hand transfer

Creator
セトウナオ（STAC STAR）

αビデオによる画面遷移

αビデオを使った画面遷移の実験。
同時に、AS1.0とAS2.0でのオブジェクト指向スクリプティングの違いにもふれる。

Title	hand transfer
Sample URL	http://stacstar.jp/hand_transfer/
Archive	handtransfer.zip
File	16
スクリプト	ActionScript 1.0
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
αビデオを使った画面遷移

HTML と違い、Flash は画面の遷移がシームレスに、そして動的に展開できます。このコンテンツは、αビデオを使った画面遷移として実験的に制作したものです。画面上を手が覆いつくし写真が遷移していく展開、コーヒーがこぼれて画面がクローズする展開、2 カ所においてαビデオを使用しています。

画面上を手が覆いつくすことで画面が遷移する

コーヒーがこぼれて画面がクローズする

このコンテンツを最初に考えたときはまだ Flash Player 7 のときでした。Flash Player 7 では、αビデオの機能が搭載されておらず、Flashビデオをデザインに馴染ませるためには一工夫必要でした。クロマキーで撮影した素材を動画編集ソフトを用い動画の下にくるデザインと合成して馴染むように見せかけたり、いったんαチャンネルを持つ PNG の連番データとして書き出したり…。

しかし、動画をデザインに馴染ませる方法では、動画のメインオブジェクトの下にくるデザインが動的なものには対応できませんし、また、PNG の連番を読み込む手法では長時間のムービーの場合、ファイル容量が巨大となってしまいます。

このような不都合を、Flash Player 8 ではαビデオの機能として解決してくれます。

○ 動画の撮影と編集

今回のようなちょっとした素材であれば、僕のような素人でも撮影や編集ができますが、本来、動画の撮影には非常にたくさんの経験と知識が必要です。正直、今回撮影した映像は、環境や時間が思いどおりいかず、とてもよい素材とはいえません。その失敗からの経験も含め、流れを説明していきます。

用意したのは下記のものです。

- DV カメラ
- 三脚
- 発色のよい緑色の A3 の紙
- 人
- コーヒー

人とコーヒーについては撮影用の素材です。カメラに関しては、本格的なビデオカメラが用意できれば編集の際に一番加工しやすいのですが、なかなか用意するのはむずかしいでしょう。今回は、一般的な家庭にあるようなコンパクトな DV カメラを使用しています。三脚は、固定視点で撮る場合にはあったほうが非常に便利です。緑色の紙はキーイング（対象となるオブジェクト以外を抜き去る処理）のための背景として利用します。文具ショップで、反射が少なく、発色のよい A3 程度の大きさで緑色（#00FF00）に近いものを購入しました。

写真であればパスで抜くこともできますが、動画ではパスで抜くことはせず、色を利用して背景を抜くことが多いです（他にも明るさや別の動画、たとえば背景だけの動画との差分などで抜いたりします）。

圧縮のないフォーマットで撮影する分には対象物と被らないどんな色の背景でもよいのですが、DV カメラでは DV 圧縮という非可逆圧縮がされています。DV 圧縮は、人間の目の特性を考慮し、色情報よりも輝度情報にデータを使います。その輝度情報の中でも一番多く持っているのが緑色の情報です。そのため、ブルーバックではなくグリーンバックのほうがきれいにキーイングができるようです。ただし、対象となる素材が緑を含むようなものであれば、わざわざグリーンである必要はないでしょう。

背景に色をしけばきれいに抜けるかというとそうではありません。キーイングをうまく行えるかどうかは、ほとんど撮影にかかっています。全体を統一の明るさでライティングさせることができるとうまく切り抜けるのですが、背景にしわが出たり影が落ちると編集のときに非常に時間がかかってしまいます。特に、撮影は時間がかかるし、撮り直しがきかなかったりしますので、十分なテストと準備をしてから臨んだほうがよいでしょう。この他の撮影やキーイング、編集に関してはタナカミノルさんの記事（102 ページ）に詳しく書かれています。

撮影の様子

スクリプト
AS1.0 によるオブジェクト指向スクリプティング

今回のコンテンツはAS1.0で構成してある他は、特記する事項はありません。ここでは、AS2.0の視線からAS1.0 の書き方を見ることにより、AS1.0、AS2.0 両方の理解をより深めたいと思います。

このコンテンツでは init.as ファイルは除き、AS2.0 同様 1 つのクラスで 1 つの AS ファイルで構成してあります（AS2.0 のクラス定義は外部 AS ファイルに行います）。
まずは、クラスの全体的な役割を見てみます。

