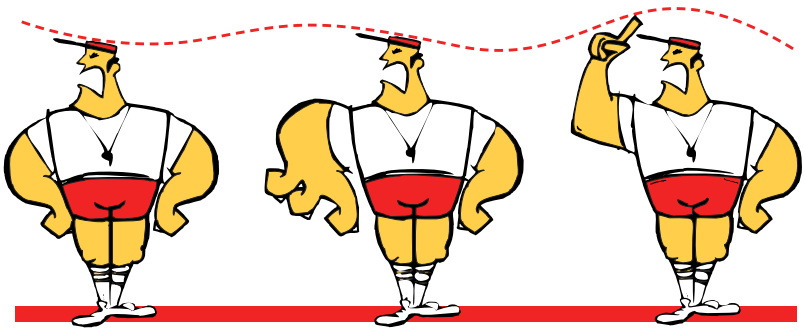
Мастера FLASH

Роберт Пеннер

ПРОГРАММИРОВАНИЕ ВО FLASH MX



СИМВОЛ®

Robert Penner's Programming Macromedia Flash™ MX

Robert Penner

McGraw-Hill/Osborne

Мастера FLASH

Программирование во Flash MX

Роберт Пеннер

Санкт-Петербург — Москва
2005

Серия «Мастера FLASH»

Роберт Пеннер

Программирование во Flash MX

Перевод С. Иноземцева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*М. Антипин*
Редактор	*А. Лосев*
Художник	*В. Гренда*
Корректор	*С. Доничкина*
Верстка	*Н. Гриценко*

Пеннер Р.
Программирование во Flash MX. – Пер. с англ. – СПб: Символ-Плюс, 2005. – 432 с., ил.
ISBN 5-93286-072-3

Роберт Пеннер – известный новатор, пишущий на языке ActionScript, – делится своим глубоким пониманием сути программирования в Macromedia Flash. Он обладает талантом объединять сложные понятия математики и физики с чистой эстетикой графики Macromedia Flash. Автор проведет вас от простейших приемов проектирования и кодирования движения до способов разработки профессионального объектно-ориентированного кода, позволяющего создавать интерактивность, цвет, звук и движение. Эта книга раздвинет границы вашего воображения. Оригинальные идеи автора, создавшего танцующий фрактал, имитацию снежной бури и северного сияния, подвигнут вас к реализации интересных динамических проектов и анимации в Macromedia Flash на мощном и гибком языке ActionScript.

Вот мнение Колина Мука, автора бестселлеров по ActionScript: «Этой книге суждено стать классической работой по программированию графики в Macromedia Flash. Flash-разработчики получили фундаментальные и практичные образцы объектно-ориентированного программирования, о которых они мечтали много лет. Как новички, так и эксперты в области ООП высоко оценят стиль Пеннера.»

ISBN 5-93286-072-3
ISBN 0-07-222356-1 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.
Подписано в печать 06.06.2005. Формат 70x100 1/16 . Печать офсетная.
Объем 27 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Вильяму Рональду Пеннеру и Вильме Джун Пеннер, которые заставляли меня «погулять и подышать свежим воздухом».

Об авторе

Роберт живет в Канаде в г. Ванкувер. Он свободный Flash-разработчик, консультант и лектор, чьи интересы сосредоточены в области математики и объектно-ориентированного проектирования. Модератор и один из основных авторов сайта Ultrashock.com, Роберт известен во всем мире своей технической изобретательностью и органичным стилем. Он предложил такие новшества, как интеграция кнопки возврата во Flash и математические уравнения для реализации ускорения. Роберт не ограничивается модерированием нескольких форумов на Ultrashock и горячими спорами об ООП с членами сообщества Flashcoders на сайте Chattyfig. Сотрудничество с Axis-media.com принесло ему две номинации на фестивалях Flash-фильмов. Законченные проекты и эксперименты Роберта можно увидеть на сайте *http://www.robertpenner.com*.

Оглавление

Введение

Больше года тому назад Джим Шахтерль (Jim Schachterle) предложил мне написать книгу о Flash для издательства Osborne/McGraw-Hill. У него был весьма необычный замысел: проект должен быть «на пересечении программирования и дизайна». Иными словами, речь в книге должна идти о создании визуальных элементов с помощью кода. К этому моменту я уже некоторое время занимался чем-то подобным и держал на своем сайте (*www.robertpenner.com*) ряд экспериментальных анимаций, выполненных просто для развлечения.

Джим был твердо убежден, что книга должна давать концепции и принципы, которые можно применять самыми разными способами, а не быть просто перечнем работ. Я с радостью согласился. В моей голове вертелось множество идей, о которых хотелось написать, и подобный подход позволял провести достаточно глубокое обсуждение темы.

