Manejo de bucles de tiempo

Web

Creando nuestro propio interpolador

Lo realizaremos de la siguiente manera: tendremos una clase `Interpolator`, que calculará la interpolación, en cada *tick* disparará un evento, y se podrá acceder a sus datos actuales. De esta manera, un cliente podrá instanciarlo y se podrán suscribir otros objetos a él. El cliente podrá ejecutarlo, obtener los datos de manera sincrónica y hacer que los escuchas se enteren de los cambios, ya que el interpolador propagará el evento.

Disparará los eventos:

- **`TICK`** – al ejecutarse un *tick*.
- **`START`** – al comenzar.
- **`COMPLETE`** – al terminar.

En el evento, estará entre otros, el dato de la interpolación, que será un valor entre 0 y 1. Será 0 cuando empiece y 1 cuando termine.

Utilizando 0 como valor inicial y 1 como valor final

¿Cuál es la ventaja de utilizar valores de interpolación entre 0 y 1? Podremos aplicarlo para el caso en que un objeto quiera calcular un valor a partir de la interpolación. El valor interpolado se obtiene con la fórmula que se elija para hacer la interpolación. El valor obtenido en cada *tick* se multiplica por la diferencia entre el (valor final – valor inicial), y se le agrega el valor inicial.

Si por ejemplo, la interpolación resultante es 0.5, y quisiéramos aplicar esta interpolación a la posición X, donde el valor inicial es 100 y el final es 500. El cálculo a realizar es:

- Obtener la diferencia entre el valor final y el inicial: 500 – 100 = 400.
- Obtener la interpolación de una fórmula determinada. Le debemos pasar por ejemplo: `formulaLineal` (`milisegundoActual`, 0, 1, `milisegundosTotales`)
- Multiplicar la interpolación por la diferencia obtenida: 400 * 0.5 = 200.
- Agregar al valor inicial, el valor obtenido en el paso anterior: 100 + 200 = 300.
- Y así tendríamos el valor a aplicar a la posición.

Podemos hacer que el interpolador haga este cálculo en lugar de un objeto, si a éste le pasa el valor inicial y final o hacer que solamente devuelva un valor entre 0 y 1, para que el objeto utilice el valor interpolado de la manera que quiera.

Veamos que cálculos haría el interpolador, a grandes rasgos:

```
//Al comenzar, obtenemos el valor total restando al valor final el
valor inicial.

var totalValue:Number = finalValue – initialValue;

//En cada Tick:
//Obtenemos el valor interpolado multiplicando
//el valor total por la interpolación actual.
var interpolation:Number = formula(currentTick, 0, 1, duration);

var interpolatedValue:Number = totalValue * interpolation;
```

```
//Agregamos al valor inicial el valor interpolado.
var finalValue:Number = initialValue + interpolatedValue;
```

La forma en la que el interpolador comunicaría el valor interpolado sería devolviéndolo a través de un método, siempre y cuando sea llamado y comunicado además con un evento `InterpolatorEvent`.

Veamos algunas particularidades de nuestra clase:

A diferencia de la implementación de otras librerías para la interpolación, no le pasaremos un objeto como parámetro para que el `Interpolator` le aplique el cambio. Podría llamarse desde un *game loop* (bucle de juego), un `LoopManager` (manejador de bucles), o un manejador de evento `ENTER_FRAME` o `TIMER`, por mencionar un par de ejemplos.

El `Interpolator` podrá calcular la diferencia entre intervalos en los que ha sido llamado para hacer el cálculo de las interpolaciones. Los objetos que lo utilicen podrán indicarle qué fotograma o qué milisegundo de la animación deberá ser calculado. O simplemente se le podrá pedir que ejecute el próximo *tick*. Esto permitirá ejecutar una animación de la posición inicial a la final, o realizar la animación desde un momento determinado.

Así, si por ejemplo, tenemos un objeto en la posición 20, y queremos llevarlo a la posición 30, se deberían seguir los 5 pasos con una fórmula lineal detallados a continuación:

```
//Al comenzar, obtenemos el valor total restando al valor final el
valor inicial.
var totalValue:Number = 30 – 20; // El resultado será: 10
//Ya tomamos la referencia de la fórmula a utilizar.
var formula:Function = Linear;
```

Ya podemos sacar una conclusión: que el paso 0 y el último paso no harían falta calcularlos. El primero nos daría 0, es decir, ningún cambio; y el último, al multiplicarlo por 1 nos daría la transformación final. Aunque en teoría esto es así con algunas fórmulas como elástico, y dependiendo de los valores adicionales que le estemos pasando, el último paso puede dar un valor muy cercano a 1, pero no exactamente 1. Aplicando directamente el último paso sin cálculo, nos aseguramos que los valores sean alcanzados con precisión. Y luego, tendríamos los siguientes:

```
//TICK 0
//Obtenemos el valor interpolado multiplicando
//el valor total por el valor de la interpolación actual entre 0 y
1.
var currentInterpolation:Number = formula(5, 0, 0, 1);
var interpolatedValue:Number = 10 * 0;

//Agregamos al valor inicial el valor interpolado.
var finalValue:Number = 20 + 0; //Posición: 20.

//TICK 1
//Obtenemos el valor interpolado multiplicando
//el valor total por el valor de la interpolación actual entre 0 y
1.
var currentInterpolation:Number = formula(5, 1, 0, 1);
var interpolatedValue:Number = 10 * 0.2;

//Agregamos al valor inicial el valor interpolado.
var finalValue:Number = 20 + 0.2; //Posición: 22.

//TICK 2
//Obtenemos el valor interpolado multiplicando
//el valor total por el valor de la interpolación actual entre 0 y
1.
var currentInterpolation:Number = formula(5, 2, 0, 1);
var interpolatedValue:Number = 10 * 0.4;

//Agregamos al valor inicial el valor interpolado.
var finalValue:Number = 20 + 4; //Posición: 24.

//TICK 3
//Obtenemos el valor interpolado multiplicando
//el valor total por el valor de la interpolación actual entre 0 y
1.
var currentInterpolation:Number = formula(5, 3, 0, 1);
var interpolatedValue:Number = 10 * 0.6;

//Agregamos al valor inicial el valor interpolado.
```

```
var finalValue:Number = 20 + 6; //Posición: 26.


//TICK 4
//Obtenemos el valor interpolado multiplicando
//el valor total por el valor de la interpolación actual entre 0 y
1.
var currentInterpolation:Number = formula(5, 4, 0, 1);
var interpolatedValue:Number = 10 * 0.8;


//Agregamos al valor inicial el valor interpolado.
var finalValue:Number = 20 + 8; //Posición: 28.


//TICK 5
//Obtenemos el valor interpolado multiplicando
//el valor total por el valor de la interpolación actual entre 0 y
1.
var currentInterpolation:Number = formula(5, 5, 0, 1);
var interpolatedValue:Number = 10 * 1;


//Agregamos al valor inicial el valor interpolado.
var finalValue:Number = 20 + 10; //Posición: 30.
```

IMPLEMENTADO LAS FÓRMULAS DE NUESTRO INTERPOLADOR

Implementaremos algunas de las fórmulas que utilizará nuestro interpolador. Muchas de las ellas ya existen en el sitio de Robert Penner, implementadas en métodos y en clases de AS 2.0 en:

http://www.robertpenner.com/easing/penner_easing_as2.zip

Obtendremos las clases en AS 2.0 y tendremos que pasarlas a AS 3.0.

Las clases que utilizaremos son:

- `Back`
- `Bounce`
- `Elastic`
- `Linear`
- `Quart`

Notemos que tenemos tres métodos por cada clase:

- **`easeIn`**: aplica el efecto al comienzo de la interpolación.
- **`easeOut`**: aplica el efecto al final de la interpolación.
- **`easeInOut`**: aplica el efecto al comienzo y al final.

Crearemos las clases antes listadas en el paquete:

`com.alfaomegaeditor.utils.interpolation.easing`

Si bien no es para nada aconsejable utilizar variables cuyo nombre sea una letra, para este único caso, mantendremos las variables de este modo, tal cual las implementó Robert Penner, pero convertiremos las funciones estáticas en métodos que pertenecerán a una clase. Y para el agregado futuro de otras clases con otras fórmulas, ya sean las que no hemos implementado de Robert Penner o cualquiera que se pudiera implementar, crearemos una interfaz. De esta manera, el uso de clases con fórmulas se hará de forma segura, y aplicaremos la buena práctica de programar a interfaz.

Cada uno de los métodos recibirá estos parámetros:

`t – tick`: es el fotograma o momento de la interpolación a calcular.

`b – begin`: es el valor inicial.

`c – change`: el valor final.

`d – duration`: es la duración ya sea en fotogramas o en tiempo.

Están los valores opcionales:

a – amplitude: valor opcional para efectos la clase `Elastic` o `Back` que determina la amplitud del efecto.

p – period: se utiliza para la clase `Elastic` indica la frecuencia del efecto de elástico.

Para no hacer chequeos mientras se está efectuando la interpolación de si existe o no el valor correspondiente a amplitud o a período, le agregaremos estos dos parámetros a todas las fórmulas. Nuestra interfaz `Iformula` estará en el paquete:
`com.alfaomegaeditor.utils.interpolation.easing`

Y quedaría así:

```
package com.alfaomegaeditor.utils.interpolation.easing
{
  public interface IFormula
  {
    function easeIn (t:Number, b:Number,
                     c:Number, d:Number,
                     a:Number=NaN, p:Number=NaN):Number

    function easeOut(t:Number, b:Number,
                     c:Number, d:Number,
                     a:Number=NaN, p:Number=NaN):Number

    function easeInOut(t:Number, b:Number,
                       c:Number, d:Number,
                       a:Number=NaN, p:Number=NaN):Number
  }
}
```

Y a continuación las clases pasadas a AS 3.0:

La clase `Back`, que provoca una animación que se pasa del valor final y luego vuelve hasta alcanzarlo definitivamente. Lo que determina cuánto se pasará del valor final es el parámetro `a` (amplitud), y es un parámetro opcional.

```
package com.alfaomegaeditor.utils.interpolation.easing
{
  public class Back implements IFormula
  {
    public function easeIn(t:Number, b:Number,
                           c:Number, d:Number,
                           a:Number=NaN, p:Number=NaN):Number
    {
      if (isNaN(a))
      {
        a = 1.70158
      }

      return c * (t /= d) * t * ((a + 1) * t - a) + b;
    }

    public function easeOut(t:Number, b:Number,
                            c:Number, d:Number,
                            a:Number=NaN, p:Number=NaN):Number
    {
      if (isNaN(a))
      {
        a = 1.70158
      }

      return c * ((t = t / d - 1) * t * ((a + 1) * t + a) + 1) + b;
    }

    public function easeInOut(t:Number, b:Number,
                              c:Number, d:Number,
                              a:Number=NaN, p:Number=NaN):Number
```

```
    {
      if (isNaN(a))
      {
        a = 1.70158
      }

      if ((t /= d / 2) < 1)
      {
        return c / 2 * (t * t * (((a *= (1.525)) + 1) * t - a)) +
b;
      }
      return c / 2 * ((t -= 2) * t * (((a *= (1.525)) + 1) * t + a)
+ 2) + b;
    }
  }
}
```

La clase `Bounce` (efecto de rebote):

```
package com.alfaomegaeditor.utils.interpolation.easing
{
  public class Bounce implements IFormula
  {
    public function easeOut(t:Number, b:Number,
                            c:Number, d:Number,
                            a:Number=NaN, p:Number=NaN):Number
    {
      if ((t /= d) < (1 / 2.75))
      {
        return c * (7.5625 * t * t) + b;
      }
      else if (t < (2 / 2.75))
      {
        return c*(7.5625 * (t -= (1.5 / 2.75)) * t + .75) + b;
      }
```

```
      else if (t < (2.5 / 2.75))
      {
        return c * (7.5625 * (t -= (2.25/2.75)) * t + .9375) + b;
      }
      else
      {
        return c * (7.5625 * (t -= (2.625 / 2.75)) * t + .984375) +
b;
      }
    }

    public function easeIn(t:Number, b:Number,
                           c:Number, d:Number,
                           a:Number=NaN, p:Number=NaN):Number
    {
      return c - easeOut (d - t, 0, c, d) + b;
    }

    public function easeInOut(t:Number, b:Number,
                              c:Number, d:Number,
                              a:Number=NaN, p:Number=NaN):Number
    {
      if (t < d / 2)
      {
        return easeIn (t * 2, 0, c, d) * .5 + b;
      }
      else
      {
        return easeOut (t * 2 - d, 0, c, d) * .5 + c * .5 + b;
      }
    }
  }
}
```

La clase `Elastic` (efecto de elástico) tiene dos valores opcionales: `a` (amplitud), que indica cuánto se excederá para hacer el efecto elástico; y `p` (período), que determina la frecuencia en la que hará el efecto de elástico. Estos dos valores son también opcionales.

```
package com.alfaomegaeditor.utils.interpolation.easing
{
  public class Elastic implements IFormula
  {
    public function easeIn(t:Number, b:Number,
                           c:Number, d:Number,
                           a:Number=NaN, p:Number=NaN):Number
    {
      if (t == 0)
      {
        return b;
      }

      if ((t /= d) == 1)
      {
        return b + c;
      }

      if (isNaN(p))
      {
        p = d * .3;
      }

      var s:Number;
      if (isNaN(a) || a < Math.abs(c))
      {
        a = c;
        s = p / 4;
      }
      else
      {
        s = p / (2 * Math.PI) * Math.asin (c / a);
      }
      return -(a * Math.pow(2, 10 * ( t -= 1)) * Math.sin( (t * d -
s) *
          ( 2 * Math.PI ) / p )) + b;
```

```
    }

    public function easeOut(t:Number, b:Number,
                            c:Number, d:Number,
                            a:Number=NaN, p:Number=NaN):Number
    {
      if (t == 0)
      {
        return b;
      }

      if ((t /= d) == 1)
      {
        return b + c;
      }

      if (isNaN(p))
      {
        p = d * .3;
      }

      var s:Number;
      if (isNaN(a) || a < Math.abs(c))
      {
        a = c; s = p / 4;
      }
      else
      {
        s = p/(2*Math.PI) * Math.asin (c/a);
      }

      return (a * Math.pow(2, -10 * t) * Math.sin( (t * d - s) *
          (2*Math.PI)/p ) + c + b);
    }

    public function easeInOut(t:Number, b:Number,
                              c:Number, d:Number,
```

```
                                a:Number=NaN, p:Number=NaN):Number
    {
      if (t == 0)
      {
        return b;
      }

      if ((t /= d / 2) == 2)
      {
        return b + c;
      }

      if (isNaN(p))
      {
        p = d*(.3 * 1.5);
      }

      var s:Number;
      if (isNaN(a) || a < Math.abs(c))
      {
        a = c; s = p / 4;
      }
      else
      {
        s = p/(2 * Math.PI) * Math.asin (c / a);
      }

      if (t < 1)
      {
        return -.5 * (a * Math.pow(2, 10 * (t -= 1)) *
          Math.sin( (t * d - s)*( 2 * Math.PI) / p )) + b;
      }

      return a * Math.pow(2, -10 * (t -= 1)) * Math.sin( (t * d -
s) *
          (2 * Math.PI) / p ) * .5 + c + b;
    }
```

```
  }
}
```

La clase `Linear`:

```
package com.alfaomegaeditor.utils.interpolation.easing
{
  public class Linear implements IFormula
  {
    public function easeIn(t:Number, b:Number,
                           c:Number, d:Number,
                           a:Number=NaN, p:Number=NaN):Number
    {
      return c*t/d + b;
    }

    public function easeOut(t:Number, b:Number,
                            c:Number, d:Number,
                            a:Number=NaN, p:Number=NaN):Number
    {
      return c*t/d + b;
    }

    public function easeInOut(t:Number, b:Number,
                              c:Number, d:Number,
                              a:Number=NaN, p:Number=NaN):Number
    {
      return c*t/d + b;
    }
  }
}
```

La clase `Quart`:

```
package com.alfaomegaeditor.utils.interpolation.easing
{
  public class Quart implements IFormula
  {
    public function easeIn(t:Number, b:Number,
                           c:Number, d:Number,
                           a:Number=NaN, p:Number=NaN):Number
    {
      return c * ( t /= d) * t * t * t + b;
    }

    public function easeOut(t:Number, b:Number,
                            c:Number, d:Number,
                            a:Number=NaN, p:Number=NaN):Number
    {
      return -c * ((t = t / d - 1) * t * t * t - 1) + b;
    }

    public function easeInOut(t:Number, b:Number,
                              c:Number, d:Number,
                              a:Number=NaN, p:Number=NaN):Number
    {
      if ((t /= d / 2) < 1)
      {
        return c / 2 * t * t * t * t + b;
      }

      return -c / 2 * ((t -= 2) * t * t * t - 2) + b;
    }
  }
}
```

Implementado la clase InterpolatorEvent

Ya tenemos las fórmulas, que son la materia prima de nuestro interpolador, la clase que lo representa se llamará `Interpolator`. Su responsabilidad es hacer todos los cálculos relacionados con una interpolación, y comunicar los datos a través de eventos y desde sus propiedades. Y lo que haremos ahora es crear la clase de eventos del `Interpolator`, a la que llamaremos `InterpolatorEvent`. Esta clase vivirá en el paquete: `com.alfaomegaeditor.events`.

