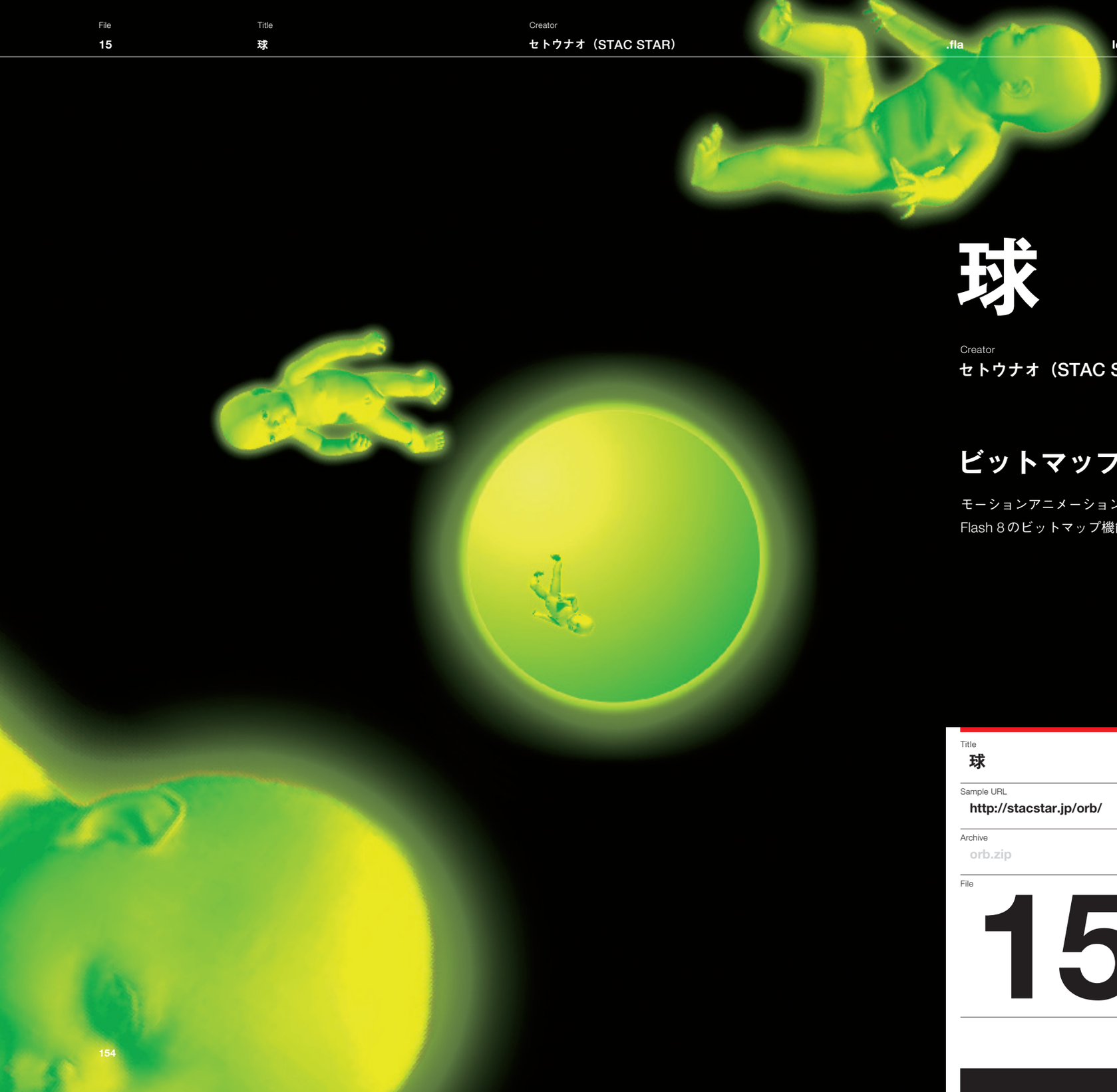


球

Creator
セトウナオ（STAC STAR）

ビットマップデータのキャッシュシング

モーションアニメーションをビットマップ化し、メモリに格納することで動作を軽くする。
Flash 8 のビットマップ機能の活用。

Title
球

Sample URL
http://stacstar.jp/orb/

Archive
orb.zip

File
15

スクリプト
ActionScript 2.0

対応プレーヤー
Flash Player 8以上

制作アプリケーション
Flash 8

発案～デザイン
コンセプトイメージを Flash 表現へ落とし込む

このコンテンツは、以前参加させていただいたエキシビジョンのために制作したものです。そのコンテンツを Flash 8 の機能を使い、改良しました。はじめに作成したときは Flash Player 7 が最新という状況でしたが、今回は Flash Player 8 の新機能である α ビデオやビットマップ機能のおかげで、大きく改善することが可能となっています。

Flash 8 の新機能で改良

○ コンセプト

このコンテンツのテーマは「球」です。はじめに考えたのは、「球」は何でできているのか？　でした。

まず、頭の中で遠くのほうに存在する巨大な「球」を想像しました。そして、頭の中でその「球」に向かってズームアップしていきました。そんな状況を想像した場合、みなさんは何が見えますか？　ボクの頭の中には、無数の「赤ん坊」が見えました。そして、少しの「カエル」も見えます。

そこで

球＝ 99% の赤ん坊＋ 1% のカエル

というコンセプトを自分の中に作りました。

このコンセプトから思い付いたイメージは、中心にめがけてあらゆる方向から赤ん坊が落下していき、だんだんと球状になっていくものでした。

○ Flash で可能な表現への落とし込み

Flash は、3D オブジェクトが扱えませんし、空間も 3D 空間ではありません。そのため、あらゆる方向から赤ん坊が集まってくる現象をコントロールするのはむずかしく、何よりパフォーマンスが追い付かないことは容易に想像できます。

そこで、球の中心を画面の中央とし、視線（カメラ）の方向から球の中心へ赤ん坊を落とすことで、オブジェクトの数を減らすことにしました。これであれば、オブジェクトの大きさで距離感を出すのも容易です。また、はじめに描いていた、赤ん坊が集まって球状の形に見せるやり方ではオブジェクトが増えてしまい、処理が重くなってしまいます。そこで、あらかじめ球を用意しておき、その球に落下したときに波紋を出すことで赤ん坊が球へ吸い込まれるようなイメージにしました。これでなんとか Flash の表現が可能になりそうです。