Итак, эта книга – концептуальное изложение моего подхода к созданию динамической компьютерной графики на языке ActionScript. Ее основными темами являются объектно-ориентированное программирование, тригонометрия, системы координат, векторы и программирование, управляемое событиями. Разобравшись в этих вопросах достаточно хорошо, можно переходить к обсуждению движения, законов физики, а также к окрашиванию и рисованию фигур. Я постараюсь разъяснить читателю основополагающие принципы этих разделов математики и физики и способы реализации их на языке ActionScript. Несомненно, Flash и ActionScript будут развиваться в ближайшие годы, и код, с помощью которого создается анимация, будет меняться вместе с ними. Однако навыки, приобретенные при изучении этой книги, и понимание вечных принципов математики и законов движения помогут читателю перенести код в новый контекст.

О чем эта книга

В части I «Начало» я рассказываю о том, что предшествовало моему увлечению программированием и что сформировало меня как программиста.

Глава 1 «Flash глазами энтузиаста: процесс и дисциплина» автобиографична. Я рассказываю историю моего открытия Flash и делюсь жизненным опытом, повлиявшим на мою работу. Я также излагаю некоторые личные принципы (которые называю «дисциплинами»), формирующие мою повседневную профессиональную деятельность.

Глава 2 «Объектно-ориентированное программирование» является введением в объектно-ориентированный анализ, проектирование и программирование. Она демонстрирует путь, который проходит разработчик от проектных спецификаций до объектно-ориентированного кода, а также способы реализации на языке ActionScript таких концепций объектно-ориентированного программирования, как классы, методы и наследование.

Часть II «Базовые концепции» представляет фундаментальные понятия математики и теории программирования.

Глава 3 «Математика 1: тригонометрия» закладывает математические основы анимации на языке ActionScript и содержит информацию, необходимую для понимания глав, посвященных векторам. Обсуждаются такие ключевые понятия, как углы, треугольники, теорема Пифагора и полярные координаты.

Глава 4 «Математика 2: векторы на плоскости» описывает векторы и способы их реализации на языке ActionScript. Читатель познакомится со сложением векторов, их скалярным произведением и умножением вектора на скаляр, а также увидит воплощение этих и других действий в коде.

Глава 5 «Математика 3: векторы в трехмерном пространстве» переводит обсуждение в третье измерение. Разъясняются трехмерные координаты, их проекция на плоскость экрана и поворот вокруг координатных осей, и все это сопровождается кодом на языке ActionScript.

Глава 6 «Программирование, управляемое событиями» представляет новую модель событий Flash MX и ее отличия от модели, принятой во Flash 5. Объясняется, что такое обработчики событий и слушатели. Кроме того, демонстрируется, как определять пользовательские источники событий, способные рассылать сообщения.

предыдущих глав и распространяет его на такие области, как движение, законы физики, цвет и форма.

Глава 7 «Движение, твининг и ускорение» посвящена концептуальным и математическим основам движения. Исследуются различные уравнения ускорения и разрабатывается объектно-ориентированный подход к реализации твининга в коде.

Глава 8 «Физика» разъясняет ключевые понятия физики: ускорение, трение и сила, а также свойства колебательного движения.

Глава 9 «Раскрашивание объектов средствами языка ActionScript» представляет разнообразные подходы к динамическому окрашиванию графики с помощью класса `Color` среды Flash.

Глава 10 «Рисование средствами языка ActionScript» обсуждает рисование прямых и кривых линий и заливку фигур с использованием Shape Drawing API, интерфейса, принятого во Flash MX. Демонстрируется, как применять его для рисования более сложных фигур.

Часть IV «Обзор проектов» представляет собой углубленное обсуждение четырех моих экспериментальных анимаций.

Глава 11 «Проект Aurora Borealis (Северное сияние)» описывает интерактивную двухмерную систему частиц, имитирующую северное сияние.

Глава 12 «Проект Snowstorm (Снежная буря)» посвящена трехмерной системе частиц, моделирующей полет снежинок при сильном ветре.

Глава 13 «Проект Fractal Dancer (Фрактал-танцор)» представляет читателю так называемое самоподобное[1] дерево (фрактальную структуру), реагирующее на поведение мыши и способное запоминать и воспроизводить движение.

Глава 14 «Проект Cyclone (Вихрь)» описывает систему частиц, представляющую интерактивную модель вихря.