Tendremos tres tipos de evento:

- **`public static const TICK:String`**: cada vez que se ejecuta el interpolador.
- **`public static const START:String`**: cuando se ejecute el primer *tick*.
- **`public static const COMPLETE:String`**: cuando se ejecute el último *tick* de la animación.

Y que tendrá las siguientes propiedades:

- **`public var type:String`**: nombre del evento.
- **`public var currentMillisecond:Number`**: tiempo transcurrido desde que comenzó la animación en milisegundos. Sería el momento actual de la animación.
- **`public var totalMilliseconds:Number`**: la duración total de la interpolación en milisegundos.
- **`public var milisecondRatio:Number`**: valor entre 1 y 0 que hace referencia al milisegundo actual en forma relativa.
- **`public var interpolation:Number`**: un valor entre 0 y 1 de la interpolación.
- **`public var tweenValue:Number`**: valor resultante de la interpolación de dos valores opcionales. Por defecto estos dos valores son 0 y 1. En caso de no ser asignados, `tweenValue` será igual a `interpolation`.

Y el siguiente método facilitará en caso que algún objeto lo requiera, el cálculo de la interpolación entre dos valores:

- **`public function calculateTween(startValue:Number, endValue:Number):Number`**: dados dos valores, calcula el valor intermedio con la interpolación actual.

Nuestra clase `InterpolatorEvent`:

```
package com.alfaomegaeditor.events
{
  import flash.events.Event;
  import flash.utils.ByteArray;
  public class InterpolatorEvent extends Event
  {
    /*
    * Esta clase tiene la responsabilidad de contener los datos de
una
    * interpolación para ser utilizados por un despachador de even-
tos.
    */

    //Evento que se dispara a cada tick.
    public static const TICK:String = "interpolationTick";

    //Evento que se dispara al ejecutarse la primera interpolación.
    public static const START:String = "interpolationStart";

    //Evento que se dispara al ejecutarse la última interpolación.
    public static const COMPLETE:String = "interpolationComplete";

    //Milisegundo actual de la interpolación.
    private var _currentMillisecond:Number;

    //Milisegundos totales.
    private var _totalMilliseconds:Number;

    //Valor entre 1 y 0 que hace referencia al milisegundo actual
en forma
    //relativa.
    private var _milisecondRatio:Number;
```

```
    //Valor de la interpolación.
    private var _interpolation:Number;

    //Valor resultante de la interpolación de dos valores opcion-
ales, por
    //default estos dos valores son 0 y 1. En caso de no ser
asignados,
    //tweenValue será igual a interpolation.
    private var _tweenValue:Number;

    /**
     * Constructor.
     * @param type: el nombre del evento al cual suscribirse.
     */
    public function InterpolatorEvent(type:String,
                                      currentMillisecond_:Number,
                                      totalMillieconds_:Number,
                                      milisecondRatio_:Number,
                                      interpolation_:Number,
                                      tweenValue_:Number)
    {
      super(type);
      _currentMillisecond = currentMillisecond_;
      _totalMilliseconds = totalMillieconds_;
      _milisecondRatio = milisecondRatio_;
      _interpolation = interpolation_;
      _tweenValue = tweenValue_;
    }

    /**
     * Milisegundo actual, propiedad de solo lectura.
     */
    public function get currentMillisecond():Number
    {
      return _currentMillisecond;
    }
```

```
    /**
     * Milisegundos totales, propiedad de solo lectura.
     */
    public function get totalMilliseconds():Number
    {
      return _totalMilliseconds;
    }

    /**
     * Milisegundo actuales en forma relativa, representado por un
valor entre
     * 0 y 1. Propiedad de solo lectura.
     */
    public function get milisecondRatio():Number
    {
      return _milisecondRatio;
    }

    /**
     * El resultado de la interpolación, propiedad de solo lectura.
     */
    public function get interpolation():Number
    {
      return _interpolation;
    }

    /**
     * El resultado de la interpolación de dos valores opcionales.
     * Por defecto estos dos valores tienen los valores 0 y 1,
     * respectivamente. En caso de no ser asignados, tweenValue
será
     * igual a interpolation.
     */
    public function get tweenValue():Number
    {
      return _tweenValue;
```

```
    }

    /**
     * Dados dos valores, calcula el valor intermedio multiplicado
por la
     * interpolación.
     * @param startValue: valor inicial.
     * @param endValue: valor final.
     * @return devuelve el valor interpolado con la interpolación
actual.
     */
    public function calculateInterpolation(startValue:Number,
                                           endValue:Number):Number
    {
      return (endValue - startValue) * interpolation + startValue;
    }

    /**
     * Clona al evento, haciendo un clon también de los objetos
complejos
     * que pudiera contener.
     * @return
     */
    override public function clone():Event
    {
      var byteArray:ByteArray = new ByteArray();
      byteArray.writeObject(this);
      byteArray.position = 0;
      return Event(byteArray.readObject());
    }
  }
}
```

Implementando la clase Interpolator

Nuestra clase se encargará de devolver y disparar eventos con los datos de una interpolación. Comenzaremos por las propiedades. Tendremos constantes estáticas que harán referencia a las clases que pueden ser utilizadas para las fórmulas, y otras con los nombres de sus métodos.

Además, utilizaremos otras constantes privadas y públicas que se describen por sí mismas en la misma clase. Entonces, aprovecharemos para ejercitar la comprensión de un código ya terminado.

Las propiedades a utilizar serán las siguientes:

- **`public var currentMillisecond:Number`**: el momento actual de la animación.
- **`public var totalMilliseconds:Number`**: la duración total de la interpolación en milisegundos.
- **`public var milisecondRatio:Number`**: el valor entre 1 y 0 que hace referencia al milisegundo actual en forma relativa.
- **`public var amplitude:Number`**: el valor opcional para ciertas fórmulas como `Back` o `Elastic`. Indica la amplitud del efecto.
- **`public var period:Number`**: el valor opcional para las fórmulas. En la clase `Elastic`, determina las repeticiones del efecto elástico.
- **`public var interpolation:Number`**: es una variable de solo lectura de la propiedad `_interpolation`. Es un valor entre 0 y 1 de la interpolación.
- **`public var startValue:Number`**: asignando los valores `startValue` y `endValue`, obtendremos además el valor de la interpolación entre estos dos, listo para ser aplicado directamente.
- **`public var tweenValue:Number`**: propiedad de solo lectura, devuelve el valor interpolado entre `startValue` y `endValue`.
- **`public var endValue:Number`**: ver `startValue`.
- **`public var typeLoop:Number`**: podemos configurar 2 tipos de *loop* utilizando las constantes: `LOOP_HOLD` para que se detenga al llegar al último fotograma; y `LOOP_REPEAT` para que vuelva a comenzar. Estos *loop*s tendrán efecto cuando estemos utilizando los métodos: `nextTick` y `prevTick`.

Las propiedades privadas:

- **`private var _formula:Number`**: es una variable de solo lectura de la propiedad `_interpolation`. Es un valor entre 0 y 1 de la interpolación.
- **`private var _lastTimer:Number`**: tiempo transcurrido desde la última interpolación.
- **`private var _eventDispatcher`**: es el objeto que manejará los eventos.

Los métodos serán divididos por regiones de código:

Constructor

```
public function Interpolator(totalMilliseconds_:Number,
                                    formulaClass:Class=null,
                                    methodName:String =
EASE_IN,
                                    loopType_:String=null,
                                    startValue_:Number=NaN,
                                    endValue_:Number=NaN)-
```

El único valor obligatorio es: `totalMillisecond`. El resto de los valores los toma por defecto, si no les han sido asignados.

Configurador de fórmula

- **`public function setFormula(formulaClass:Class, methodName:String)void`**: configura la fórmula con que se harán las interpolaciones.

Configuradores de milisegundo

- Los *getters* y *setters* de `currentMillisecond` y `totalMilliseconds`.

Propiedades opcionales de las fórmulas

- Los *getters* y *setters* de `amplitude` y `period`.

Manejadores de ticks

Estos métodos juegan el papel de lo que sería un cabezal reproductor:

- **`public function nextTick():Number`**: calcula la siguiente interpolación al momento correspondiente de ser llamado y devuelve

`interpolation`, también, y despacha el evento `InterpolationEvent.TICK`.

- **`public function prevTick():Number`**: calcula la interpolación anterior al momento correspondiente de ser llamado, devuelve `interpolation` y despacha el evento `InterpolationEvent.TICK`.
- **`public function tickAtMillisecond (millisecond:Number):Number`**: calcula la interpolación en un milisegundo determinado. Este es el método central del manejo de *ticks*. El resto de los métodos terminan por converger en éste.
- **`public function tickAtMillisecondRatio (value:Number):Number`**: toma un valor entre 0 y 1. El 1 representa la duración total de la animación. Internamente, multiplica el valor recibido por la cantidad total de milisegundos y luego llama a `tickAtMillisecond`.

Calculadores de la interpolación

- **`public function calculateTween(startValue:Number, endValue:Number):Number`**: dados dos valores, calcula el valor intermedio con la interpolación actual.

Manejadores de escuchas y despacho de eventos

- **`public addEventListener(type:String, listener:Function):void`**: la clase `Interpolator`, utilizará la clase `EventDispatcher` por composición, y delegará en el agregar, remover y despachar eventos. Para agregar un escucha se le debe pasar el nombre del evento y la referencia del método escucha.
- **`public removeEventListener(type:String, listener:Function):void`**: para remover un escucha se le debe pasar: el nombre del evento y la referencia del método escucha.

Métodos auxiliares

- **`private function buildAndDispatchEvent():void`**: construye y despacha el evento.
- **`private function getLastInterval():void`**: devuelve el último intervalo de tiempo desde que fue llamado este método.

Para entender bien la idea de esta clase y no perdernos entre sus líneas de código, repasamos lo siguiente. El objetivo de esta clase es manejar estas propiedades para obtener datos de interpolación con algunas de las fórmulas que se configuren.

Y veamos en detalle cómo se relacionan con nuestras propiedades:

t - tick: es el milisegundo de la interpolación a calcular `currentMillisecond`.

b - begin: es el valor inicial. Este valor siempre será: 0 (cero).

C - change: el valor final. Este valor siempre será: 1 (uno).

d - duration: es la duración ya sea en fotogramas o en tiempo: `totalMillisecond`.

En nuestra clase el método que hará el cálculo de la interpolación es `tickAtMillisecond`.

Lo que determina el avance de nuestra animación o su retroceso son los métodos correspondientes a Manejadores de *tick*s.

La manera en que hemos planteado la clase nos permitirá hacer efectos interesantes como *time bullet* (tiempo de bala), que es el efecto de cámara lenta en medio de una animación, variando en tiempo de ejecución distintas propiedades.

Y la clase se vería así:

Clase `Interpolator`:

```
package com.alfaomegaeditor.utils.interpolation
{
  /**
   * Clase encargada de devolver y disparar eventos con los datos
   * de una interpolación.
   */

  import flash.events.EventDispatcher;
  import flash.utils.getTimer;

  import com.alfaomegaeditor.events.InterpolatorEvent;
  import com.alfaomegaeditor.utils.interpolation.easing.Back;
  import com.alfaomegaeditor.utils.interpolation.easing.Bounce;
  import com.alfaomegaeditor.utils.interpolation.easing.Elastic;
  import com.alfaomegaeditor.utils.interpolation.easing.Linear;
  import com.alfaomegaeditor.utils.interpolation.easing.Quart;
  import com.alfaomegaeditor.utils.interpolation.easing.IFormula;
```

```
  public class Interpolator
  {
    //Referencia a las clases para calcular las fórmulas.
    public static const BACK_FORMULA:Class = Back;
    public static const BOUNCE_FORMULA:Class = Bounce;
    public static const ELASTIC_FORMULA:Class = Elastic;
    public static const LINEAR_FORMULA:Class = Linear;
    public static const QUART_FORMULA:Class = Quart;

    //Nombres de los métodos a llamar de las clases del tipo IFor-
mula.
    public static const EASE_IN:String = "easeIn";
    public static const EASE_OUT:String = "easeOut";
    public static const EASE_IN_OUT:String = "easeInOut";

    public static const LOOP_HOLD:String = "loopHold";
    public static const LOOP_REPEAT:String = "loopRepeat";

    //Constantes utilizadas para agregar y remover escuchas.
    private static const USE_CAPTURE:Boolean = false;
    private static const PRIORITY:uint = 0;
    private static const USE_WEAK_REFERENCE:Boolean = true;

    //Valor inicial y final por defecto son 0 y 1.
    private static const START_VALUE:Number = 0;
    private static const END_VALUE:Number = 1;

    //Los milisegundos actuales y totales.
    private var _currentMillisecond:Number=0;
    private var _totalMilliseconds:Number;

    //Se utiliza mientras se está realizando la animación, en el se
guarda cual es
    //los últimos milisegundos transcurridos.
    private var _lastTimer:Number;
```

```
    //Referencia a la función con la cual se hará la animación.
    private var _formula:Function;

    //Los valores inicial y final eventualmente pueden cambiarse
    //por otros valores que no sean 1 y 0 si queremos obtener el
valor
    //listo para aplicar directamente.
    private var _startValue:Number = START_VALUE;
    private var _endValue:Number = END_VALUE;

    //El valor interpolado. Por defecto es un valor entre 0 y 1.
    private var _interpolation:Number = 0;

    //Valor opcional que se pasa para ciertas clases como Back o
Elastic.
    private var _amplitude:Number = NaN;
    private var _period:Number = NaN;

    //Indica el tipo de loop que realizará el interpolador, por de-
fecto es:
    //LOOP_HOLD.
    private var _loopType:String;

    //Referencia al objeto que por composición manejará la suscrip-
ción,
    //desuscripción y despacho de eventos.
    private var _eventDispatcher:EventDispatcher;

    //Con algunas fórmulas y en algunas situaciones especiales ob-
tendremos
    //un número muy cercano a 0, pero no igual a 0. Por lo tanto,
para asegurarnos
    //Que se disparen los eventos correctamente utilizamos este
número y //una vez alcanzado redondeamos a 0
    private const NEAR_TO_CERO:Number = 0.0000000001;

    //----------------------------------------------------------------
------------
    // Constructor e inicializador
```

```
    //------------------------------------------------------------------
------------

    /**
     * Constructor
     * @param totalMilliseconds : milisegundos totales de la inter-
polación.
     * @param formulaClass: referencia a la clase de una fórmula
determinada.
     * @param methodName: nombre del método a utilizar de la clase.
     */
    public function Interpolator(totalMilliseconds_:Number,
                                 formulaClass:Class=null,
                                 methodName:String = EASE_IN,
                                 loopType_:String=null,
                                 startValue_:Number=NaN,
                                 endValue_:Number=NaN)
    {
      var errorMessage:String;


      //Si este valor no es un número o es negativo, no podrá
usarse como
      //cantidad totales de milisegundos, por lo tanto no podemos
permitir
      //que la aplicación siga corriendo y se detendrá
      if (isInvalidNumber(totalMilliseconds_))
      {
        errorMessage = "Interpolator: el parámetro 'totalMillisec-
onds_'  debe ser del tipo Number y mayor a 0"
        throw new Error(errorMessage);
      }


      //Se asigna el valor inicial de los milisegundos totales.
      _totalMilliseconds = totalMilliseconds_;

      //Se asigna por defecto el primer frame.
      _currentMillisecond = 0;

      //ASignamos la fórmula a utilizar para la interpolación.
      setFormula(formulaClass, methodName);
```

```
      //Asignamos el tipo de loop.
      loopType = loopType_;

      //Asignamos los valores opcionales startValue y endValue.
      _startValue = START_VALUE;
      _endValue = END_VALUE;

      //Asignamos los valores obtenidos del constructor, en caso
que sean NaN,
      //estas dos propiedades mantendrán sus valores por defecto.
      startValue = startValue_;
      endValue = endValue_;

      //Instanciamos el dispatcher para agregar, remover escuchas y
despachar
      //eventos.
      _eventDispatcher = new EventDispatcher();
    }

    //------------------------------------------------------------
------------
    // Configurador de fórmula.
    //------------------------------------------------------------
------------
    /**
     * Este método configura en base a una clase del tipo IFormula
el objeto
     * que contendrá métodos a ser utilizados para calcular la in-
terpolación.
     * @param formulaClass: clase del tipo IFormula.
     * @param methodName: nombre del método a utilizar para la in-
terpolación.
     */
    public function setFormula(formulaClass:Class, method-
Name:String):void
    {
      var iFormula:IFormula = new formulaClass() as IFormula;
```

```
      //Si lo obtenido de la clase que estamos pasando al método no
es
      //un IFormula, disparamos un error.
      if (null  == iFormula)
      {
        throw Error ("Interolator/setFormula: el parámetro: 'formu-
laClass', debe implementar:
'com.alfaomegaeditor.utils.interpolation.easing.IFormula'")
      }


      //Si no es ninguno de estos nombres de métodos, entonces
lanzamos un error.
      var isInvalidMethodName:Boolean = methodName != EASE_IN &&
                                        methodName != EASE_OUT &&
                                        methodName != EASE_IN_OUT
      if (isInvalidMethodName)
      {
        throw new Error ("Interolator/setFormula: el parámetro:
'methodName' debe ser igual a cualquiera de estos valores: '" +
EASE_IN +"', '" + EASE_OUT + "' ó '" + EASE_IN_OUT + "'");
      }


      _formula = iFormula[methodName];
    }


    //-----------------------------------------------------------------
------------
    // Configuradores de milisegundos
    //-----------------------------------------------------------------
------------
    /**
     * Getter de _currentMillisecond.
     */
    public function get currentMillisecond():Number
    {
      if(isNaN(_currentMillisecond))
      {
        _currentMillisecond = 0;
      }
```

```
      return _currentMillisecond;
    }

    /**
     * Utilizando el setter de  currentMillisecond, nos aseguramos
que se asigne
     * un valor entre 0 y la cantidad total de milisegundos.
     */
    public function set currentMillisec-
ond(millisecond_:Number):void
    {
      //Si no es un núnero, detenemos el método.
      if (isNaN(millisecond_))
      {
        return;
      }

      //Si el milisegundo a configurar es muy cercano a 0. Lo re-
dondeamos a
      //cero.
      if (millisecond_ < NEAR_TO_CERO)
      {
        millisecond_ = 0;
      }

      //Si el milisegundo a configurar es muy cercano a los mil-
lisegundos
      //totales. Le asignamos el valor de los milisegundos totales.
      if (millisecond_ > totalMilliseconds - NEAR_TO_CERO)
      {
        millisecond_ = totalMilliseconds;
      }

      _currentMillisecond = millisecond_;
    }

    //Es un getter, nos devuelve la posición relativa del
    //milisegundo actual con un número entre 0 y 1.
```

```
    public function get millisecondRatio():Number
    {
      return (_currentMillisecond / _totalMilliseconds);
    }

    public function set millisecondRatio(value:Number):void
    {
      //Verificamos que el valor sea mayor o igual a 0.
      if (value < 0)
      {
        value = 0
      }

      //Verificamos que el valor sea menor o igual a 1.
      if (value > 1)
      {
        value = 1
      }

      //Asignamos el valor del millisegundo actual utilizando val-
ue.
      _currentMillisecond = _totalMilliseconds * value;
    }

    /**
     * Getter de  totalMilliseconds. El valor de
_totalMilliseconds, nunca
     * puede ser IsNaN, ya que todos los cálculos dependen en gran
medida de
     * su valor. Su valor siempre debe haber sido asignado.
     */
    public function get totalMilliseconds():Number
    {
      if (isNaN(_totalMilliseconds))
      {
        throw new Error("Interpolator/totalMilliseconds: El valor
de _totalMilliseconds debe ser un Number positivo.")
      }
```

```
      //Devolvemos los milisegundos totales.
      return _totalMilliseconds;
    }

    /**
     * Setter de _totalMilliseconds. No tiene un valor por defecto.
     * La asignación de un valor no válido provoca que se detenga
la aplicación.
     */
    public function set totalMilliseconds(value:Number):void
    {
      if (isInvalidNumber(value))
      {
        throw new Error("Interpolator: la propiedad 'totalMillisec-
onds' debe ser del tipo Number y mayor a 0");
      }

      //Antes de asignar el valor tomamos los milisegundos actu-
ales, relativos.
      var ratio:Number = millisecondRatio;

      //Asignamos los milisegundos totales.
      _totalMilliseconds = value;

      //Actualizamos también los milisegundos actuales con el ratio
obtenido.
      //De esta manera podremos variar la duración de la animación
mientras
      //se está ejecutando sin notar saltos en la animación.
      currentMillisecond =  ratio * _totalMilliseconds;
    }

    //---------------------------------------------------------------
------------
    // Propiedades opcionales.
    //---------------------------------------------------------------
------------
    /**
```

```
     * Getter de  amplitude, valor opcional para utilizar en las
fórmulas.
     * Se aplica para algunas fórmulas de clases como Back o Elas-
tic.
     * Determina la amplitud del efecto.
     */
    public function get amplitud():Number
    {
      return _amplitude;
    }

    /**
     * Setter de  amplitude, ver explicación del getter de esta
misma propiedad.
     */
    public function set amplitud(value:Number):void
    {
      //Sino es un número, detenemos el método.
      if (isNaN(value))
      {
        return;
      }
      _amplitude = value;
    }

    /**
     * Getter de  period, valor opcional para fórmulas de la clase
Elastic
     * por ejemplo. Indica la frecuencia en que por ejemplo el
efecto elástico
     * se ejecutará.
     */
    public function get period():Number
    {
      return _period;
    }

    /**
     * Setter de _period, ver el getter de esta misma propiedad.
```

```
     */
    public function set period(value:Number):void
    {
      //Si el valor a asignar no es un número, detenemos el método.
      if (isNaN(value))
      {
        return;
      }

      _period = value;
    }

    /**
     * Getter de _interpolation, propiedad de solo lectura.
     * Devuelve el resultado de la última interpolación realizada.
     */
    public function get interpolation():Number
    {
      return _interpolation;
    }

    /**
     * Getter de _startValue.
     */
    public function get startValue():Number
    {
      return _startValue;
    }

    /**
     * Setter de _starValue.
     */
    public function set startValue(value:Number):void
    {
      //Detenemos el método si el valor no es un número.
      if (isNaN(value))
      {
```

```
        return
      }

      _startValue = value;
    }

    /**
     * Getter de _endValue.
     */
    public function get endValue():Number
    {
      return _endValue;
    }

    /**
     * Setter de _endValue.
     */
    public function set endValue(value:Number):void
    {
      //Detenemos el método si el valor no es un número.
      if (isNaN(value))
      {
        return
      }

      _endValue = value;
    }

    //Propiedad de solo lectura, devuelve el valor interpolado en-
tre:
    //_startValue y _endValue.
    public function get tweenValue():Number
    {
      return calculateTween(_startValue, _endValue);
    }
```

```
    //--------------------------------------------------------------
------------
    // Manejadores de ticks
    //--------------------------------------------------------------
------------
    /**
     * Dispara la próxima interpolación
     * @return devuelve la interpolación actual, un número entre 0
y 1.
     */
    public function nextTick():Number
    {
      //Si el milisegundo actual es igual al total de milisegundos,
      //hemos llegado al último frame. Vemos cómo seguir dependien-
do del valor
      //de loopType.
      if (currentMillisecond == _totalMilliseconds)
      {
        switch(loopType)
        {
          case LOOP_HOLD:
            //Ahorramos todos los cálculos y devolvemos la
            //última interpolación que siempre es 1. Y al detener
el método
            //se dejarán de despachar eventos.
            return 1;

          case LOOP_REPEAT:
            //Reseteamos los valores para volver a comenzar.
            _lastTimer = NaN;
            currentMillisecond = 0;
            break;
        }
      }

      //Tomamos el intervalo a sumar.
      var interval:Number = getLastInterval();

      //Se lo agregamos a los milisegundos actuales.
```

```
      var currentMillisecond :Number =  currentMillisecond + inter-
val;

      //El valor obtenido se lo pasamos al método que se encargará
de realizar
      //la interpolación con el milisegundo obtenido.
      tickAtMillisecond(currentMillisecond_);

      //Devolvemos la interpolación obtenida.
      return _interpolation;
    }

    /**
     * Dispara la interpolación previa.
     * @return devuelve la interpolación actual, un número entre 0
y 1.
     */
    public function prevTick():Number
    {
      //Si el milisegundo actual es 0, ya hemos llegado al comienzo
de la
      //animación. Nos ahorramos los cálculos y devolvemos 0.
      if (_currentMillisecond == 0)
      {
        switch(loopType)
        {
          case LOOP_HOLD:
            //Ahorramos todos los cálculos y devolvemos la
            //primer interpolación que siempre es 0. Y al detener
el método
            //se dejarán de despachar eventos.
            return 0;

          case LOOP_REPEAT:
            //Restauramos los valores necesarios para que empiece
nuevamente.
            _lastTimer = NaN;
            _currentMillisecond = _totalMilliseconds;
            break;
```

```
        }
      }

      //Tomamos el intervalo a restar.
      var interval:Number = getLastInterval();

      //Se lo restamos a los milisegundos actuales.
      var currentMillisecond :Number =  currentMillisecond - inter-
val;

      //El valor obtenido se lo pasamos al método que se encargará
de realizar
      //la interpolación con el milisegundo obtenido.
      tickAtMillisecond(currentMillisecond_);

      //A este momento la interpolación ha sido calculada devolve-
mos su valor.
      return _interpolation;
    }



    /**
     * Ejecuta la interpolación en un milisegundo determinado.
     */
    public function tickAtMillisecond(value:Number):Number
    {
      //Asignamos el valor a milisecond mediante el setter. De esta
manera nos
      //aseguramos que el valor asignado sea válido.
      currentMillisecond = value;

      //Calcula la interpolación.
      _interpolation = _formula(_currentMillisecond,
                                START_VALUE,
                                END_VALUE,
                                _totalMilliseconds,
                                _amplitude,
                                _period);
```

```
      //Despacha el evento correspondiente.
      buildAndDispatchEvent(InterpolatorEvent.TICK);

      //Si el milisegundo actual y la interpolación son muy cerca-
nos a 0
      //Es porque estamos en el comienzo de la animación.
      var isStartMilisecond:Boolean = _currentMillisecond == 0 &&
                                      interpolation <
NEAR_TO_CERO;
      if (isStartMilisecond)
      {
        //Despachamos el evento correspondiente.
        buildAndDispatchEvent(InterpolatorEvent.START);
      }
      else

      //Si el milisegundo actual es muy cercano a los milisegundos
totales
      //y la interpolación es muy cercana a 1. Estamos al final de
la animación.
      //Despachamos el evento correspondiente.
      if (_currentMillisecond == totalMilliseconds &&
          _interpolation > 1 - NEAR_TO_CERO)
      {
        buildAndDispatchEvent(InterpolatorEvent.COMPLETE);
      }

      //Devolvemos el resultado de la interpolación.
      return _interpolation;
    }

    /**
     * Ejecuta la interpolación en un momento relativo de la ani-
mación.
     * El número debe ser un valor entre 0 y 1.
     */
    public function tickAtMillisecondRatio(value:Number):Number
    {
```

```
      //Asignamos el valor a millisecondRatio, es un setter que
corregirá un valor
      //no válido.
      millisecondRatio = value;

      //Al ser asigando millisecondRatio, cambia el valor de cur-
rentSecond
      //en consecuencia.
      tickAtMillisecond(_currentMillisecond);

      //Devolvemos la interpolación obtenida.
      return _interpolation;
    }

    /**
     * Devuelve el intervalo de tiempo desde la última interpol-
ación.
     */
    private function getLastInterval():Number
    {
      //Si  lastTimer no fue asignado, le asignamos el getTimer ac-
tual.
      if (isNaN(_lastTimer))
      {
        _lastTimer = getTimer();
      }

      //Al milisegundo actual le resta el milisegundo obtenido en
la última
      //interpolación.
      var interval:Number = (getTimer() - _lastTimer);

      _lastTimer = getTimer();

      return interval;
    }

    public function get loopType():String
    {
```

```
      return _loopType;
    }

    public function set loopType(value:String):void
    {
      //Si el valor no es ninguno de los predeterminados, asignamos
      //como valor por defecto: LOOP_HOLD.
      if (value != LOOP_HOLD || value != LOOP_REPEAT)
      {
        _loopType = LOOP_HOLD;
      }

      _loopType = value;
    }
    //--------------------------------------------------------------
------------
    // Calculadores del valor interpolado.
    //--------------------------------------------------------------
------------
    /**
     * Dados dos valores, calcula el valor intermedio multiplicado
por la
     * interpolación.
     * @param startValue: valor inicial.
     * @param endValue: valor final.
     * @return devuelve el valor interpolado con la interpolación
actual.
     */
    public function calculateTween(startValue:Number,
endValue:Number):Number
    {
      return (endValue - startValue) * interpolation + startValue;
    }

    //--------------------------------------------------------------
------------
    // Manejadores de escuchas y despachador de eventos.
    //--------------------------------------------------------------
------------
```

```
    /**
     * Agrega un escucha al objeto.
     * @param type: nombre del evento a escuchar.
     * @param listener: método escucha.
     */
    public function addEventListener(type:String, listen-
er:Function):void
    {
      //Al agregar al escucha, asignamos a los valores opcionales
de
      //addEventListener, las constantes de la clase.
      _eventDispatcher.addEventListener(type, listener,
                                        USE_CAPTURE,
                                        PRIORITY,
                                        USE_WEAK_REFERENCE);
    }

    /**
     * Remueve el escucha de objeto.
     * @param type: nombre del evento que se dejará de escuchar.
     * @param listener: método a desuscribir.
     */
    public function removeEventListener(type:String, listen-
er:Function):void
    {
       eventDispatcher.removeEventListener(type, listener,
USE_CAPTURE);
    }

    /**
     * Despacha el evento Tick tomando los valores actuales del ob-
jeto.
     */
    private function buildAndDispatchEvent(eventName:String):void
    {
      //Creamos el evento y le asignamos los valores.
      var event:InterpolatorEvent;
      event = new InterpolatorEvent(eventName,
                                    _currentMillisecond,
```

```
                                    _totalMilliseconds,
                                    millisecondRatio,
                                    interpolation,
                                    tweenValue);

      //Se despacha el evento.
      _eventDispatcher.dispatchEvent(event);
    }

    //------------------------------------------------------------
------------
    // Métodos auxiliares.
    //------------------------------------------------------------
------------
    /**
     * Indica si no es un número válido, ya sea que no sea un
número o que
     * sea un número negativo.
     * @param number: el valor a verificar.
     * @return devuelve true, sino es válido.
     */
    private function isInvalidNumber(number:Number):Boolean
    {
      return (isNaN(number) || 0 > number);
    }
  }
}
```