● Flash Player 7 への落とし込み

まずは、赤ん坊をどう表現するかを考えます。赤ん坊のモデリングとモーション付けは「Poser」を使用しています。

URL台 **Poser**
http://graphic.e-frontier.co.jp/poser/

そのモデリングしたデータを Flash のどの形式で表現するのが実現性があるでしょう？

- ベクター
- ビットマップ
- Flash ビデオ

の 3 つが考えられます。

Flash Player 7 では Flash ビデオに α チャンネルの情報がないため、透過させることができず重なり合いが表現できません。次に、リアリティでいうならビットマップになります。しかし、試してみると毎フレーム α チャンネルを持つビットマップを入れ替える処理は非常に負荷がかかり、とても耐えうるものにはなりませんでした。そこで、最後に残ったベクターでの表現になりました。

しかし、赤ん坊のような複雑なオブジェクトをベクターで表すには非常に多くのパス情報が必要となります。パス情報が多いとレンダリングの負荷が高くなるので、パスを削除していく作業が必要です。Flash でもパスを最適化させる機能がありますが、これほど多くのパスを調整するのはなかなかむずかしいです。

そこで、このときは「Optimaze!」というソフトを使いました。このソフトは、アニメーションを再生させながらパスを調整でき、調整した状態の SWF を保存することができます。このソフトを使って、何体かの赤ん坊を作成しました。

データをベクターで表現

URL台 **Optimaze!**
http://www.erain.com/products/optimaze/

あとは、その赤ん坊のモーションをステージ上に貼り付け、球に対して落下（縮小）させればほぼ完成です。

TIPS Optimaze! と Flash Optimizer

このときに使用した Optimaze! では、直線のパスとカーブのパスの値をスライダーを動かすだけで、簡単に調整することができます。オブジェクトを見ながら、リアルタイムにパスの削除が可能です。ただし、このソフトは購入したままでは日本語環境で動作しません。メールにてパッチをもらう必要があります。Optimaze! については、「Optimaze Ultimate Flash Vector Graphics Compression」、あるいは「Optimaze!」に詳しい情報があります。

