`Stopwatch` という JavaScript オブジェクトの実装です。

リスト 8. `Stopwatch` オブジェクト

```
// Stopwatch.....................................................................
//
// You can start and stop a stopwatch and you can find out the elapsed
// time the stopwatch has been running. After you stop a stopwatch,
// its getElapsedTime() method returns the elapsed time
// between the start and stop.

Stopwatch = function ()  {
   this.startTime = 0;
   this.running = false;
   this.elapsed = undefined;

   this.paused = false;
   this.startPause = 0;
   this.totalPausedTime = 0;
};

// You can get the elapsed time while the stopwatch is running, or after it's
// stopped.

Stopwatch.prototype = {
start: function () {
      this.startTime = +new Date();
      this.running = true;
      this.totalPausedTime = 0;
      this.startPause = 0;
   },
```

```
stop: function () {
      if (this.paused) {
         this.unpause();
      }

      this.elapsed = (+new Date()) - this.startTime -
                                     this.totalPausedTime;
      this.running = false;
   },

pause: function () {
      this.startPause = +new Date();
      this.paused = true;
   },

unpause: function () {
      if (!this.paused) {
         return;
      }

      this.totalPausedTime += (+new Date()) - this.startPause;
      this.startPause = 0;
      this.paused = false;
   },

getElapsedTime: function () {
      if (this.running) {
         return (+new Date()) - this.startTime - this.totalPausedTime;
      }
      else {
        return this.elapsed;

      }
   },


isPaused: function() {
      return this.paused;
   },

isRunning: function() {
      return this.running;
   },

reset: function() {
     this.elapsed = 0;
     this.startTime = +new Date();
     this.running = false;
     this.totalPausedTime = 0;
     this.startPause = 0;
   }
};
```

リスト 8 のストップウォッチ・オブジェクトは、スタート、ストップ、一時停止、一時停止解除、リセットを実行することができます。また、経過時間を取得することや、ストップウォッチが動いているか、それとも一時停止しているかを判断することもできます。

この連載第 3 回の記事の「ゲームをフリーズさせる」セクションでは、一時中断したゲームを中断したそのままの状態から再開するために、ゲームが中断されている時間を計算に入れる方法を説明しました。ゲームそのものと同じように、一時停止したストップウォッチは、一時停止したまさにその時点から計測を再開する必要があります。つまりストップウォッチの場合も一時停止されている時間を計算に入れます。

ストップウォッチの実装は単純ですが、非常に重要です。というのも、一定時間継続されるビヘイビア (この場合は、自然なジャンプ) の実装はストップウォッチによって実現されるからです。

ジャンプ・ビヘイビアを改善する

ストップウォッチを用意できたので、ストップウォッチを使用してジャンプ・ビヘイビアを改善します。まず、リスト 5 の `equipRunner()` メソッドをリスト 9 のように変更します。

リスト 9. 修正された `equipRunner()` メソッド

```
SnailBait.prototype = {
   ...

   this.RUNNER_JUMP_HEIGHT = 120,    // pixels
   this.RUNNER_JUMP_DURATION = 1000, // milliseconds

equipRunnerForJumping: function () {
      this.runner.JUMP_HEIGHT = this.RUNNER_JUMP_HEIGHT;
      this.runner.JUMP_DURATION = this.RUNNER_JUMP_DURATION;

      this.runner.jumping = false;


      this.runner.ascendStopwatch  = new Stopwatch(this.runner.JUMP_DURATION/2);
      this.runner.descendStopwatch = new Stopwatch(this.runner.JUMP_DURATION/2);

      this.runner.jump = function () {
         if (this.jumping) // 'this' is the runner
            return;

         this.jumping = true;
         this.runAnimationRate = 0; // Freeze the runner while jumping
         this.verticalLaunchPosition = this.top;
         this.ascendStopwatch.start();
      };
   },

equipRunner: function () {
      ...

      this.equipRunnerForJumping();
   },
   ...
};
```

修正された `equipRunner()` の実装では、新しいメソッド `equipRunnerForJumping()` を呼び出します。名前からもわかるように、このメソッドはランナーがジャンプするための準備を行います。このメソッドにより、ジャンプでの上昇用ストップウォッチ `runner.ascendStopwatch` と下降用ストップウォッチ `runner.descendStopwatch` が作成されます。