Y ahora, haremos nuestra clase cliente que será la clase del documento. La llamaremos `MainInterpolation`, y con ella haremos distintos tipos de pruebas con nuestra clase recién creada. Ante todo, realizaremos lo siguiente: moveremos de izquierda a derecha un rectángulo.

Para ello utilizaremos este código:

```
package
{
```

```
import flash.events.Event;
import flash.display.Sprite;
import flash.utils.getTimer;

import com.alfaomegaeditor.utils.interpolation.Interpolator;
import com.alfaomegaeditor.events.InterpolatorEvent;

public class MainInterpolation extends Sprite
{
  private var _interpolator:Interpolator;
  private var _rectangle:Sprite;
  private var _initialXPosition:Number;
  private var _finalXPosition:Number;

  public function MainInterpolation():void
  {
    initialize();
  }

  private function initialize():void
  {
    //Creamos un rectángulo.
    _rectangle = new Sprite();
    _rectangle.graphics.beginFill(0xFF0000);
    _rectangle.graphics.drawRect(-10, -50, 20, 100);

    //Definimos la posición X inicial y final.
    _initialXPosition = 100;
    _finalXPosition = 500;

    //Ubicamos al rectángulo.
    _rectangle.x = _initialXPosition;
    _rectangle.y = 150;

    //Lo agregamos al escenario.
    addChild(_rectangle);
```

```
      //Creamos la interpolación.
      _interpolator = new Interpolator(1000,
                                       Interpolator.BOUNCE_FORMULA,
                                       Interpolator.EASE_OUT,
                                       Interpolator.LOOP_HOLD,
                                       0,
                                       750);

      //Agregamos un escucha al Event.ENTER_FRAME
      this.addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
    {
      _interpolator.nextTick();
      _rectangle.x = _interpolator.tweenValue;
    }
  }
}
```

Nuestras carpetas y clases, en este momento quedarían así:

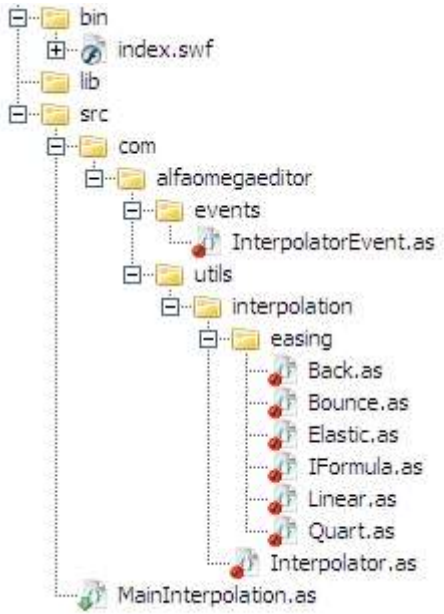

Fig. 1 Organización de carpetas y archivos de la aplicación.

Y si probamos nuestro .SWF, veríamos un rectángulo moviéndose de izquierda a derecha con un efecto de rebote al final de la animación.

Aplicación de interpolación a diferentes propiedades

Como ya lo hemos planteado, el `Interpolator` puede ser utilizado obteniendo sus valores directamente o por eventos. Otra ventaja adicional es que no debemos asignar o pasar por referencia un objeto. Un solo objeto `Interpolator` puede aplicar un efecto a distintos objetos, a distintas propiedades sin necesidad de pre-determinarlo cuando lo creamos. Antes de pasar a mejoras y efectos que podemos realizar, analicemos un poco más en detalle que está sucediendo a cada *tick*.

En esta porción de código correspondiente al escucha del `Event.ENTER_FRAME`, luego de haber sido creado el `Interpolator`, observamos lo siguiente:

```
private function onEnterFrame(event:Event):void
{
  _interpolator.nextTick();
  _rectangle.x = _interpolator.tweenValue;
}
```

La línea de código resaltada es la que provoca que la animación se produzca. Cada vez que se ejecuta el `nextTick`, a grandes rasgos, sucede lo siguiente:

1. Se verifica si se llegó al final de la animación.
    a. Si se llegó al final de la animación, dependiendo del valor de `loopType`, se repite la animación.
2. Se toma el intervalo de tiempo transcurrido.
3. Al milisegundo actual, se le agrega el intervalo transcurrido.
4. Se llama a `tickAtMillisecond` con el milisegundo obtenido.
5. El método `tickAtMillisecond` asigna el valor tomado a `currentMillisecond`. Este último es un *setter* que se encarga de dar un valor válido a `_currentMillisecond`.
6. Calcula la interpolación sobre la base de: `_currentMillisecond` y `_totalMilliseconds` y los valores inicial y final con 0 y 1.
7. El método `tickAtMillisecond` de acuerdo con los chequeos que se pueden ver en el código, determina si se debe despachar algún evento, y si es necesario lo despacha.
8. Devuelve la interpolación obtenida.
9. Y el método `nextTick` devuelve el valor que obtuvo a su vez.

Cada vez que se ejecuta un *tick*, se toma el intervalo transcurrido desde la última vez que se ejecutó el método `getLastTimer`. Este método es llamado cada vez que cualquiera de los manejadores de *tick* provoca un cambio en el milisegundo actual.
En el caso de `nexTick`, cuando obtiene el valor del intervalo transcurrido, lo agrega al milisegundo actual. Esto da por resultado un cálculo acumulativo del tiempo

transcurrido. Luego, el método `tickAtMillisecond` será llamado y calculará con los datos y propiedades configuradas un valor entre 0 y 1 correspondiente. Y allí reside toda la magia.

Nuestra clase tiene la flexibilidad de poder variarle los parámetros en tiempo de ejecución, simplificando la forma en la que podemos aplicar efectos. Veamos algo interesante. Podemos asignar un par de propiedades con un valor inicial y otro final, y al recibir el evento, obtendremos el valor a ser aplicado para lograr el efecto, utilizando la propiedad: `tweenValue`. Para afectar otras propiedades como la rotación por el eje Z, utilizamos el método: `calculateTween`.

La sección de código donde agregamos esta línea quedaría así:

```
private function onEnterFrame(event:Event):void
{
   _interpolator.nextTick();
   _rectangle.x = _interpolator.tweenValue;
   _rectangle.rotationZ = _interpolator.calculateTween(0, 360);
 }
```

Esta animación provocará que el rectángulo se mueva de izquierda a derecha, rotando mientras se desplaza y provocando además un rebote en la rotación.

Mejora de aplicación de interpolación a varias propiedades

Si bien no hemos hecho la clase en este caso paso a paso. Veremos cómo modificar y perfeccionar las clases que ya tenemos. Tenemos la opción de aplicar la interpolación a muchas propiedades utilizando el método `calculateTween`. Podríamos mejorar aún más esta opción sin descartar el método `calculateTween`, agregando un objeto del tipo `Object` del cual se obtengan la interpolación de un par de variables lista para ser aplicada. Entonces, de esta manera, cargamos con la responsabilidad de hacer los cálculos a la clase `Interpolator`. Y en vez de tener solamente dos propiedades (`startValue` y `endValue`), tendremos un `Object` que contiene un conjunto de pares de estas variables, y otro objeto en donde se guardan sus interpolaciones.

La idea es que `Interpolator` tendrá un objeto en el cual se podrán guardar pares de propiedades de comienzo y fin con un identificador. Luego, al realizarse la interpolación se la podría recuperar lista para ser aplicada, utilizando ese identifica-

dor. Incluso este objeto con las interpolaciones calculadas puede ser transmitido en el evento.

Remociones de código

Cambios en la clase Interpolator

Para hacer esta modificación comenzaremos removiendo las propiedades: `_startValue`, `_endValue`, los *getters* y *setters* correspondientes el `getter tweenValue`, y en todas partes donde se los utilice, como en el constructor y en el método `buildAndDispatchEvent`. No remover nada de **`calculateInterpolation`.**

Cambios en la clase InterpolatorEvent

Removeremos a `tweenValue_` en el parámetro del constructor, la asignación de `_tweenValue`, la propiedad `_tweenValue` y su *getter.*

Cambios en la clase MainInterpolation

En `MainInterpolation` removemos en el llamado al constructor, los parámetros a 0 y 750. Y en el método `onTick`, removemos la línea: `_rectangle.x = _interpolator.tweenValue;`

Probamos nuestra película y deberíamos ver solamente la animación de rotación.

Agregando la funcionalidad de cálculo de interpolación de varias propiedades

En la clase InterpolatorEvent

En negrita, veremos resaltado el código agregado, luego de haber removido todo el código relacionado con `tweenValue`.

```
package com.alfaomegaeditor.events
{
  import flash.events.Event;
  import flash.utils.ByteArray;
  public class InterpolatorEvent extends Event
  {
    /*
    * Esta clase tiene la responsabilidad de contener los datos de
una
    * interpolación para ser utilizado por un despachador de even-
tos.
    */

    //Evento que se dispara a cada tick.
    public static const TICK:String = "interpolationTick";

    //Evento que se dispara al ejecutarse la primera interpolación.
    public static const START:String = "interpolationStart";

    //Evento que se dispara al ejecutarse la última interpolación.
    public static const COMPLETE:String = "interpolationComplete";

    //Milisegundo actual de la interpolación.
    private var _currentMillisecond:Number;

    //Milisegundos totales.
    private var _totalMilliseconds:Number;

    //Valor entre 1 y 0 que hace referencia al milisegundo actual
en forma
    //relativa.
    private var _milisecondRatio:Number;

    //Valor de la interpolación.
    private var _interpolation:Number;
```

```
    //Es un objeto del tipo Object que tiene el valor de interpol-
aciones
    //asociadas a un isString.
    private var _tweenValues:Object;

    /**
     * Constructor.
     * @param type: el nombre del evento al cual suscribirse.
     */
    public function InterpolatorEvent(type:String,
                                      currentMillisecond_:Number,
                                      totalMillieconds_:Number,
                                      milisecondRatio_:Number,
                                      interpolation_:Number,
                                      tweenValues_:Object)
    {
      super(type);
      _currentMillisecond = currentMillisecond_;
      _totalMilliseconds = totalMillieconds_;
      _milisecondRatio = milisecondRatio_;
      _interpolation = interpolation_;
      _tweenValues = tweenValues_;
    }

    /**
     * Milisegundo actual, propiedad de solo lectura.
     */
    public function get currentMillisecond():Number
    {
      return _currentMillisecond;
    }

    /**
     * Milisegundos totales, propiedad de solo lectura.
     */
    public function get totalMilliseconds():Number
    {
```

```
      return _totalMilliseconds;
    }

    /**
     * Milisegundo actuales en forma relativa,representado por un
valor entre
     * 0 y 1. Propiedad de solo lectura.
     */
    public function get milisecondRatio():Number
    {
      return _milisecondRatio;
    }

    /**
     * El resultado de la interpolación, propiedad de solo lectura.
     */
    public function get interpolation():Number
    {
      return _interpolation;
    }

    /**
     * En este objeto se encuentran las interpolaciones asociadas
     * a un isString.
     */
    public function get tweenValues():Object
    {
      return cloneObject(_tweenValues) as Object;
    }

    /**
     * Dados dos valores calcula el valor intermedio multiplicado
por la
     * interpolación.
     * @param startValue: valor inicial.
     * @param endValue: valor final.
     * @return devuelve el valor interpolado con la interpolación
actual.
```

```
     */
    public function calculateInterpolation(startValue:Number,
                                           endValue:Number):Number
    {
      return (endValue - startValue) * interpolation + startValue;
    }

    /**
     * Devuelve el valor de acuerdo a un identificador.
     * @param idString: nombre del identificador.
     * @return devuelve el valor de la interpolación corre-
spondiente.
     */
    public function getTweenByName(idString:String):Number
    {
      return _tweenValues[idString];
    }

    /**
     * Clona al evento, haciendo un clon también de los objetos
complejos
     * que puediera contener.
     * @return
     */
    override public function clone():Event
    {
      return cloneObject(this) as Event;
    }

    /**
     * Este método clona un objeto y todos los objetos que con-
tenga.
     * @param source: objeto a clonar.
     * @return devuelve el objeto clonado.
     */
    private function cloneObject(source:*):*
    {
      var byteArray:ByteArray = new ByteArray();
```

```
            byteArray.writeObject(source);
            byteArray.position = 0;
            return Event(byteArray.readObject());
        }
    }
}
```

En la clase Interpolator

Agregaremos dos propiedades, una es `_tweenProperties`, donde guardaremos las propiedades iniciales y finales; y la otra, es el objeto: `_tweenValues`, donde guardaremos las interpolaciones calculadas correspondientes.

La propiedad `_tweenValues` es la que se propagará en el evento. De esta manera, podremos hacer recaer en el `Interpolator` los cálculos de estas propiedades. También necesitaremos agregar métodos para agregar, remover y obtener los pares de propiedades. Agregaremos además el mismo método que utilizamos en la clase `InterpolatorEvent` para clonar objetos. Para ello no debemos olvidar importar la clase de ActionScript 3.0, `flash.utils.ByteArray`.