使用するクラス

クラス	概要
init.as	ステージの設定やパッケージ空間を生成する
CoffeeMovie	画面をコーヒーがこぼれるムービーで覆いつくし、JavaScript 経由でウィンドウを閉じるクラス
HandMovie	画面を手で覆い隠すムービーとなるクラス
HandTransferContents	このコンテンツのステージとなるクラス
HandTransferControl	各オブジェクトのイベントを受け取り、全体を制御するクラス
PaperController	紙くずの画像ビューアコントローラとなるクラス
Photo	写真の情報を持つエンティティとなるクラス
PhotoAlbum	写真情報を複数格納するクラス
PhotoAlbumFactory	XML から写真情報を格納するクラスを作成するクラス
PhotoIterator	写真情報を格納するオブジェクトから写真情報を数えるクラス
PhotoViewer	画像のある URL から画像を読み込み、表示させるクラス

AS1.0 でのクラスの作成

AS2.0では、classキーワードによりクラスを作成しました。しかし AS1.0には classキーワードはありません。では、どうすればクラスを作成できるのか？　AS1.0 では、コンストラクタを作成してあげることがクラスの作成といえます。コンストラクタとは、オブジェクトが生成されたときに最初に実行される関数です。つまり、関数を作成することで、その関数をクラスのコンストラクタとして機能させることが可能なわけです。
一般的にクラス名は英字の大文字から始まるので、コンストラクタの関数もそれに従います。

SOURCE コンストラクタのサンプル

```
var Sample = function() {
    trace('Sample called.');
};
```

ここでは、変数 Sample に無名関数のオブジェクトを代入しましたが、これは以下のような書き方でも問題ありません。

SOURCE 変数 Sample を呼び出す関数 Sample()

```
function Sample() {
    trace('Sample called.');
}
```

このコンストラクタを作成するだけでクラスを定義したのと同意義になります。

パッケージ空間の作成

クラスを定義するときに、特定のパッケージにそのクラスを定義することがあります。AS2.0 であれば、次のようにクラス定義の際にパッケージ空間も同時に定義できます。

SOURCE AS2.0 でのパッケージ空間の定義

```
class packageNameA.packageNameB.ClassName {
    public function ClassName() {
    }
}
```

では、AS1.0 ではどうするのでしょうか？　ActionScript では、_global オブジェクトを 1 番上のレベルとし、そこにオブジェクトのプロパティを連鎖的に格納していくことでパッケージ空間が生成されています。たとえば、上記の AS2.0 のスクリプトを AS1.0 で書いてみましょう。

SOURCE AS1.0 でのパッケージ空間の定義

```
_global.packageNameA = new Object();
_global.packageNameA.packageNameB = new Object();
_global.packageNameA.packageNameB.ClassName = function() {
};
```

このようになります。つまり、Object クラスがダイナミッククラスなのを利用して、動的にプロパティを加えていき、そこにクラス（コンストラクタ）を作成しているのです。

○ プロパティの宣言

プロパティの宣言について、AS2.0 との比較をしていきます。
AS2.0 では、次のようにクラス定義の中に宣言していました。

SOURCE AS2.0 でのプロパティの宣言

```
class packageNameA.packageNameB.ClassName {
    private var propA:Object;
}
```

それでは、AS1.0 のほうを見てみましょう。

SOURCE AS1.0 でのプロパティの宣言

```
_global.packageNameA.packageNameB.ClassName = function() {
    this._propA;
};
```

このようにコンストラクタ内でダイナミックに定義していきます。定義しなくとも使えるのですが、定義しておいたほうがわかりやすいでしょう。
また、AS1.0 には可視性（public / private）がないため、識別するために private のものには「_」（アンダーバー）を付けたりなど工夫をすることがあります。

○ メソッドの宣言

メソッドの宣言について AS2.0 との比較をしていきます。
AS2.0 では、次のようにクラス定義の中に宣言していました。

SOURCE AS2.0 でのメソッドの宣言

```
class packageNameA.packageNameB.ClassName {
    public function method():Void {
        trace('method called.');
    }
}
```

では AS1.0 のほうを見てみましょう。

SOURCE AS1.0 でのメソッドの宣言

```
_global.packageNameA.packageNameB.ClassName = function() {
    this.method = function() {
        trace('method called.');
    };
};
```

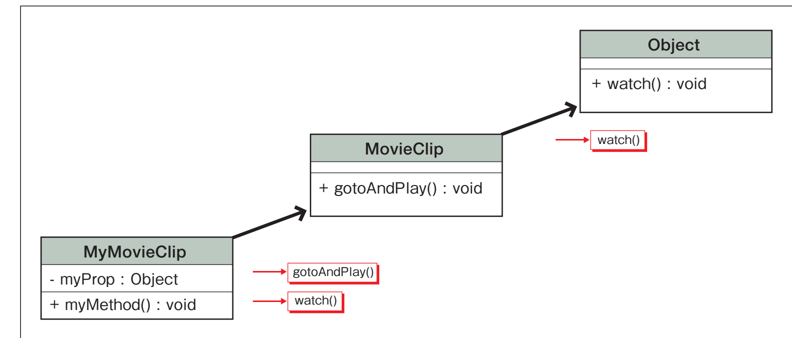

このように、プロパティと同様に定義することが可能です。しかし、プロパティと違って、メソッドというのはあまり書き換えることがありません。1 つ 1 つのオブジェクトがそれぞれ同じ機能のメソッドを持つのではなく、1 カ所に持つほうがメモリを節約できます。その 1 カ所にまとめたやり方を見てみましょう。