ジャンプが開始されると、`jump()` メソッドはランナーの上昇用ストップウォッチをスタートします (リスト 9)。また jump() メソッドにより、ランナーが走るアニメーションのフレーム・レート (このフレーム・レートによって、ランナーが走るアニメーションでランナーが先に進む速さが決まります) がゼロに設定され、ランナーが空中にいるときにはランナーがフリーズされます。また `run()` メソッドもランナーの垂直方向の位置を記録し、ジャンプが完了したときにランナーが元の位置に戻れるようにします。

リスト 9 で設定されるランナー・オブジェクトの属性のすべてをまとめたものが表 1 です。

表 1. ジャンプに関連する、ランナー・オブジェクトの属性

属性	説明
`JUMP_DURATION`	ジャンプに要する時間をミリ秒で表す定数。例えば、1000。
`JUMP_HEIGHT`	ジャンプの高さをピクセルで表す定数。例えば、120。この場合、ジャンプの最高点でのランナーの位置は、現在のトラックの 1 つ上のトラックよりも 20 ピクセル上となります。
`ascendStopwatch`	ランナーがジャンプで上昇している時間を測定するストップウォッチ
`descendStopwatch`	ランナーがジャンプで下降している時間を測定するストップウォッチ
`jumpApex`	ランナーがジャンプで到達する最高点。ジャンプ・ビヘイビアは、ジャンプの下降時にランナーが各フレームでどの程度降下すればよいかを、jumpApex を使用して判断します。
`jumping`	ランナーのジャンプ中は `true` に設定されるフラグ
`verticalLaunchPosition`	ジャンプを開始したときのランナーの位置 (ランナー・スプライトの左上隅)。ランナーはジャンプを完了すると、この位置に戻ります。

次に、リスト 7 で最初に実装されたジャンプ・ビヘイビアをリスト 10 のようにリファクタリングします。

リスト 10. 修正されたジャンプ・ビヘイビア

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

execute: function(sprite, context, time, fps) {
         if ( ! sprite.jumping) {
            return;
         }

         if (this.isJumpOver(sprite)) {
            sprite.jumping = false;
            return;
         }

         if (this.isAscending(sprite)) {
            if ( ! this.isDoneAscending(sprite)) this.ascend(sprite);
            else                                 this.finishAscent(sprite);
         }
         else if (this.isDescending(sprite)) {
            if ( ! this.isDoneDescending(sprite)) this.descend(sprite);
            else                                  this.finishDescent(sprite);
         }
      }
   },
   ...
```

リスト 10 のジャンプ・ビヘイビアでは、ジャンプの詳細は `ascend()` や `isDescending()` など他のメソッドに委ね、上位レベルの抽象化を実装しています。これで、あとはランナーの上昇用ストップウォッチと下降用ストップウォッチを使用して以下のメソッドを実装することにより、細部を詰めるだけです。

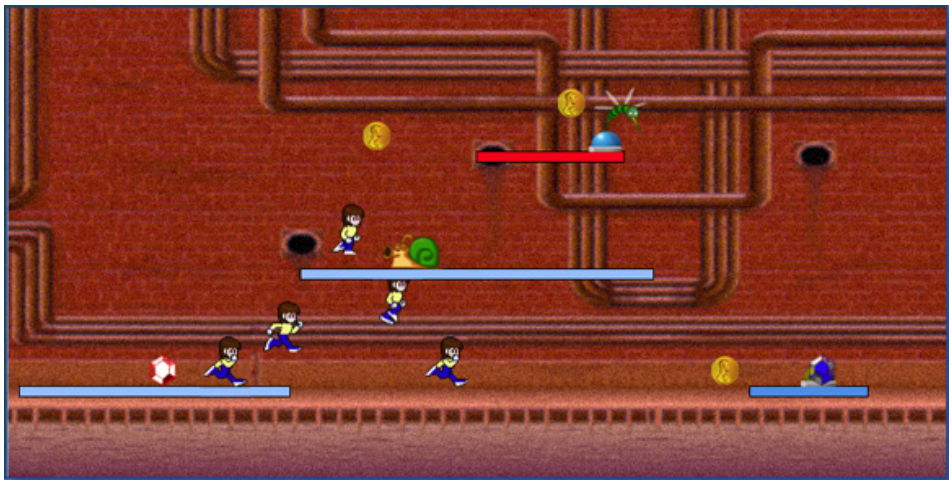
- `isJumpOver()`
- `ascend()`
- `isAscending()`

- `isDoneAscending()`
- `finishAscent()`
- `descend()`
- `isDescending()`
- `isDoneDescending()`
- `finishDescent()`

リニアな動き

今のところ、上記のメソッドによって生成されるのはリニアな動きです。つまり、ランナーは一定の速度で上昇したり、下降したりします (図 4)。

図 4. スムーズでリニアなジャンプ・シーケンス



リニアな動きでは、不自然なジャンプの動きになります。なぜなら、ランナーが上昇または下降しているときには、実際のランナーは常に重力によって減速または加速されるからです。次回の記事では、重力を考慮して実装し直すため、リニアでない動き (図 1 のような動き) になります。今の段階ではより単純な、リニアな動きのままにしておきます。

まずリスト 11 に、ジャンプ・ビヘイビアの `isJumpOver()` メソッドの実装を示します。この実装は、動きがリニアであってもリニアでなくても同じであり、どちらのストップウォッチも動いていなければ、ジャンプは終了しています。

リスト 11. ジャンプが終了しているかどうかを判断する