Ahora veremos los métodos que agregaremos:

- **`public function setTweenProperties(stringId:String, startValue:Number, endValue:Number):void`**: configura un par de propiedades de inicio y fin.
- **`public function setStartProperty(stringId:String, startValue:Number):void`**: configura la propiedad inicial de un identificador determinado.
- **`public function setEndProperty(stringId:String, endValue:Number):void`**: configura la propiedad final de un identificador determinado.
- **`public function deleteTweenProperties(stringId:String):void`**: borra un par de propiedades al `stringId` correspondiente.
- **`public function getTweenValues():Object void`**: Devuelve un clon del objeto con todas las interpolaciones.

- **`public function getTweenByName(stringId:String):Number void:`** devuelve el valor de la última interpolación correspondiente al `stringId` correspondiente.

En el método `buildAndDispatchEvent`, agregamos el clon de `getTweenValues`.

Y la clase `Interpolator` quedaría de la siguiente manera:

```
package com.alfaomegaeditor.utils.interpolation
{
  /**
   * Clase encargada de devolver y disparar eventos con los datos
   * de una interpolación.
   */

  import flash.events.EventDispatcher;
  import flash.utils.getTimer;
  import flash.utils.ByteArray;

  import com.alfaomegaeditor.events.InterpolatorEvent;
  import com.alfaomegaeditor.utils.interpolation.easing.Back;
  import com.alfaomegaeditor.utils.interpolation.easing.Bounce;
  import com.alfaomegaeditor.utils.interpolation.easing.Elastic;
  import com.alfaomegaeditor.utils.interpolation.easing.Linear;
  import com.alfaomegaeditor.utils.interpolation.easing.Quart;
  import com.alfaomegaeditor.utils.interpolation.easing.IFormula;

  public class Interpolator
  {
    //Referencia a las clases para calcular las fórmulas.
    public static const BACK_FORMULA:Class = Back;
    public static const BOUNCE_FORMULA:Class = Bounce;
    public static const ELASTIC_FORMULA:Class = Elastic;
    public static const LINEAR_FORMULA:Class = Linear;
    public static const QUART_FORMULA:Class = Quart;
```

```
    //Nombres de los métodos a llamar de las clases del tipo IFor-
mula.
    public static const EASE_IN:String = "easeIn";
    public static const EASE_OUT:String = "easeOut";
    public static const EASE_IN_OUT:String = "easeInOut";

    public static const LOOP_HOLD:String = "loopHold";
    public static const LOOP_REPEAT:String = "loopRepeat";

    //Constantes utilizadas para agregar y remover escuchas.
    private static const USE_CAPTURE:Boolean = false;
    private static const PRIORITY:uint = 0;
    private static const USE_WEAK_REFERENCE:Boolean = true;

    //Valor inicial y final por defecto son 0 y 1.
    private static const START_VALUE:Number = 0;
    private static const END_VALUE:Number = 1;

    //Los milisegundos actuales y totales.
    private var _currentMillisecond:Number=0;
    private var _totalMilliseconds:Number;

    //Se utiliza mientras se está realizando la animación, en el
que se guarda cuáles son
    //los últimos milisegundos transcurridos.
    private var _lastTimer:Number;

    //Referencia a la función con la cual se hará la animación.
    private var _formula:Function;

    //Este objeto opcional contiene propiedades con un Array de 2
posiciones,
    //correspondientes a la propiedad inicial y final.
    private var _tweenProperties:Object = new Object();

    //En este objeto se aplican los cálculos de las propiedades
iniciales y finales
    //correspondientes.
```

```
    private var _tweenValues:Object = new Object();

    //El valor interpolado. Por defecto es un valor entre 0 y 1.
    private var _interpolation:Number = 0;

    //Valor opcional que se pasa para ciertas clases como Back o
Elastic.
    private var _amplitude:Number = NaN;
    private var _period:Number = NaN;

    //Indica el tipo de loop que realizará el interpolador, por de-
fecto es:
    //LOOP_HOLD.
    private var _loopType:String;

    //Referencia al objeto que por composición manejará la suscrip-
ción,
    //desuscripción y despacho de eventos.
    private var _eventDispatcher:EventDispatcher;

    //Con algunas fórmulas y en algunas situaciones especiales ob-
tendremos
    //un número muy cercano a 0, pero no igual a 0. Por lo tanto
para asegurarnos
    //Que que se disparen los eventos correctamente utilizamos este
número y //una vez alcanzado redondeamos a 0
    private const NEAR_TO_CERO:Number = 0.0000000001;

    //-------------------------------------------------------------
------------
    // Constructor e inicializador
    //-------------------------------------------------------------
------------
    /**
     * Constructor
     * @param totalMilliseconds : milisegundos totales de la inter-
polación.
     * @param formulaClass: referencia a la clase de una fórmula
determinada.
     * @param methodName: nombre del método a utilizar de la clase.
     */
```

```
    public function Interpolator(totalMilliseconds_:Number,
                                 formulaClass:Class=null,
                                 methodName:String = EASE_IN,
                                 loopType_:String=null,
                                 startValue_:Number=NaN,
                                 endValue_:Number=NaN)
    {
      var errorMessage:String;

      //Si este valor no es un número o es negativo, no podrá
usarse como
      //cantidad totales de milisegundos, por lo tanto no podemos
permitir
      //que la aplicación siga corriendo y se detendrá
      if (isInvalidNumber(totalMilliseconds_))
      {
        errorMessage = "Interpolator: el parámetro 'totalMillisec-
onds_'  debe ser del tipo Number y mayor a 0"
        throw new Error(errorMessage);
      }

      //Se asigna el valor inicial de los milisegundos totales.
      _totalMilliseconds = totalMilliseconds_;

      //Se asigna por defecto el primer frame.
      _currentMillisecond = 0;

      //Asignamos la fórmula a utilizar para la interpolación.
      setFormula(formulaClass, methodName);

      //Asignamos el tipo de loop.
      loopType = loopType_;

      //Instanciamos el dispatcher para agregar, remover escuchas y
despachar
      //eventos.
      _eventDispatcher = new EventDispatcher();
    }
```

```
    //------------------------------------------------------------
------------
    // Configurador de fórmula.
    //------------------------------------------------------------
------------
    /**
     * Este método configura sobre la base de una clase del tipo
IFormula el objeto
     * que contendrá métodos a ser utilizados para calcular la in-
terpolación.
     * @param formulaClass: clase del tipo IFormula.
     * @param methodName: nombre del método a utilizar para la in-
terpolación.
     */
    public function setFormula(formulaClass:Class, method-
Name:String):void
    {
      var iFormula:IFormula = new formulaClass() as IFormula;

      //Si lo obtenido de la clase que estamos pasando al método no
es
      //un IFormula, disparamos un error.
      if (null  == iFormula)
      {
        throw Error ("Interolator/setFormula: el parámetro: 'formu-
laClass', debe implementar:
'com.alfaomegaeditor.utils.interpolation.easing.IFormula'")
      }

      //Si no es ninguno de estos nombres de métodos, entonces
lanzamos un error.
      var isInvalidMethodName:Boolean = methodName != EASE_IN &&
                                        methodName != EASE_OUT &&
                                        methodName != EASE_IN_OUT
      if (isInvalidMethodName)
      {
        throw new Error ("Interolator/setFormula: el parámetro:
'methodName' debe ser igual a cualquiera de estos valores: '" +
EASE_IN +"', '" + EASE_OUT + "' ó '" + EASE_IN_OUT + "'");
      }
```

```
      _formula = iFormula[methodName];
    }

    //------------------------------------------------------------
------------
    // Configuradores de milisegundos
    //------------------------------------------------------------
------------
    /**
     * Getter de _currentMillisecond.
     */
    public function get currentMillisecond():Number
    {
      if(isNaN(_currentMillisecond))
      {
        _currentMillisecond = 0;
      }

      return _currentMillisecond;
    }

    /**
     * Utilizando el setter de  currentMillisecond, nos aseguramos
que se asigne
     * un valor entre 0 y la cantidad total de milisegundos.
     */
    public function set currentMillisec-
ond(millisecond_:Number):void
    {
      //Si no es un núnero, detenemos el método.
      if (isNaN(millisecond_))
      {
        return;
      }

      //Si el milisegundo a configurar es muy cercano a 0. Lo re-
dondeamos a
      //cero.
```

```
      if (millisecond_ < NEAR_TO_CERO)
      {
        millisecond_ = 0;
      }

      //Si el milisegundo a configurar es muy cercano a los mil-
lisegundos
      //totales. Le asignamos el valor de los milisegundos totales.
      if (millisecond_ > totalMilliseconds - NEAR_TO_CERO)
      {
        millisecond_ = totalMilliseconds;
      }

      _currentMillisecond = millisecond_;
    }

    //Es un getter, nos devuelve la posición relativa del
    //milisegundo actual con un número entre 0 y 1.
    public function get millisecondRatio():Number
    {
      return (_currentMillisecond / _totalMilliseconds);
    }

    public function set millisecondRatio(value:Number):void
    {
      //Verificamos que el valor sea mayor o igual a 0.
      if (value < 0)
      {
        value = 0
      }

      //Verificamos que el valor sea menor o igual a 1.
      if (value > 1)
      {
        value = 1
      }
```

```
      //Asignamos el valor del millisegundo actual utilizando val-
ue.
      _currentMillisecond = _totalMilliseconds * value;
    }


    /**
     * Getter de  totalMilliseconds. El valor de
_totalMilliseconds, nunca
     * puede ser IsNaN, ya que todos los cálculos dependen en gran
medida de
     * su valor. Su valor siempre debe haber sido asignado.
     */
    public function get totalMilliseconds():Number
    {
      if (isNaN(_totalMilliseconds))
      {
        throw new Error("Interpolator/totalMilliseconds: El valor
de _totalMilliseconds debe ser un Number positivo.")
      }


      //Devolvemos los milisegundos totales.
      return _totalMilliseconds;
    }


    /**
     * Setter de _totalMilliseconds. No tiene un valor por defecto.
     * La asignación de un valor no válido provoca que se detenga
la aplicación.
     */
    public function set totalMilliseconds(value:Number):void
    {
      if (isInvalidNumber(value))
      {
        throw new Error("Interpolator: la propiedad 'totalMillisec-
onds' debe ser del tipo Number y mayor a 0");
      }


      //Antes de asignar el valor tomamos los milisegundos actu-
ales, relativos.
```

```
      var ratio:Number = millisecondRatio;

      //Asignamos los milisegundos totales.
      _totalMilliseconds = value;

      //Actualizamos también los milisegundos actuales con el ratio
obtenido.
      //De esta manera podremos variar la duración de la animación
mientras
      //se está ejecutando sin notar saltos en la animación.
      currentMillisecond =  ratio * _totalMilliseconds;
    }

    //------------------------------------------------------------
------------
    // Propiedades opcionales.
    //------------------------------------------------------------
------------
    /**
     * Getter de  amplitude, valor opcional para utilizar en las
fórmulas.
     * Se aplica para algunas fórmulas de clases como Back o Elas-
tic.
     * Determina la amplitud del efecto.
     */
    public function get amplitud():Number
    {
      return _amplitude;
    }

    /**
     * Setter de  amplitude, ver explicación del getter de esta
misma propiedad.
     */
    public function set amplitud(value:Number):void
    {
      //Sino es un número, detenemos el método.
      if (isNaN(value))
      {
```

```
        return;
      }
      _amplitude = value;
    }

    /**
     * Getter de  period, valor opcional para fórmulas de la clase
Elastic
     * por ejemplo. Indica la frecuencia en que por ejemplo el
efecto elástico
     * se ejecutará.
     */
    public function get period():Number
    {
      return _period;
    }

    /**
     * Setter de _period, ver el getter de esta misma propiedad.
     */
    public function set period(value:Number):void
    {
      //Si el valor a asignar no es un número, detenemos el método.
      if (isNaN(value))
      {
        return;
      }

      _period = value;
    }

    /**
     * Getter de _interpolation, propiedad de solo lectura.
     * Devuelve el resultado de la última interpolación realizada.
     */
    public function get interpolation():Number
    {
```

```
      return _interpolation;
    }

    //--------------------------------------------------------------
------------
    // Manejadores de ticks
    //--------------------------------------------------------------
------------
    /**
     * Dispara la próxima interpolación
     * @return devuelve la interpolación actual, un número entre 0
y 1.
     */
    public function nextTick():Number
    {
      //Si el milisegundo actual es igual al total de milisegundos,
      //hemos llegado al último frame. Vemos cómo seguir dependien-
do del valor
      //de loopType.
      if (currentMillisecond == _totalMilliseconds)
      {
        switch(loopType)
        {
          case LOOP_HOLD:
            //Ahorramos todos los cálculos y devolvemos la
            //última interpolación que siempre es 1. Y al detener
el método
            //se dejarán de despachar eventos.
            return 1;

          case LOOP_REPEAT:
            //Reseteamos los valores para volver a comenzar.
            _lastTimer = NaN;
            currentMillisecond = 0;
            break;
        }
      }

      //Tomamos el intervalo a sumar.
```

```
      var interval:Number = getLastInterval();

      //Se lo agregamos a los milisegundos actuales.
      var currentMillisecond :Number =  currentMillisecond + inter-
val;

      //El valor obtenido se lo pasamos al método que se encargará
de realizar
      //la interpolación con el milisegundo obtenido.
      tickAtMillisecond(currentMillisecond_);

      //Devolvemos la interpolación obtenida.
      return _interpolation;
    }

    /**
     * Dispara la interpolación previa.
     * @return devuelve la interpolación actual, un número entre 0
y 1.
     */
    public function prevTick():Number
    {
      //Si el milisegundo actual es 0, ya hemos llegado al comienzo
de la
      //animación. Nos ahorramos los cálculos y devolvemos 0.
      if (_currentMillisecond == 0)
      {
        switch(loopType)
        {
          case LOOP_HOLD:
            //Ahorramos todos los cálculos y devolvemos la
            //primer interpolación que siempre es 0. Y al detener
el método
            //se dejarán de despachar eventos.
            return 0;

          case LOOP_REPEAT:
            //Restauramos los valores necesarios para que empiece
nuevamente.
```

```
            _lastTimer = NaN;
            _currentMillisecond = _totalMilliseconds;
            break;
        }
      }

      //Tomamos el intervalo a restar.
      var interval:Number = getLastInterval();

      //Se lo restamos a los milisegundos actuales.
      var currentMillisecond :Number =  currentMillisecond - inter-
val;

      //El valor obtenido se lo pasamos al método que se encargará
de realizar
      //la interpolación con el milisegundo obtenido.
      tickAtMillisecond(currentMillisecond_);

      //A este momento la interpolación ha sido calculada, devolve-
mos su valor.
      return _interpolation;
    }



    /**
     * Ejecuta la interpolación en un milisegundo determinado.
     */
    public function tickAtMillisecond(value:Number):Number
    {
      //Asignamos el valor a milisecond mediante el setter. De esta
manera nos
      //aseguramos que el valor asignado sea válido.
      currentMillisecond = value;

      //Calcula la interpolación.
      _interpolation = _formula(_currentMillisecond,
                                START_VALUE,
                                END_VALUE,
```

```
                                    _totalMilliseconds,
                                    _amplitude,
                                    _period);

      //Configura los valores de: _tweenValues.
      var idString:String;
      var tweenProperties:Object;
      for (idString in _tweenProperties)
      {
        tweenProperties = _tweenProperties[idString];
         tweenValues[idString] = calculat-
eTween(tweenProperties.startValue,
                                                  tweenProper-
ties.endValue);
      }

      //Despacha el evento correspondiente.
      buildAndDispatchEvent(InterpolatorEvent.TICK);

      //Si el milisegundo actual y la interpolación son muy cerca-
nos a 0
      //Es porque estamos en el comienzo de la animación.
      var isStartMilisecond:Boolean = _currentMillisecond == 0 &&
                                      interpolation <
NEAR_TO_CERO;
      if (isStartMilisecond)
      {
        //Despachamos el evento correspondiente.
        buildAndDispatchEvent(InterpolatorEvent.START);
      }
      else

      //Si el milisegundo actual es muy cercano a los milisegundos
totales
      //y la interpolación es muy cercana a 1. Estamos al final de
la animación.
      //Despachamos el evento correspondiente.
      if (_currentMillisecond == totalMilliseconds &&
          _interpolation > 1 - NEAR_TO_CERO)
```

```
      {
        buildAndDispatchEvent(InterpolatorEvent.COMPLETE);
      }

      //Devolvemos el resultado de la interpolación.
      return _interpolation;
    }

    /**
     * Ejecuta la interpolación en un momento relativo de la ani-
mación.
     * El número debe ser un valor entre 0 y 1.
     */
    public function tickAtMillisecondRatio(value:Number):Number
    {
      //Asignamos el valor a millisecondRatio, es un setter que
corregirá un valor
      //no válido.
      millisecondRatio = value;

      //Al ser asigando millisecondRatio, cambia el valor de cur-
rentSecond
      //en consecuencia.
      tickAtMillisecond(_currentMillisecond);

      //Devolvemos la interpolación obtenida.
      return _interpolation;
    }

    /**
     * Devuelve el intervalo de tiempo desde la última interpol-
ación.
     */
    private function getLastInterval():Number
    {
      //Si  lastTimer no fue asignado, le asignamos el getTimer ac-
tual.
      if (isNaN(_lastTimer))
      {
```

```
        _lastTimer = getTimer();
      }

      //Al milisegundo actual le resta el milisegundo obtenido en
la última
      //interpolación.
      var interval:Number = (getTimer() - _lastTimer);

      _lastTimer = getTimer();

      return interval;
    }

    public function get loopType():String
    {
      return _loopType;
    }

    public function set loopType(value:String):void
    {
      //Si el valor no es ninguno de los predeterminados, asignamos
      //como valor por defecto: LOOP_HOLD.
      if (value != LOOP_HOLD || value != LOOP_REPEAT)
      {
        _loopType = LOOP_HOLD;
      }

      _loopType = value;
    }
    //--------------------------------------------------------------
------------
    // Calculadores el valor interpolado.
    //--------------------------------------------------------------
------------
    /**
     * Dados dos valores calcula el valor intermedio multiplicado
por la
     * interpolación.
```

```
     * @param startValue: valor inicial.
     * @param endValue: valor final.
     * @return devuelve el valor interpolado con la interpolación
actual.
     */
    public function calculateTween(startValue:Number,
endValue:Number):Number
    {
      return (endValue - startValue) * interpolation + startValue;
    }

    /**
     * Configura un par de propiedades de inicio y final para in-
terpolar.
     * @param stringId: con el se identificará al de valores.
     * @param startValue: valor inicial.
     * @param endValue: valor final.
     */
    public function setTweenProperties(stringId:String,
                                       startValue:Number,
                                       endValue:Number):void
    {
      var properties:Object = new Object();
      properties.startValue = startValue;
      properties.endValue = endValue;
      _tweenProperties[stringId] = properties;
    }

    /**
     * Configura la propiedad inicial de un identificador determi-
nado en el
     * objeto: _tweenProperties
     * @param stringId - Identificador del objeto.
     * @param startValue - Valor que se quiere asignar.
     */
    public function setStartProperty(stringId:String,
                                     startValue:Number):void
```

```
    {
      if (isNaN(startValue))
      {
        return
      }

      var properties:Object = _tweenProperties[stringId];
      if (properties == undefined)
      {
        return
      }

      properties.startValue = startValue;
    }

    /**
     * Configura la propiedad final de un identificador determinado
en el
     * objeto: _tweenProperties
     * @param stringId - Idenficador del objeto.
     * @param startValue - Valor que se quiere asignar.
     */
    public function setEndProperty(stringId:String,
                                   endValue:Number):void
    {
      if (isNaN(endValue))
      {
        return
      }

      var properties:Object = _tweenProperties[stringId];
      if (properties == undefined)
      {
        return
      }

      properties.endValue = endValue;
```

```
    }

    /**
     * Borra un par de valores inicial/final en base al stringId.
     * @param stringId: identificador del par de propiedades a
eliminar.
     */
    public function deleteTweenProperty(stringId:String):void
    {
      _tweenProperties[stringId] = null;
      _tweenValues[stringId] = null;

      delete _tweenProperties[stringId];
      delete _tweenValues[stringId];
    }

    /**
     * Devuelve un clon de los pares de valores inicial/final.
     * @return clon del objeto: _tweenProperties.
     */
    public function getTweenValues():Object
    {
      return cloneObject(_tweenProperties);
    }

    /**
     * Devuelve la interpolación correspondiente a un identifica-
dor.
     * @param stringId: nombre del identificador.
     * @return: valor interpolado del par de variables ini-
cial/final
     * correspondientes
     */
    public function getTweenByName(stringId:String):Number
    {
      return _tweenValues[stringId];
    }
```

```
    //--------------------------------------------------------------
------------
    // Manejadores de escuchas y despachador de eventos.
    //--------------------------------------------------------------
------------
    /**
     * Agrega un escucha al objeto.
     * @param type: nombre del evento a escuchar.
     * @param listener: método escucha.
     */
    public function addEventListener(type:String, listen-
er:Function):void
    {
      //Al agregar al escucha, asignamos a los valores opcionales
de
      //addEventListener, las constantes de la clase.
      _eventDispatcher.addEventListener(type, listener,
                                        USE_CAPTURE,
                                        PRIORITY,
                                        USE_WEAK_REFERENCE);
    }

    /**
     * Remueve el escucha de objeto.
     * @param type: nombre del evento que se dejará de escuchar.
     * @param listener: método a desuscribir.
     */
    public function removeEventListener(type:String, listen-
er:Function):void
    {
       eventDispatcher.removeEventListener(type, listener,
USE_CAPTURE);
    }

    /**
     * Despacha el evento Tick tomando los valores actuales del ob-
jeto.
     */
    private function buildAndDispatchEvent(eventName:String):void
```

```
    {
      //Creamos el evento y le asignamos los valores.
      var event:InterpolatorEvent;
      event = new InterpolatorEvent(eventName,
                                    _currentMillisecond,
                                    _totalMilliseconds,
                                    millisecondRatio,
                                    interpolation,
                                    cloneObject( tweenValues) as
Object);

      //Se despacha el evento.
      _eventDispatcher.dispatchEvent(event);
    }

    //---------------------------------------------------------------
------------
    // Métodos auxiliares.
    //---------------------------------------------------------------
------------
    /**
     * Indica si no es un número válido, ya sea que no es un número
o que
     * es un número negativo.
     * @param number: el valor a verificar.
     * @return devuelve true sino es válido.
     */
    private function isInvalidNumber(number:Number):Boolean
    {
      return (isNaN(number) || 0 > number);
    }

    /**
     * Este método permite crear un clon de objeto y los objetos
que contiene.
     * @param source: objeto a copiar.
     * @return: Devuelve una copia del objeto y de los objetos que
éste pudiera
     * contener.
```

```
     */
    public function cloneObject(source:Object):*
    {
      var byteArray:ByteArray = new ByteArray();
      byteArray.writeObject(source);
      byteArray.position = 0;
      return(byteArray.readObject());
    }
  }
}
```