Как читать эту книгу

Вероятно, лучше всего читать эту книгу нелинейным и интерактивным образом. Конечно, ее главы расположены в определенном логическом порядке, и излагаемые в них идеи вытекают друг из друга. Но в то же время какие-то вопросы могут стать до конца понятными лишь при повторном прочтении, когда знаний станет больше. Итак, я предлагаю читателю проделать несколько проходов по тексту для лучшего усвоения информации. При этом можно перескакивать с одного раздела на другой, так сказать, ради «перекрестного опыления» идей.

Исходные файлы

Исходные файлы, упоминающиеся в книге, доступны на сайте издательства *http://www.osborne.com* и на моем личном сайте *http://www.robertpenner.com/profmx/*.

[1] Самоподобие – одно из основополагающих понятий фрактальной геометрии. – *Примеч. науч. ред.*

13

Проект Fractal Dancer (Фрактал-танцор)

«Облака – это не сферы, горы – не конусы, побережье – не полукруг, кора деревьев не гладкая, а молния не проходит по прямой».

– Бенуа Мандельброт (Benoit Mandelbrot),
The Fractal Geometry of Nature, 1983
(Фрактальная геометрия природы)

Я очарован фракталами еще со школы. Существует огромное разнообразие форм и категорий фракталов, но, вообще, они могут быть определены как структуры, подобные сами себе: это фигуры, составленные из своих уменьшенных копий, которые, в свою очередь, составлены из еще более мелких копий, и так далее. Узнав, что такое фракталы, начинаешь видеть их повсюду – в брокколи, кровеносных сосудах, листве и, вслед за Мандельбротом, в облаках, горах, береговой линии, коре и молнии. В качестве упражнения я рекомендую читателю купить головку цветной капусты и всего лишь разглядывать ее некоторое время. Она не просто состоит из веточек, состоящих из веточек, но еще имеет спирали внутри спиралей на своей поверхности.

Я исследую фракталы на своем компьютере еще с тех времен, когда у моих родителей был 286-й, и поначалу для этой цели пользовался программами FractalVision и Fractint в MS-DOS. Когда появилось приложение Flash, я размечтался, как в один прекрасный день достигну такого уровня мастерства, что стану создавать собственные фракталы. В какой-то момент я попытался создать фрактал, который мог бы стать предшественником Fractal Dancer во Flash 4. Было очень трудно, и я бросил эту затею.

Появление Flash 5 возродило былые надежды. Помнится, одной из первых демонстраций новых возможностей Flash 5 было графическое представление знаменитого снегопада Коха (Koch), выполненное Брэнденом Холлом. Его и по сей день можно видеть по адресу *http://www.figleaf.com/development/flash5/koch.swf*. Этот фильм был также первым наглядным примером рекурсивного вызова метода `attachMovie()` во Flash 5.

Препятствие, на которое я натолкнулся во Flash 4, заключалось в том, что в результате копирования клипа новые экземпляры создавались, как его «братья», а не как «потомки», вложенные в него. Зато с помощью метода `attachMovie()` во Flash 5 можно вставлять клип внутрь другого экземпляра того же символа. Имея такой инструмент, я быстро вырастил самоподобное дерево (рис. 13.1).

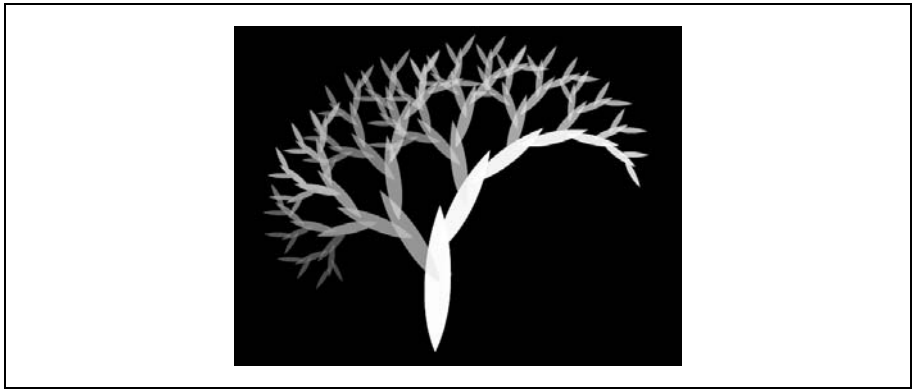


Рис. 13.1. *Самоподобное дерево в проекте Fractal Dancer*

Создав статическую структуру, я, естественно, захотел сделать ее динамической и интерактивной. Мне пришла в голову идея реализации рекурсивного движения в рамках структуры, зависящего от перемещений мыши. Говоря более конкретно, одна координата указателя мыши должна будет управлять левыми ветвями, а другая – правыми.