```
SnailBait.prototype = {
   this.jumpBehavior = {
isJumpOver: function (sprite) {
         return !sprite.ascendStopwatch.isRunning() &&
                !sprite.descendStopwatch.isRunning();
      },
      ...
   },
   ...
};
```

ジャンプ・ビヘイビアのメソッドのうち、上昇を扱うメソッドをリスト 12 に示します。

リスト 12. 上昇を扱うメソッド

```
SnailBait.prototype = {
   ...

   this.jumpBehavior = {
isAscending: function (sprite) {
         return sprite.ascendStopwatch.isRunning();
      },

ascend: function (sprite) {
         var elapsed = sprite.ascendStopwatch.getElapsedTime(),
             deltaY  = elapsed / (sprite.JUMP_DURATION/2) * sprite.JUMP_HEIGHT;

         sprite.top = sprite.verticalLaunchPosition - deltaY; // Moving up
      },

isDoneAscending: function (sprite) {
         return sprite.ascendStopwatch.getElapsedTime() > sprite.JUMP_DURATION/2;
      },

finishAscent: function (sprite) {
         sprite.jumpApex = sprite.top;
         sprite.ascendStopwatch.stop();
         sprite.descendStopwatch.start();
      }
   },
   ...
};
```

リスト 12 のメソッドをまとめたものが表 2 です。

表 2. `jumpBehavior` の上昇を扱うメソッド

メソッド	説明
`isAscending()`	ランナーの上昇用ストップウォッチが動いている場合に `true` を返します
`ascend()`	最後のアニメーション・フレームからの経過時間と、ジャンプに要する時間の設定値およびジャンプの高さの設定値を基に、ランナーを上に移動します。
`isDoneAscending()`	ランナーの上昇用ストップウォッチでの経過時間が、ジャンプに要する時間の設定値の半分を超えている場合に `true` を返します。
`finishAscent()`	ランナーの上昇用ストップウォッチをストップし、ランナーの下降用ストップウォッチをスタートして、上昇を終了します。 ランナーがジャンプの最高点に達すると、`jumpBehavior` はこのメソッドを呼び出します。そのため、`finishAscent()` はランナーの位置をランナー・オブジェクトの `jumpApex` 属性に格納します。`descend()` メソッドはその `jumpApex` 属性を使用します。

リスト 9 に示したように、ランナー・オブジェクトの `jump()` メソッドが上昇用ストップウォッチをスタートすることを思い出してください。そして、その動いているストップウォッチによって、ジャンプ・ビヘイビアの `isAscending()` メソッドが一時的に `true` を返します。リスト 10 を見るとわかるように、ランナーが上昇を終了するまで、つまりジャンプが半分終わるまで、ランナーのジャンプ・ビヘイビアは繰り返し `ascend()` メソッドを呼び出します。

上昇と下降

`ascend()` メソッドは、ランナーを少しずつ上に移動します。このメソッドは、ストップウォッチによる経過時間 (ミリ秒) を、ジャンプに要する時間の設定値の半分 (ミリ秒) で割り算し、その値にジャンプの高さの設定値 (ピクセル) を掛け算することによって、各アニメーション・フレームでランナーを移動させるピクセル数を計算します。分子と分母のミリ秒は互いに打ち消し合うため、`deltaY` の単位はピクセルになります。この値は、現在のアニメーション・フレームでランナーを垂直方向に移動するピクセル数を表します。

ランナーが上昇を終了すると、ジャンプ・ビヘイビアの `finishAscent()` メソッドがスプライトの位置をジャンプの最高点として記録し、上昇用ストップウォッチをストップし、下降用ストップウォッチをスタートします。

ジャンプ・ビヘイビアのうち、下降に関係するメソッドをリスト 13 に示します。

リスト 13. 下降を扱うメソッド