Optimaze! は Flash MX の頃から開発がとまっているようです。現在では、パスの削除以外にもいろんな機能を持っていて、Optimaze! よりも安価な「Flash Optimizer」というソフトが登場しています。

URL台 **Optimaze Ultimate Flash Vector Graphics Compression**
http://www.vecta3d.com/

URL台 **Optimaze!**
http://numerous.org/bazooka/mt/archives/2003/02/15_100311.html

URL台 **Flash Optimizer(show-kit.com)**
http://show-kit.com/flash-optimizer/

*Optimaze! については、開発が終了してから年数が経っており、当時掲載していたリンクが無効となっています。

● Flash Player 8 へ改善

Flash 8 の大きな新機能の 1 つとして、α チャンネル付きのムービーが使えます。そこで、以前ベクター化していた赤ん坊のアニメーションをαビデオに置き換えてみました。しかし、αビデオのオブジェクトをステージに 4 つ、5 つと配置してみるとひどい処理落ちをしてしまいました。ただでさえ重いのに、これにカラー調整をしてしまうと、とてつもない重さになってしまいます。

対応策として、次のように考えました。Flash Player 8 版のコンテンツを見ればわかりますが、最初に「initializing bitmap data」というアニメーションが入ります。ここが Flash Player 7 とは違う流れであり、肝となる部分です。

先の案では、いくつものαビデオの再生に CPU が追いついていないため、処理落ちが起こっていました。そこで、αビデオの再生よりもレンダリングの速いビットマップデータを使うことにしました。始めのアニメーションはただの導入部分ではなく、ムービーを再生しながらその 1 フレーム 1 フレームをビットマップ化し、メモリに蓄積しています。

つまり、メモリに大量のビットマップデータを格納することで、CPU の処理を軽減させているのです。とはいえ、640×480 ドット以上のサイズの画像を 150 フレーム分もメモリに格納するので、とても速いとはいえませんが、先の方法よりも断然見られる速度になりました。

導入のアニメーションでビットマップデータをキャッシュ

スクリプト
ビットマップをキャプチャしムービーとして再生

このコンテンツは以下のクラスで成り立っています。
それぞれの役割を見てみましょう。

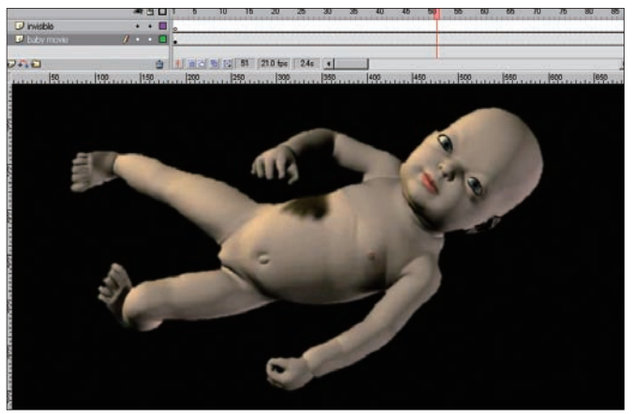
使用するクラス

クラス	概要
Baby	赤ん坊を表すクラス。ムービーではなくムービーから生成したビットマップを 1 フレームごとに貼りつけてアニメーションを実現する
BabyBitmapCreator	赤ん坊のムービーから毎フレーム、ビットマップを attachBitmap し配列内に格納する
OrbControl	各オブジェクトのイベントをもとに全体をコントロールするクラス
OrbContents	このコンテンツのステージとなるクラス
OrbEvent	赤ん坊を生成するイベントを制御するクラス
Orb	球を表すクラス
Frog	カエルを表すクラス