Ahora, en la clase `MainInterpolation`, a continuación de la creación del objeto `Interpolator`, configuramos las propiedades que queremos que ya vengan calculadas:

```
//Creamos la interpolación.
      _interpolator = new Interpolator(1000,
                                       Interpolator.BOUNCE_FORMULA,
                                       Interpolator.EASE_OUT,
                                       Interpolator.LOOP_HOLD);

      _interpolator.setTweenProperties("x", 0, 750);
      _interpolator.setTweenProperties("rotationZ", 0, 360);
```

Y en el método `onEnterFrame` simplemente obtenemos los valores y aplicamos directamente:

```
private function onEnterFrame(event:Event):void
{
  _interpolator.nextTick();
  _rectangle.x = _interpolator.getTweenByName("x");
  _rectangle.rotationZ = _interpolator.getTweenByName("rotationZ");
}
```

Si probamos la película, veremos el rectángulo pasar de izquierda a derecha, mientras va rotando, rebota contra el extremo derecho, y luego se detiene.

Probando la escucha de eventos de InterpolatorEvent

En la clase `MainInterpolation`, agregaremos esta línea donde importamos las clases `import com.alfaomegaeditor.events.InterpolatorEvent;`

Y ahora, agregaremos escuchas a los tres eventos de `InterpolationEvent`.

Por lo que nuestra clásica `MainInterpolation`, tendrá cuatro escuchas:

- **`onEnterFrame`**: al evento `Event.ENTER_FRAME` para ejecutar el `nextTick`.
- **`onTick`**: al evento `InterpolatorEvent.TICK` para aplicar las interpolaciones cada vez que `_interpolator`, dispare el evento.
- **`onStart`**: al evento `InterpolatorEvent.START`, que se ejecutará cada vez que la animación llegue al milisegundo 0.
- **`onComplete`**: al evento `InterpolatorEvent.COMPLETE`, que se ejecutará cuando la animación llegue al último milisegundo.

Así agregaremos los escuchas:

```
/Creamos la interpolación.
      _interpolator = new Interpolator(1000,
                                       Interpolator.BOUNCE_FORMULA,
                                       Interpolator.EASE_OUT,
                                       Interpolator.LOOP_HOLD);

      _interpolator.setTweenProperties("x", 0, 750);
      _interpolator.setTweenProperties("rotationZ", 0, 360);

      //Agregamos los escuchas.
      interpolator.addEventListener(InterpolatorEvent.TICK, on-
Tick);
      interpolator.addEventListener(InterpolatorEvent.START, on-
Start);
      interpolator.addEventListener(InterpolatorEvent.COMPLETE, on-
Complete);
```

Y así se verían los métodos escuchas:

```
private function onTick(event:InterpolatorEvent):void
{
  _rectangle.x = event.getTweenByName("x");
  _rectangle.rotationZ = event.getTweenByName("rotationZ");

  var message:String = "";
  message += "\nMilisegundo actual: " + event.currentMillisecond +
"\n";
  message += "Milisegundos totales: " + event.totalMilliseconds+
"\n";
  message += "Interpolación: " + event.interpolation+ "\n";
  message += "Momento relativo: " + event.milisecondRatio+ "\n";
  message += "---------------\n";
  trace(message);
}

private function onStart(event:InterpolatorEvent):void
```

```
{
  trace("\nCOMENZÓ LA ANIMACIÓN!!!\n");
}

private function onComplete(event:InterpolatorEvent):void
{
  trace("\nANIMACIÓN COMPLETA!!!\n");
}
```

Probamos y vemos que la animación se sigue ejecutando como antes y se verá la salida.

```
Milisegundo actual: 0
Milisegundos totales: 5000
Interpolación: 0
Momento relativo: 0
---------------


COMENZÓ LA ANIMACIÓN!!!


Milisegundo actual: 65
Milisegundos totales: 5000
Interpolación: 0.0012780625
Momento relativo: 0.013
---------------

Milisegundo actual: 190
Milisegundos totales: 5000
Interpolación: 0.01092025
Momento relativo: 0.038
---------------
```

Luego, al llegar al final de la animación, veremos:

```
Milisegundo actual: 4999
Milisegundos totales: 5000
Interpolación: 0.9998628025
Momento relativo: 0.9998
---------------


Milisegundo actual: 5000
Milisegundos totales: 5000
Interpolación: 1
Momento relativo: 1
---------------


ANIMACIÓN COMPLETA!!!
```

Podemos hacer distintas pruebas cambiando los parámetros. Si por ejemplo, cambiamos el parámetro `Interpolator.LOOP_HOLD` por `Interpolator.LOOP_REPEAT`, la animación se volverá a repetir indefinidamente.

EFECTO TIME BULLET.

Un efecto que particularmente me gusta mucho ver en el cine y los videojuegos es el *Time bullet (*tiempo de bala). Es un efecto que consiste en ralentizar extremadamente el tiempo de tal manera que podemos ver todos los detalles de los movimientos muy rápidos, en un momento determinado. Este efecto se utilizó en películas como *Matrix*, y se usa también en la película *300*.

Dado que nuestra clase nos ofrece flexibilidad para cambiar los parámetros en tiempo real, agregaremos algunas líneas de código para generar este efecto. Vamos a la clase `MainInterpolation` al método: `onTick`. Allí podremos cambiar las propiedades de la animación mientras está sucediendo.

Lo que determina la velocidad en la que veremos la animación es su duración. Aumentando la duración de la animación en cada `millisecondRatio`, es decir, en la misma posición relativa de la animación se efectuará la misma transformación. Modificando la duración, lo que varía es en qué milisegundo se verá la animación.

Este es un tema interesante, porque independientemente de la duración, tenemos valores relativos que no cambiarán. Así, si el valor de `millisecondRatio` es 0.5, sabemos que estamos en la mitad de la animación. Entonces, lo que hare-

mos será que la animación se desarrolle a la velocidad en la que fue definida al comienzo (5000 milisegundos), y en distintos momentos de la animación, variaremos la duración. Para saber en qué fracción de la animación nos encontramos utilizaremos la propiedad `millisecondRatio`.

A partir del `millisecondRatio` 0.59 cambiaremos, además de la duración, la fórmula y la propiedad final de `rotationZ`. Cuando se complete la animación en el manejador de evento `onComplete`, cambiaremos las propiedades a su valor inicial. Así, al comenzar, la animación se ve siempre igual. Por supuesto que las combinaciones y variaciones que podemos hacer son infinitas.