SOURCE メソッドを 1 つにまとめる

```
// クラスを宣言
_global.packageNameA.packageNameB.ClassName = function() {
};
// メソッドを宣言
_global.packageNameA.packageNameB.ClassName.prototype.method = function() {
    trace('method called.');
};
```

ここで、prototype というキーワードが出てきました。prototype とは、ActionScript の肝となるキーワードです。指定のオブジェクト自身が持っていないメンバーを呼び出した場合に、オブジェクトは prototype に格納されたオブジェクトから指定のメンバーを探し出します。そこにもない場合には、さらにそのオブジェクトの中にある prototype を探しに行きます。これは、prototype チェーンと呼ばれています。この特性を利用してメソッドを 1 つの場所だけにおいておき、参照させるのです。

prototype チェーン



○ 継承

AS2.0 では extends キーワードを使ってクラスを拡張することができます。

SOURCE AS2.0 でのクラスの拡張

```
class packageNameA.packageNameB.ClassName extends MovieClip {
    public function ClassName() {
    }
}
```

AS1.0 では、先ほど出てきた prototype キーワードを使います。prototype はオブジェクトに指定したメンバ

ーが存在しない場合にさかのぼって探すオブジェクトを指定するものでした。つまり、prototype に継承したいクラスのオブジェクトを設定すると、そのクラスをベースに自分のクラスを作成することができます。これが AS1.0 における継承の仕組みです。何も指定しない場合は、Object クラスのインスタンスが格納されます。

SOURCE AS1.0 での継承

```
// クラスを宣言
_global.packageNameA.packageNameB.ClassName = function() {
};
//MC を継承
_global.packageNameA.packageNameB.ClassName.prototype = new MovieClip();
```

この prototype というキーワードが ActionScript を支える大きな柱になっています。しかし、AS2.0 では prototype という概念を隠蔽することにより、クリエーターが仕組みの裏側を意識することなく、プログラミングできるように設計されています。

シンボルへの割り当て

AS2.0 では、ライブラリの MC シンボルに対して AS2.0 のクラスを指定することにより、MC の機能から独自のクラスの機能のシンボルとして置き換えることができました。AS1.0 では下記のように行います。

SOURCE シンボルにクラスを適用する

```
Object.registerClass(' 識別子 ', クラス );
```

これにより、特定の識別子を持つシンボルに自作のクラスを適応することができます。

手のムービーの実装

最後に、本コンテンツで使用したスクリプトを紹介します。写真の切り替えに使っている手のムービーの実装を見てみましょう。

SOURCE HandMovie クラスの定義

```
/**
* コンストラクタ。
*
* @return        Void
*/
org.graffiti_web.handTransfer.HandMovie = function() {
    /**
    * 手で覆われたときにそのまま再生していいか停止するかのフラッグ。
    *
    * @param  Boolean
    */
```

次のページに続く

```
    this._playFlag = false;

    this.stop();
};
```

このコンストラクタの宣言でクラスを定義しています。
パッケージ空間に関しては init.as にて宣言してあります。

SOURCE パッケージ空間の宣言

```
/**
* MovieClip を継承。
*/
org.graffiti_web.handTransfer.HandMovie.prototype = new MovieClip();
```

ここで prototype に対して MC のインスタンスを格納することで、MC を継承しています。このクラスは、HandMovie.start() メソッドにて画面を覆い隠し、HandMovie.open() メソッドにて覆い隠していた画面を開きます。そして画面を隠したときと、画面が開かれたときにカスタムのイベントハンドラ（onHide() イベントハンドラと onEnd() イベントハンドラ）を呼び出します。
この画面を覆い隠すメソッドと開くメソッドを見てみます。

SOURCE 画面を覆い隠すメソッドと開くメソッド

```
/**
* ムービーを開始する。
*
* @return     Void
*/
org.graffiti_web.handTransfer.HandMovie.prototype.start = function() {
    this.play();
};

/**
* 終了アニメーションに移るようにする。
*
* @return     Void
*/
org.graffiti_web.handTransfer.HandMovie.prototype.open = function() {
    this._playFlag = true;
    this.play();
};
```

このように、非常にシンプルな実装となっています。
また、最後に制作したクラスをライブラリのシンボルへ割り当てています。

SOURCE クラスをライブラリのシンボルへ割り当てる

```
// シンボルに割り当て
Object.registerClass('handMovie', org.graffiti_web.handTransfer.HandMovie);
```