この中でも、ムービーからビットマップをキャプチャし、ムービーとして再生する部分のスクリプトにターゲットをあてて解説していきます。

○ ColorTransform を用いた色の変更

もともとは肌色



赤ん坊のムービーは、もともとは極力白くした肌色です。まず、flash.geom.ColorTransform クラスを用いて着色を行います。

Flash 8 より前のバージョンでは、Color クラスで MC を引数に取り、そのインスタンスの設定を変えることで MC の色を変えていました。Flash 8 以降では、ColorTransform クラスを用います。ColorTransform では直接 MC を引数には取らず、flash.geom.Transform クラスを経由して MC の色を変更します。

下記のスクリプトが、この BabyBitmapCreator の赤ん坊のカラー設定に関連する部分です。BabyBitmapCreator._onOpening メソッドは、エフェクトがかけられた赤ん坊のムービーを包含している MC が（BabyBitmapCreator クラスのインスタンス）open ラベルにきたときに呼ばれる、プライベートなイベントハンドラです。

SOURCE 赤ん坊の色の変更

```
public function initialize(colorTrans:ColorTransform):Void {
     .
     .
     .

     this.colorTrans = colorTrans;


     // 再生
     this.gotoAndPlay('opening');
}

private function _onOpening():Void {
     // エフェクトをかけたムービーへのパスの設定
     this.effectedMovie = this.babyEffectWrapped_mc.effectedMovie_mc;
     .
     .
     .

     // 色の設定
     var trans:Transform = new Transform(this.effectedMovie);
     trans.colorTransform = this.colorTrans; // 色の設定

     .
     .
     .
}
```

ムービーをビットマップ化しメモリに格納する

次に、この解説のメインであるムービーをビットマップ化し、メモリに格納していくところを見てみます。

SOURCE BabyBitmapCreator._onOpening メソッド

```
private function _onOpening():Void {
     // エフェクトをかけたムービーへのパスの設定
     this.effectedMovie = this.babyEffectWrapped_mc.effectedMovie_mc;
```

次のページに続く

```
     .
     .
     .

     // キャプチャした BitmapData を全フレーム格納していく
     var target:BabyBitmapCreator = this;

     this.effectedMovie.onEnterFrame = function():Void {
          var babyBitmap:BitmapData = new BitmapData(this._width, this._height, true,
                                                                                0x00FFFFFF);

          babyBitmap.draw(target.babyEffectWrapped_mc);

          target.arrBitmap.push(babyBitmap);

          if (this._currentframe == this._totalframes) {
               delete this.onEnterFrame;

               target.gotoAndPlay('ending');
          }
     };
}
```

ここでは、エフェクトをかけた赤ん坊のムービーの onEnterFrame() イベントハンドラを利用して、フレームごとにビットマップを作成しています。ビットマップのキャプチャには、flash.display.BitmapData のインスタンスを生成し、キャプチャしたいMC を引数に取り BitmapData.draw() メソッドを行います。この draw() メソッドは、ステージからの見た目ではなく、引数に取られた MC から見たレンダリング結果を書き込みます。そして、その BitmapData を BabyBitmapCreator の保持する配列に格納していきます。

これで、赤ん坊の一連のアニメーションをビットマップとしてメモリに格納する作業が終わりました。配列への格納が終わると、onEnterFrame() イベントハンドラを停止し、［ending］ラベルへムービーを飛ばします。［ending］ラベルのアニメーションが終わると、BabyBitmapCreator.onInitEnd() イベントハンドラが呼ばれ、OrbControl によりメインとなるコンテンツが開始されます。

赤ん坊のアニメーション

先のスクリプトが終了すると、OrbControl.createDrop() メソッドから attachMovie() により、Baby クラスのインスタンスが生成されます。Baby のインスタンスが attachMovie() されると、Baby.fall() メソッドが即座に呼ばれます。

SOURCE Baby.fall

```
private function fall():Void {
     var arrBitmap:Array = Baby.babyBitmapCreator.getBitmaps();
     var index:Number = Math.floor(Math.random() * arrBitmap.length);
     var fallScale:Number = 10; // 落ちたとみなすサイズ
```