На дальнейшие шаги меня вдохновила работа Юго Накамуры (Yugo Nakamura) с записанным движением, фрагменты которой можно увидеть на сайте *www.yugop.com*. Я подумал, что фрактал будет еще интереснее, если пользователи смогут не только интерактивно менять его форму, но и записывать его движения, а затем воспроизводить их.

Компонент FractalTree

Перед тем как написать эту главу, я существенно переработал FLA-файл Fractal Dancer из Flash 5, чтобы улучшить структуру кода и воспользоваться новыми функциональными возможностями Flash MX. По ходу дела я превратил дерево в простой компонент, легко настраиваемый на любые пользовательские предпочтения.

Параметры компонента

Компонент `FractalTree` принимает ряд параметров, позволяющих легко настраивать фрактал на пользовательские предпочтения. Они перечислены в табл. 13.1.

Таблица 13.1. Параметры компонента FractalTree

Имя параметра	Значение по умолчанию
maxLevels	6
maxBranches	2
filmFrames	150

Параметр maxLevels определяет, сколько раз ветка дерева выпустит новые ветки. Если его значение равно 2, из ствола вырастут две ветки, и на этом процесс закончится. Если параметр равен 3, каждая из этих двух веток выпустит еще две ветки. Всего на трех уровнях будет $1 + 2 + 4 = 7$ веток. Следующий уровень приведет к появлению еще восьми веток, затем 16, 32 и так далее. Значением параметра maxLevels по умолчанию является 6, в результате чего общее количество ветвей составит $1 + 2 + 4 + 8 + 16 + 32 = 63$.

Примечание

Количество ветвей для n уровней можно рассчитать по формуле $b = 2^n - 1$. Оно растет экспоненциально, с основанием 2.

Параметр maxBranches определяет количество подветвей, вырастающих из одной ветви. Значением по умолчанию является 2, но я разработал компонент FractalTree так, что он может справиться и с большим количеством веток. Читатель может попробовать установить значение 3, а затем преобразовать слой poser3 (в компоненте) из ведущего в обычный. Впрочем, должен предупредить, что количество веток растет очень быстро. На четырех уровнях их 80, а на шести (если ваш процессор справится) – уже 728!

Примечание

Формула общего количества ветвей на n уровнях теперь примет вид: $b = 3^n - 1$. Вообще говоря, для произвольного основания формула такова: $b = maxBranches^n - 1$.

Параметр filmFrames определяет, в течение скольких кадров перемещение мыши записывается и воспроизводится.

Методы

На слое methods определены три метода этого компонента: init(), onMouseDown() и onMouseUp().

Метод init()

Метод init() инициализирует компонент FractalTree. Вот его полный код:

```
this.init = function () {
```

```
    this.posers = [];
    for (var i=1; i <= this.maxBranches; i++) {
        this.posers[i] = this["poser"+i];
        this.posers[i]._visible = false;
    }
    this.xCam = new MotionCam (this.x_txt, "text", this.filmFrames);
    this.xCam.setActor (_level0, "_xmouse");
    this.xCam.setLooping (true);

    this.yCam = new MotionCam (this.y_txt, "text", this.filmFrames);
    this.yCam.setActor (_level0, "_ymouse");
    this.yCam.setLooping (true);
};
```

Вначале я создаю массив posers:

```
this.posers = [];
```

Он будет хранить ссылки на клипы в компоненте FractalTree, определяющие «позу» фрактала. Это две ветки, выращенные под углом к стволу. Их положение, угол поворота и размер в конечном счете определяют форму дерева. Именами их экземпляров являются poser1 и poser2, и каждый находится на слое с тем же именем.

В следующем фрагменте кода я сканирую такие клипы и записываю их в массив posers:

```
for (var i=1; i <= this.maxBranches; i++) {
    this.posers[i] = this["poser"+i];
    this.posers[i]._visible = false;
}
```

Параметр maxBranches используется в цикле for для указания, сколько клипов следует найти. Если он имеет значение по умолчанию (2), то ссылка на клип poser1 будет сохранена в элементе this.posers[1], а ссылка на poser2 — в элементе this.posers[2]. Вот почему значения счетчика цикла i начинаются с 1, а не 0. Одновременно каждый клип делается невидимым.

В следующей секции кода я создаю два объекта, которые будут «снимать на пленку» движение мыши и транслировать их веткам дерева. Эти объекты являются экземплярами класса MotionCam, которые обсуждается далее в этой главе. Первый объект создается так:

```
this.xCam = new MotionCam (this.x_txt, "text", this.filmFrames);
```

Здесь устанавливается объект xCam, управляющий свойством text объекта x_txt класса TextField, который находится на текущей диаграмме времени (на слое textfields). Продолжительность фильма задается параметром компонента filmFrames.