Y aquí, el código para que nuestra animación se vea como un *time bullet*:

Clase `MainInterpolation`:

```
package
{
  import flash.events.Event;
  import flash.display.Sprite;

  import com.alfaomegaeditor.utils.interpolation.Interpolator;
  import com.alfaomegaeditor.events.InterpolatorEvent;

  public class MainInterpolation extends Sprite
  {
    private var _interpolator:Interpolator;
    private var _rectangle:Sprite;
    private var _initialXPosition:Number;
    private var _finalXPosition:Number;

    public function MainInterpolation():void
    {
      initialize();
    }

    private function initialize():void
    {
      //Creamos un rectángulo.
```

```
      _rectangle = new Sprite();
      _rectangle.graphics.beginFill(0xFF0000);
      _rectangle.graphics.drawRect(-10, -50, 20, 100);

      //Definimos la posición X inicial y final.
      _initialXPosition = 100;
      _finalXPosition = 500;

      //Ubicamos al rectángulo.
      _rectangle.x = _initialXPosition;
      _rectangle.y = 150;

      //Lo agregamos al escenario.
      addChild(_rectangle);

      //Creamos la interpolación.
      _interpolator = new Interpolator(5000,
                                       Interpolator.BOUNCE_FORMULA,
                                       Interpolator.EASE_OUT,
                                       Interpolator.LOOP_REPEAT);

     //Agregamos los valores a calcular.
     _interpolator.setTweenProperties("x", 0, 750);
     _interpolator.setTweenProperties("rotationZ", 0, 360);

     //Agregamos los escuchas.
      interpolator.addEventListener(InterpolatorEvent.TICK, on-
Tick);
      interpolator.addEventListener(InterpolatorEvent.START, on-
Start);
      interpolator.addEventListener(InterpolatorEvent.COMPLETE, on-
Complete);

      //Agregamos un escucha al Event.ENTER_FRAME
      this.addEventListener(Event.ENTER_FRAME, onEnterFrame);
    }

    private function onEnterFrame(event:Event):void
```

```
    {
      _interpolator.nextTick();
    }

    private function onTick(event:InterpolatorEvent):void
    {
      _rectangle.x = event.getTweenByName("x");
      _rectangle.rotationZ = event.getTweenByName("rotationZ");

      var message:String = "";
      message += "\nMilisegundo actual: " +
event.currentMillisecond + "\n";
      message += "Milisegundos totales: " +
event.totalMilliseconds+ "\n";
      message += "Interpolación: " + event.interpolation+ "\n";
      message += event.milisecondRatio+ "\n";
      message += "---------------\n";
      trace(message);

      //Configuraciones para el efecto Time bullet.
      if (_interpolator.millisecondRatio < .26)
      {
        _interpolator.totalMilliseconds = 5000;
        _interpolator.setFormula(Interpolator.BOUNCE_FORMULA,
                                 Interpolator.EASE_OUT);
        _interpolator.setTweenProperties("x", 0, 750);
        _interpolator.setTweenProperties("rotationZ", 0, 360);
      }

      if (_interpolator.millisecondRatio > .26)
      {
        _interpolator.totalMilliseconds = 50000
      }

      if (_interpolator.millisecondRatio > .5)
      {
        _interpolator.totalMilliseconds = 5000
```

```
      }

      if (_interpolator.millisecondRatio > .59)
      {
        _interpolator.totalMilliseconds = 3500
        _interpolator.setEndProperty("rotationZ", -90)
        _interpolator.setFormula(Interpolator.ELASTIC_FORMULA,
                                 Interpolator.EASE_OUT);
      }
    }

    private function onStart(event:InterpolatorEvent):void
    {
      trace("\nCOMENZÓ LA ANIMACIÓN!!!\n");
    }

    private function onComplete(event:InterpolatorEvent):void
    {
      trace("\nTERMINÓ LA ANIMACIÓN!!!\n");
    }
  }
}
```

Manejo de bucles de tiempo sin el Event.ENTER_FRAME

Como habíamos comentado, existen bucles de tiempo que podemos utilizar en ActionScript 3.0 que no están directamente vinculados con la representación. Estos bucles de tiempo se ejecutan independientemente del que se captura por `Event.ENTER_FRAME`, y no son útiles para realizar animaciones, por las razones ya mencionadas. Al no estar directamente vinculado a la representación de objetos visuales, si lo usamos para ese fin, puede suceder que un proceso se ejecute, intentando mostrar algún efecto y consumiendo recursos innecesariamente.

Sin embargo, estos tipos de bucles se pueden utilizar para hacer depuración de código o para otros procesos que no estén directamente vinculados con la repre-

sentación visual. Es común utilizarlos para hacer chequeos de datos cada cierta cantidad de tiempo, realizar detecciones o capturas de algún tipo particular de evento, disparar actualizaciones de página o pantalla.

Nosotros cubriremos aquellos bucles de tiempo que se realizan por código. Debemos, entonces, tener en cuenta que utilizaremos este código para procesos repetitivos independientes de la representación visual.

Forzando la representación de pantalla mediante updateAfterEvent

Para como inciden estas prácticas, efectuaremos algunos ajustes a nuestra clase `MainInterpolator`. Haremos las siguientes pruebas:

1. Configuraremos a 1 fotograma por segundo la propiedad `frameRate` del `Stage`.
2. Configuraremos a 1 fotograma por segundo la propiedad `frameRate` del `Stage`, y reemplazaremos el bucle con `ENTER_FRAME` por el `Timer`.
3. Configuraremos a 1 fotograma por segundo la propiedad `frameRate` del `Stage`, y reemplazaremos el bucle con `ENTER_FRAME` por el `Timer`, llamaremos al método: `updateAfterEvent`.

Prueba 1

En la clase `MainInterpolation`, en el método `initialize`, en la primera línea, configuraremos los fotogramas por segundo a 1.

```
private function initialize():void
{
  //Configuramos los fotogramas por segundo a: 1.
  stage.frameRate = 1;
```

Al probar la película no veremos la animación más lenta. Es decir, no veremos todos los fotogramas de la animación: veremos saltos (el salto de frame), lo cual nos indica que indiscutiblemente el *drop frames* está funcionando.

Prueba 2

En la clase `MainInterpolation` reemplazamos el `ENTER_FRAME` por el `Timer`.

Veremos resaltado en el código, las líneas agregadas.

```
package
{
  import adobe.utils.CustomActions;
  import flash.events.Event;
  import flash.events.TimerEvent;

  import flash.display.Sprite;
  import flash.utils.Timer;

  import com.alfaomegaeditor.utils.interpolation.Interpolator;
  import com.alfaomegaeditor.events.InterpolatorEvent;

  public class MainInterpolation extends Sprite
  {
    private var _interpolator:Interpolator;
    private var _rectangle:Sprite;
    private var _initialXPosition:Number;
    private var _finalXPosition:Number;
    private var _timer:Timer;

    public function MainInterpolation():void
    {
      initialize();
    }

    private function initialize():void
    {
      //Configuramos los fotogramas por segundo a: 1.
      stage.frameRate = 1;

      //Creamos un rectángulo.
      _rectangle = new Sprite();
```

```
      _rectangle.graphics.beginFill(0xFF0000);
      _rectangle.graphics.drawRect(-10, -50, 20, 100);

      //Definimos la posición X inicial y final.
      _initialXPosition = 100;
      _finalXPosition = 500;

      //Ubicamos al rectángulo.
      _rectangle.x = _initialXPosition;
      _rectangle.y = 150;

      //Lo agregamos al escenario.
      addChild(_rectangle);

      //Creamos la interpolación.
      _interpolator = new Interpolator(5000,
                                       Interpolator.BOUNCE_FORMULA,
                                       Interpolator.EASE_OUT,
                                       Interpolator.LOOP_REPEAT);

     //Agregamos los valores a calcular.
     _interpolator.setTweenProperties("x", 0, 750);
     _interpolator.setTweenProperties("rotationZ", 0, 360);

     //Agregamos los escuchas.
      interpolator.addEventListener(InterpolatorEvent.TICK, on-
Tick);
      interpolator.addEventListener(InterpolatorEvent.START, on-
Start);
      interpolator.addEventListener(InterpolatorEvent.COMPLETE, on-
Complete);

      //Definimos los fotogramas por segundo que queremos emular.
      var framesPerSecond:uint = 60;

      //Calculamos de cuánto será el intervalo al que tiene que ser
llamado
      //el escucha del Timer.
```

```
      var updatesPerSecond:Number = 1000 / framesPerSecond;

      //Creamos el Timer. Y le asignamos el intervalo obtenido y le
pasamos
      //como parámetro de repetición 0 que indica que se repro-
ducirá
      //indefinidamente.
      _timer = new Timer(updatesPerSecond, 0);

      //Asignamos como escucha de _timer a onEnterFrame.
      _timer.addEventListener(TimerEvent.TIMER, onEnterFrame);

      //Iniciamos el temporizador.
      _timer.start();
    }

    private function onEnterFrame(event:TimerEvent):void
    {
      _interpolator.nextTick();
    }

    private function onTick(event:InterpolatorEvent):void
    {
      _rectangle.x = event.getTweenByName("x");
      _rectangle.rotationZ = event.getTweenByName("rotationZ");

      //Configuraciones para el efecto Time bullet.
      if (_interpolator.millisecondRatio < .26)
      {
        _interpolator.totalMilliseconds = 5000;
        _interpolator.setFormula(Interpolator.BOUNCE_FORMULA,
                                 Interpolator.EASE_OUT);
        _interpolator.setTweenProperties("x", 0, 750);
        _interpolator.setTweenProperties("rotationZ", 0, 360);
      }

      if (_interpolator.millisecondRatio > .26)
```

```
      {
        _interpolator.totalMilliseconds = 50000
      }

      if (_interpolator.millisecondRatio > .5)
      {
        _interpolator.totalMilliseconds = 5000
      }

      if (_interpolator.millisecondRatio > .59)
      {
        _interpolator.totalMilliseconds = 3500
        _interpolator.setEndProperty("rotationZ", -90)
        _interpolator.setFormula(Interpolator.ELASTIC_FORMULA,
                                 Interpolator.EASE_OUT);
      }
    }

    private function onStart(event:InterpolatorEvent):void
    {
      trace("\nCOMENZÓ LA ANIMACIÓN!!!\n");
    }

    private function onComplete(event:InterpolatorEvent):void
    {
      trace("\nTERMINÓ LA ANIMACIÓN!!!\n");
    }
  }
}
```

Al probar la película, veremos exactamente el mismo funcionamiento, solamente que esta vez utilizando la clase `Timer`. Lo cual es correcto y es lo que estábamos esperando. Sin embargo, en este caso, se estarán ejecutando en cada segundo 59 procesos que serán calculados, pero no representados.

Prueba 3

Y aquí nuestra prueba final. El manejador de evento `onTick`, que es el que recibe el evento `TimerEvent`, podrá ejecutar a través del objeto `TimerEvent` el método `updateAfterFrame`.

Vemos las modificaciones en negrita:

```
private function onEnterFrame(event:TimerEvent):void
{
  _interpolator.nextTick();
  event.updateAfterEvent();
}
```

Y ahora, vemos que a pesar de que el `frameRate` es de 1 fotograma por segundo, cada vez que se ejecuta `onEnterFrame`, tiene lugar el procesamiento del evento; y el llamado **`event.updateAfterEvent()`** provoca que se represente la pantalla a pesar de que el intervalo de 1 segundo configurado no se haya cumplido.

Recordemos que el llamado **`event.updateAfterEvent()`** no interrumpe los procesos o los bloques de códigos que se estén ejecutando al momento de ser llamado. El reproductor tratará de ejecutar el llamado al escucha en el intervalo del valor configurado, pero no hay garantía de que esto suceda de manera precisa.

Utilizando el momento oportuno para la representación con stage.invalidate y Event.RENDER

Existe otra técnica para diferir los cálculos y procesos de dibujado hasta un momento inmediatamente anterior a que se produzca la representación y dibujo. Consiste en utilizar el evento `Event.RENDER` en combinación con el llamado `stage.invalidate`.

Para que el evento `Event.RENDER` sea disparado antes de la representación del fotograma, previamente se tuvo que haber llamado desde cualquier objeto que tenga acceso al objeto `Stage`, al método `stage.invalidate`. De lo contrario, este evento no se disparará nunca.

Habíamos visto cómo se produce la reproducción de fotogramas. Ahora bien, veremos remarcado un punto del cual sacaremos ventaja para aplicar una técnica que analizaremos luego de ésta explicación:

1 Ejecución de código: el reproductor ejecuta el código de un fotograma.

2 Espera: espera la fracción de tiempo de acuerdo con la configuración de fotogramas por segundo.
   a) Si se disparan eventos, permite que se ejecuten los escuchas.

3 Representación: el reproductor determina si se debe realizar alguna representación visual de acuerdo con lo sucedido en el paso 1. En estos casos, realizará un cambio de la representación:
   a) Si hubo cambios generados desde la línea de tiempo.
   b) Si hubo cambios generados por código.

Cómo funciona el stage.invalidate con el evento: Event.RENDER

El reproductor antes de realizar una representación identifica las áreas del `Stage` que han cambiado. Estas áreas las identifica en forma de rectángulos. Cuando hubo cambio con respecto al fotograma anterior, las áreas que han cambiado son marcadas con un rectángulo para ser dibujadas nuevamente. Si queremos ver en acción al reproductor marcando estas áreas, debemos hacer clic con el botón derecho del ratón sobre el .SWF y seleccionar: “Mostrar regiones de redibujo”. Veremos distintos cuadrados rojos dependiendo del caso en donde el reproductor aplicará cambios. De esta manera, contamos con una mejor *performance* al momento de reproducir.

Los rectángulos a actualizar se los conoce en video y videojuegos como rectángulos sucios.

Lo que haremos será indicar que hace falta una actualización, y que el reproductor avise un momento antes de hacer una representación para contar con la última oportunidad previa a la representación del fotograma de hacer cambios.

Repasemos los pasos teniendo en cuenta el tiempo. Supongamos que los fotogramas por segundo son 50, o sea, que el reproductor intentará ejecutar el `Event.ENTER_FRAME` cada 20 milisegundos. Es decir, que 20 milisegundos representarían 1 fotograma.

Milisegundo actual: 0

Paso actual: Ejecución de código

- Supongamos que tenemos un código muy complejo de procesar que demande 40 milisegundos.

Milisegundo actual: 40 (ya tenemos un retraso de 2 fotogramas).

Paso actual: Espera

- Se permiten disparar y procesar eventos.
- Si existe un llamado a `updateAfterEvent`, se pasa al siguiente paso.
- De lo contrario, se esperan 20 milisegundos para el segundo paso.

Milisegundo actual: 60

Paso actual: Representación

- Se realizará la representación de lo que sucedió en el paso 1. Es decir, que en el milisegundo 60, veremos la representación de lo sucedido en el fotograma 0. Pensándolo en función de fotogramas, en el fotograma actual real (que es el número 3) veríamos la representación del fotograma 0 o la representación del último estado de un objeto.

Sin embargo, mediante una técnica que mostraremos, es posible sacar ventaja del procesamiento de eventos que se da en el paso 2 mientras el reproductor espera que pasen los milisegundos configurados.

Milisegundo actual: 0

Paso actual: Ejecución de código

- Supongamos que tenemos un código muy complejo de procesar que demande 40 milisegundos.
- Se dispara el pedido de ejecutar el evento `Event.RENDER` mediante: `stage.invalidate()`.

Lo que estaríamos pidiéndole al reproductor es: “Verifica, por favor, si hay algún cambio en la pantalla que representar. Si vas a hacer una representación, antes de

realizarla, envía un evento notificando que estás a punto de efectuarla". Estamos pidiendo que se ejecute un evento para "diferir" una representación. Y también por medio de `invalidate`, le estamos diciendo al reproductor: "La representación actual ya no es válida, debes hacer otra para mantener consistencia con lo que ha sucedido con el código".

Milisegundo actual: 40 (ya tenemos un retraso de 2 fotogramas).

Paso actual: Espera

- Se permiten disparar y procesar eventos.
- Si existe un llamado a `updateAfterEvent`, se pasa al siguiente paso.
- De lo contrario, se esperan de 20 milisegundos para el segundo paso.