次のページに続く

```
        this.attachBitmap(arrBitmap[index], 1);

        this.onEnterFrame = function():Void {
             index += Baby.PLAY_SPEED;

             this.attachBitmap(arrBitmap[index % arrBitmap.length], 1); // 同じ深度で上書きする

             this._x += Baby.FALL_EASING * (this.targetPoint.x - this._x);
             this._y += Baby.FALL_EASING * (this.targetPoint.y - this._y);

             this._xscale += Baby.FALL_EASING * (0 - this._xscale);
             this._yscale += Baby.FALL_EASING * (0 - this._xscale);


             // 落下終了
             if (this._xscale < fallScale and this._yscale < fallScale) {
                  this.onFallEnd(this);
                  this.removeMovieClip();
             }
        };
   }
```

まずは、Baby クラスの static 変数として BabyBitmapCreator のインスタンスが設定してあるので、そこから配列としてビットマップを取り出します。そして、配列を順次取り出すための index 値を設定します。ムービー自体はループになるように作成していますので、ランダムな地点からスタートするように、index の初期値を配列の数からランダムに生成しています。

そして、深度が 1 のところへ attachBitmap() によりビットマップを貼り付けます。onEnterFrame() でも同じように index 値をインクリメントして、その配列のビットマップを attachBitmap() します。ここでは毎回深度 1 のところへ attachBitmap() しているため、ビットマップが重なることなく上書きされていきます。

特定のサイズになると赤ん坊が落下したとみなし、Baby.onFallEnd() イベントハンドラを呼び、自身を removeMovieClip() により消します。

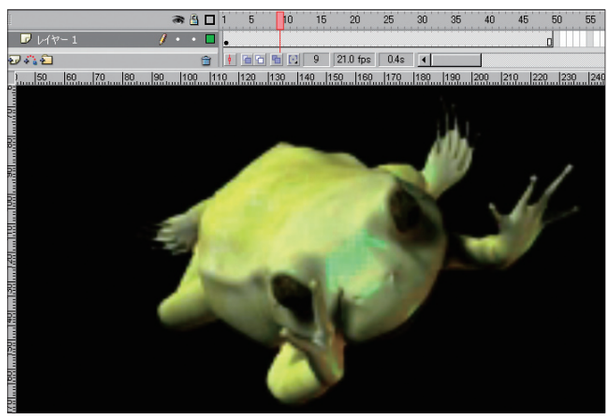
これが赤ん坊が生成されてから落下するまでの一連の流れとなっています。

カエルはαビデオで

カエルと赤ん坊は同じ役割ですが、大きく違うのは、赤ん坊はαビデオをビットマップ化して流しているのに対し、カエルはαビデオのまま使っていることです。

これは、見せ方的に何か意味があるわけではなく、メモリにビットマップ化して格納したものとそのままαビデオとして使う方法を比較しようと思い、あえてαビデオのまま使っています。

カエルはαビデオのまま



Frog クラスと Baby クラスは中身の実装の違いはあるものの、似たような役割を担うクラスとなります。そのような場合は同じインターフェースを実装するのもよいでしょう。

SOURCE Frog.fall メソッド

```
    /**
    * カエルを落下させる
    *
    * @return Void
    */
    public function fall():Void {
         var fallScale:Number = 3; // 落ちたとみなすサイズ

         this.effectedMovie_mc.gotoAndStop(Math.floor(Math.random() * this._totalframes) +
                                                                                          1);

         this.onEnterFrame = function():Void {
              this._x += Baby.FALL_EASING * (this.targetPoint.x - this._x);
              this._y += Baby.FALL_EASING * (this.targetPoint.y - this._y);

              this._xscale += Baby.FALL_EASING * (0 - this._xscale);
              this._yscale += Baby.FALL_EASING * (0 - this._xscale);

              this.effectedMovie_mc.gotoAndStop((this.effectedMovie_mc._currentframe +
                                   Baby.PLAY_SPEED) % this.effectedMovie_mc._totalframes + 1);

              // 落下終了
              if (this._xscale < fallScale and this._yscale < fallScale) {
                   this.onFallEnd(this);
                   this.removeMovieClip();
              }
         };
    }
```

前ページの Baby.fall メソッドと比較してみましょう。Baby クラスのほうは配列に入った BitmapData を描画していくのに対し、Frog はタイムラインの gotoAndStop() で描画しています。しかし、実装が異なっても使う側からは同じように使えます。