```
SnailBait.prototype = {
   this.jumpBehavior = {
isDescending: function (sprite) {
         return sprite.descendStopwatch.isRunning();
      },

descend: function (sprite, verticalVelocity, fps) {
         var elapsed = sprite.descendStopwatch.getElapsedTime(),
             deltaY  = elapsed / (sprite.JUMP_DURATION/2) * sprite.JUMP_HEIGHT;

         sprite.top = sprite.jumpApex + deltaY; // Moving down
      },

isDoneDescending: function (sprite) {
         return sprite.descendStopwatch.getElapsedTime() > sprite.JUMP_DURATION/2;
      },

finishDescent: function (sprite) {
         sprite.top = sprite.verticalLaunchPosition;
         sprite.descendStopwatch.stop();
         sprite.jumping = false;
         sprite.runAnimationRate = snailBait.RUN_ANIMATION_RATE;
      }
   },
   ...
};
```

リスト 13 のメソッドをまとめたものが表 3 です。

表 3. jumpBehavior の下降を扱うメソッド

属性	説明
`isDescending()`	ランナーの下降用ストップウォッチが動いている場合に `true` を返します。
`descend()`	最後のアニメーション・フレームからの経過時間と、ジャンプに要する時間の設定値およびジャンプの高さの設定値を基に、ランナーを下に移動します。
`isDoneDescending()`	ランナーがジャンプをする前の位置よりも下に落ちると `true` を返します。

`finishDescent()`	ランナーの下降用ストップウォッチをストップし、ランナー・オブジェクトの `jumping` フラグを `false` に設定することにより、下降を停止し、ジャンプを停止します。 下降した後、ランナーの高さはジャンプを開始した時の高さとまったく同じではないかもしれません。そのため、`finishDescent()` はランナーの位置を、ジャンプをする前の垂直方向の位置に設定します。 最後に、`finishDescent()` はランナーのアニメーションのフレーム・レートを通常の値に設定し、それによってランナーが走り始めます。

上昇のためのメソッド (リスト 12) と下降のためのメソッド (リスト 13) は多くの点で対応しています。`ascend()` と `descend()` は、現在のフレームでランナーを垂直方向に移動するためのピクセル数を、まったく同じ方法で計算します。ただし、`descend()` メソッドはそのピクセル数をジャンプの最高点の位置に足し算しますが、`ascend()` はジャンプの開始位置からそのピクセル数を引き算します (Canvas の Y 軸は上から下へと値が増加することを思い出してください)。

ジャンプの下降が終了すると、`finishDescent()` はジャンプを開始したときと同じ垂直方向の位置にランナーを戻し、走るアニメーションを再開します。

次回は

この連載の次回の記事では、リニアでない動きを実装し、図 1 に示したようなリアルな動きのジャンプにする方法を説明します。その中で、時間を操作する方法を紹介します。時間を操作することにより、時間の経過によって派生するあらゆるもの (例えば、色の変化など) に対し、リニアでないエフェクトを生み出すことができます。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-game6.zip	1.2MB

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。