Milisegundo actual: 60

Paso actual: Espera

- El reproductor ejecuta el evento Event.RENDER justo antes de hacer la representación, es decir, al cumplirse los 20 milisegundos, tal cual le fue solicitado en el paso 1 en forma diferida. El escucha del evento puede dibujar nuevamente o cambiar la representación de un gráfico, y tener en cuenta el milisegundo actual para calcular el fotograma, en caso de estar aplicando la técnica de *drop frames* o de representar objetos visuales basados en su estado.

Milisegundo actual: 62

Paso actual: Representación.

- Supongamos que los escuchas del evento Event.RENDER se hayan demorado 2 milisegundos en procesar el código al momento de la representación, estaríamos en el milisegundo 62. Se realizará la representación de lo que sucedió en el paso 2. Es decir, que en el milisegundo 62, veremos la representación de lo sucedido en el fotograma 2. Pensándolo en función de fotogramas, en el fotograma actual real que es el número 2,03, veríamos la representación del fotograma 2.

Implementando stage.invalidate con el evento: Event.RENDER

La aplicación de esta técnica no es usual en aplicaciones Web debido a que este tipo de técnicas se utiliza para ganar *performance* y ahorrar recursos, especialmente, en juegos donde cualquier detrimento en la representación visual impacta direc-

tamente en la calidad del juego, en como se "siente". Y además que en un juego existen varios frentes que cuidar: manejo de datos, intercambio de datos externos, animaciones y representaciones, aplicación de lógica, inteligencia artificial, y quizás todo esto esté sucediendo en un mismo momento.

Se siente inmediatamente con la caída de fotogramas, y a veces da por resultado el congelamiento de la pantalla o incluso que la aplicación deje de responder. Y lo que es aún peor, en todo tipo de juegos, incluso en juegos para consola triple A, existe una prueba que consiste en entrar en una pantalla del juego y quedarse allí sin hacer absolutamente nada. Al cabo de unos segundos o minutos la aplicación deja de responder, la mayoría de las veces por una mala administración de los procesos, es decir, procesos que consumen memoria y procesador en momentos en los que no deberían ocurrir.

Veamos un ejemplo sencillo, pero representativo. Tenemos un objeto `VisualDisplay` al cual le queremos agregar un rectángulo de selección. Este rectángulo rodeará a la imagen en cualquiera de sus estados. Este rectángulo de selección deberá en todo momento estar posicionado sobre la imagen demarcando sus límites. Cada vez que el cuadrado cambie su alto, ancho, rotación y posición, el rectángulo de selección deberá adaptarse automáticamente.

Para mantenerlo sencillo y solo a modo de ejemplo, utilizaremos una clase para el `VisualDisplay` y otra para el manejo de selección: `SelectionObject`.

Estas dos clases vivirán en la clase: `MainRenderTest`.

- Al cambiar cualquiera de las propiedades del **`VisualDisplay`** cambiará la selección.

La clase `MainRenderTest` le indicará a la clase `SelectionObject` cuál es el rectángulo actual de la imagen a "rodear". Es importante aclarar que para este caso particular no es necesario aplicar esta estrategia, es simplemente un ejemplo sencillo para poder captar el concepto y aplicarlo en los casos donde es primordial cada "gota" de *performance* que podamos ganar.

La manera menos ineficaz de implementar este ejemplo en la clase `MainRenderTest` sería:

```
private function onTick(event:Event):void
{
  _lastInterval = getTimer();
  _interpolator.nextTick();
   visualDisplay.x =
_interpolator.calculateTween(_initialXPosition, 500);
  _visualDisplay.rotation = _interpolator.calculateTween(0, 360);
```

```
    _visualDisplay.draw();
    _rectangle.draw(_visualDisplay.getBounds(this));
}
```

Debemos subrayar que no tiene un `updateAfterFrame`, en caso de ser llamado desde un `Timer` podría pasar lo siguiente:

Durante la espera a la representación del próximo fotograma se recibe el evento:

1. `MainRenderTest` cambia la posición de `VisualDisplay`.
2. `MainRenderTest` cambia la rotación de `VisualDisplay`.
3. `MainRenderTest` manda a dibujar a `VisualDisplay`.
4. `MainRenderTest` manda a dibujar a `SelectionObject`.

5. `MainRenderTest` cambia la posición de `VisualDisplay`.
6. `MainRenderTest` cambia la rotación de `VisualDisplay`.
7. `MainRenderTest` manda a dibujar a `VisualDisplay`.
8. `MainRenderTest` manda a dibujar a `SelectionObject`.
   ...Aquí siguen llegando llamados a `onTick` hasta llegar al paso 39...

39. `MainRenderTest` manda a dibujar a `VisualDisplay`.
40. `MainRenderTest` manda a dibujar a `SelectionObject`.

Al representar el fotograma:

41. Se ven representados `VisualDisplay y SelectionObject` pero de lo dibujado en el paso 39 y 40.

El conteo de pasos hasta 40 es un ejemplo arbitrario, podrían ser también menos. Lo importante es que solo se representará lo que sucedió en el último paso. Por lo cual, todos los llamados previos han sido un desperdicio. Si a esto le sumáramos que los procesos de dibujado son complejos, podríamos tener serios inconvenientes de *performance*, ejecutando procesos demandantes de los cuales solo se verá representado el último.

Aplicando la representación diferida

Ahora aplicaremos la representación en forma diferida. Como ya nos habremos dado cuenta, diferiremos la representación de los estados hasta un momento previo a la representación. Este momento solo lo conoce el reproductor y no tenemos manera de calcularlo. Por eso, necesitamos que nos avise con el evento `Event.RENDER`, que está por hacer la representación del fotograma.

Ahora emplearemos otra estrategia. En vez de llamar al método que dibuja cada vez, solo modificaremos el estado de `VisualDisplay` y llamaremos a `stage.invalidate`(). Previo a todo esto, debemos agregar un escucha al evento: `Event.RENDER`.

Y ahora veamos cómo aplicar la tánica de la representación diferida `MainRenderTest` en forma general:

En el método inicializador, suscribimos la clase al evento Event.RENDER.

```
stage.addEventListener(Event.RENDER);
```

Luego, en el `onTick` solo cambiamos el estado de `_visualDisplay` y disparamos el `stage.invalidate()`. No estamos cargando a este manejador de eventos con más proceso del que se está representando como antes.

```
private function onTick(event:Event):void
{
  _lastInterval = getTimer();
  _interpolator.nextTick();
   visualDisplay.x =
_interpolator.calculateTween(_initialXPosition, 500);
  _visualDisplay.rotation = _interpolator.calculateTween(0, 360);

  stage.invalidate();
}
```

Luego, en el manejador de evento del `Event.RENDER`, tendremos lo siguiente:

```
private function onRender(event:Event):void
{
  _visualDisplay.draw();
  _rectangle.draw(_visualDisplay.getBounds(this));
}
```

Y esto sucedería durante la reproducción:

Espera a la representación del próximo fotograma:

1. `MainRenderTest` cambia la posición de `VisualDisplay`.
2. `MainRenderTest` cambia la rotación de `VisualDisplay`.
3. `MainRenderTest` ejecuta `stage.invalidate()`.

4. `MainRenderTest` cambia la posición de `VisualDisplay`.
5. `MainRenderTest` cambia la rotación de `VisualDisplay`.
6. `MainRenderTest` ejecuta `stage.invalidate()`.

...Aquí siguen llegando llamados a `onTick` hasta llegar al paso 37...

37. `MainRenderTest` cambia la posición de `VisualDisplay`.
38. `MainRenderTest` cambia la rotación de `VisualDisplay`.
39. `MainRenderTest` ejecuta `stage.invalidate()`.

Justo antes de representar el fotograma, el reproductor recibió el pedido de disparar el evento Event.Render, y lo dispara.

40. `MainRenderTest` captura el evento y recién en ese momento manda dibujar a `VisualDisplay` y a `SelectionObject`.

Al representar el fotograma:

39. Se ven representados `VisualDisplay` y `SelectionObject` pero de lo dibujado en el paso 38 y 39.

Si además tuviéramos por caso que el proceso de dibujo fuera demandante de memoria y procesos estaríamos reduciendo considerablemente las exigencias y aumentando la *performance*, ya que solo llamaremos al método para dibujar, justo antes de la representación.

Y las clases para representar este ejemplo son:

Comenzamos con la clase `VisualDisplay`, que extiende a `Sprite`. Algo a tener en cuenta es que hemos sobrescrito las propiedades con las que trabajaremos en este ejemplo, y son: x, y y rotation. Esto provocará que los cambios no se apliquen hasta que el método `draw` sea llamado. Cuando éste sea llamado, aplicará los cambios del estado por medio de `super`.

Clase: `VisualDisplay`:

```
package
{
  import flash.display.Sprite;

  public class VisualDisplay extends Sprite
  {
    /**
     * Esta propiedad indica si hubo cambios en el estado.
     * Sino hubo cambios, no se dibuja el estado para ahorrar re-
cursos.
     */
    private var _stateHasChanged:Boolean = false;
    private var _sprite:Sprite;
    private var _x:Number = 0;
    private var _y:Number = 0;
    private var _rotation:Number = 0;

    /**
     * Constructor
     */
    public function VisualDisplay():void
    {
      initialize();
    }

    /**
     * Inicializa el objeto.
     */
    private function initialize():void
```

```
{
  //Creamos un rectángulo.
  _sprite = new Sprite();
  _sprite.graphics.beginFill(0x00CC55);
  _sprite.graphics.drawRect(-10, -50, 20, 100);
  addChild(_sprite)
}

/**
 * Getters y setter de posición.
 */
override public function get x():Number
{
  return _x;
}
override public function set x(value:Number):void
{
  if (isNaN(value) || value == _x)
  {
    return;
  }
  _stateHasChanged = true;
  _x = value;
}
override public function get y():Number
{
  return _y;
}
override public function set y(value:Number):void
{
  if (isNaN(value) || value == _y)
  {
    return;
  }
  _y = value;
  _stateHasChanged = true;
```

```
    }

    /**
     * Getters y setter de rotación.
     */
    override public function get rotation():Number
    {
      return _rotation;
    }

    override public function set rotation(value:Number):void
    {
      if (isNaN(value) || value == _rotation)
      {
        return;
      }
      _rotation = value;
      _stateHasChanged = true;
    }

    /**
     * Dibuja en función del estado si es que hubo cambios.
     */
    public function draw():void
    {
      //Sino hubo cambios en el estado desde el último dibujo.
      //Detenemos el método para ahorrar recursos.
      if (!_stateHasChanged)
      {
        return;
      }

      super.x = _x;
      super.y = _y;
      super.rotation = _rotation;
```

```
      //Luego de dibujar volvemos la propiedad a:  stateHasChanged
a false.
      //Si no llegaran a producirse cambios, esta propiedad perman-
ecería como
      //false y al ser llamado este método, no se volvería a dibu-
jar.
      _stateHasChanged = false;
    }
  }
}
```

Clase `SelectorObject`:

```
package
{
  import flash.display.Sprite;
  import flash.display.LineScaleMode;
  import flash.geom.Rectangle;

  public class SelectionObject extends Sprite
  {
    private const PADDING:Number = 10;

    //Esta propiedad guarda los datos del rectángulo a representar.
    private var _rectangleBounds:Rectangle;

    //A través de este objeto visual representaremos el rectángulo
de selección.
    private var _sprite:Sprite;
    /**
     * Constructor
     */
    public function SelectionObject():void
    {
      initialize();
    }
```

```
    /**
     * Inicializa el objeto.
     */
    private function initialize():void
    {
      //Creamos un rectángulo.
      _sprite = new Sprite();
      _rectangleBounds = new Rectangle();
      addChild(_sprite)
    }
    /**
     * Dibuja en función del estado si es que hubo cambios.
     */
    public function draw(rectangle:Rectangle):void
    {
      //Si el rectángulo es nulo o es el mismo, no lo dibujamos.
      if (rectangle == null || _rectangleBounds.equals(rectangle))
      {
        return;
      }
      //Guardamos el último rectángulo dibujado.
      _rectangleBounds = rectangle;

      _sprite.graphics.clear();
       sprite.graphics.lineStyle(.6, 0xFF0000, 1, true, LineScale-
Mode.NONE);
      _sprite.graphics.drawRect(_rectangleBounds.x - PADDING,
                                _rectangleBounds.y - PADDING,
                                 rectangleBounds.width + PADDING *
2,
                                 rectangleBounds.height + PADDING *
2);
    }
  }
}
```

Clase `MainTimerTest`:

```
package
{
  import flash.events.TimerEvent;
  import flash.events.Event;
  import flash.utils.Timer;
  import flash.display.Sprite;

  import com.alfaomegaeditor.utils.interpolation.Interpolator;
  public class MainRenderTest extends Sprite
  {
    private var _timer:Timer;
    private var _interpolator:Interpolator;
    private var _selectionObject:SelectionObject;
    private var _visualDisplay:VisualDisplay;
    private var _initialXPosition:Number;
    private var _finalXPosition:Number;

    public function MainRenderTest():void
    {
      initialize();
    }

    private function initialize():void
    {
      //Creamos el rectángulo selector.
      _selectionObject = new SelectionObject();

      //Creamos el objeto a desplegar.
      _visualDisplay = new VisualDisplay();

      //Definimos la posición X inicial y final.
      _initialXPosition = 100;
      _finalXPosition = 500;

      //Ubicamos al rectángulo.
      _visualDisplay.x = _initialXPosition;
```

```
      _visualDisplay.y = 150

      //Agregamos ambos objetos visuales al escenario.
      addChild(_selectionObject);
      addChild(_visualDisplay);

      //Creamos la interpolación.
      _interpolator = new Interpolator(5000,
                                       Interpolator.BOUNCE_FORMULA,
                                       Interpolator.EASE_OUT,
                                       Interpolator.LOOP_REPETE,
                                       0,
                                       750);

      //Ponemos 1 para forzar a que el reproductor ejecute el
TimerEvent la
      //mayor cantidad de veces posibles. Y pasamos 0 para que se
repita
      //indefinidamente.
      _timer = new Timer(1, 0);

      //Suscribmos al Event.RENDER.
      this.addEventListener(Event.RENDER, onRender);
      //Suscribimos a los eventos de Timer.
      _timer.addEventListener(TimerEvent.TIMER, onTick);

      //Iniciamos el Timer.
      _timer.start();
    }

    private function onTick(event:TimerEvent):void
    {
      //Le pedimos al interpolador que avance su próximo tick.
      _interpolator.nextTick();

      //Solo cambiamos el estado de  visualDisplay con los datos
obtenidos,
      //del interpolador.
```

```
       visualDisplay.x =
_interpolator.calculateTween(_initialXPosition, 500);
       visualDisplay.rotation =  interpolator.calculateTween(0,
360);


      //Llamamos al método que ocasionará que se dispare
Event.RENDER antes
      //de representarse el fotograma.
      stage.invalidate();
    }


    private function onRender(event:Event):void
    {
      //Cuando se dispara este evento significa que a continuación
se
      //representará el fotograma. Entonces mandamos a dibujar a
cada uno
      //de los objetos su estado.
      _visualDisplay.draw();
      _selectionObject.draw(_visualDisplay.getBounds(this));
    }
  }
}
```