Однако объект класса TextField всего лишь *выводит на экран* данные, поступающие в объект xCam. Поэтому необходимо указать камере цель, «актера», который будет поставлять ей данные. В следующей строчке

кода объект класса MotionCam нацеливается на x-координату указателя мыши:

```
this.xCam.setActor (_level0, "_xmouse");
```

Далее я «зацикливаю» объект xCam вплоть до конца фильма:

```
this.xCam.setLooping (true);
```

Второй объекта класса MotionCam отслеживает данные от свойства _ymouse. Его код почти полностью совпадает с кодом первого объекта:

```
this.yCam = new MotionCam (this.y_txt, "text", this.filmFrames);
this.yCam.setActor (_level0, "_ymouse");
this.yCam.setLooping (true);
```

Разница заключается в замене x на y в нескольких местах. Координата y указателя мыши читается из свойства _level0._ymouse и выводится в текстовое поле this.y_txt. Инициализация компонента на этом завершается.

Обработчик событий onMouseDown()

Код обработчика событий onMouseDown() весьма прост:

```
this.onMouseDown = function () {
    this.xCam.startRecord();
    this.yCam.startRecord();
};
```

Когда пользователь нажимает кнопку мыши, камеры начинают запись. Метод MotionCam.startRecord() рассматривается далее в этой главе.

Обработчик событий onMouseUp()

Код обработчика событий onMouseUp() ничуть не сложнее:

```
this.onMouseUp = function () {
    this.xCam.stopRecord();
    this.yCam.stopRecord();
};
```

Когда пользователь отпускает кнопку мыши, объекты-камеры прекращают запись. Вот и все, что касается кода компонента. Далее мы рассмотрим класс, предназначенный для представления веток фрактала.

Класс FractalBranch

Ветви фрактального дерева являются копиями одного клипа, имеющего определенную структуру. Экземпляры клипа FractalBranchSymbol создаются и затем связываются с другим экземплярами того же символа. Клипы окрашиваются и анимируются в индивидуальном порядке.

В версии проекта Fractal Dancer, подготовленной во Flash 5, я помещал в класс FractalBranchSymbol (который тогда назывался иначе)

функции, занимающиеся копированием, окраской и анимацией. Когда я перерабатывал проект под Flash MX, то обнаружил, что эти функции вполне могут быть методами класса, который я назвал FractalBranch.

Новая функция Object.registerClass() позволила мне превратить FractalBranch в подкласс класса MovieClip. При такой новой структуре внутренние функции класса FractalBranchSymbol лишь однажды объявляются (как методы прототипа класса), в то время как раньше каждый клип-ветка создавал свои копии функций. При значениях, установленных по умолчанию, у дерева будет 63 ветки. Учитывая, что в каждой ветке было пять функций, нетрудно подсчитать, что в старой версии создавалось свыше 300 функций, и для каждой отводилась память. Зато при новом подходе создается только пять функций. Как видим, создание класса, производного от MovieClip, в сочетании с методом Object.registerClass() может реально оптимизировать управление ресурсами в фильме Flash в смысле требований к памяти.

Примечание

Функция Object.registerClass() обычно тесно связана с компонентами, от чего может возникнуть впечатление, что она нужна *только* компонентам. Класс FractalBranch является примером не-компонента, вызывающего функцию Object.registerClass(). Зато компонент FractalTree ее не вызывает.

Конструктор класса FractalBranch

Код класса FractalBranch находится в файле *fractal_branch_class.as*. Как обычно, начинаем с обсуждения конструктора, который в данном случае достаточно прост:

```
_global.FractalBranch = function () {
    this.init();
};
```

Функция-конструктор просто вызывает метод FractalBranch.init() для настройки экземпляра. Брэнден Холл и другие эксперты рекомендуют при кодировании подклассов класса MovieClip, в частности, компонентов, помещать весь инициализирующий код в отдельный метод init(). Впоследствии это облегчит решение некоторых задач.

В следующей строчке кода вызывается метод Object.registerClass(), который связывает библиотечный символ клипа FractalBranchSymbol с классом FractalBranch:

```
Object.registerClass ("FractalBranchSymbol",
                      FractalBranch);
```

С этого момента конструктор FractalBranch() будет вызываться всякий раз, когда создается экземпляр класса FractalBranchSymbol. Кроме того, экземпляр клипа унаследует методы класса. Рассмотрим методы класса FractalBranch.

Методы

Во-первых, я устанавливаю цепочку прототипов, чтобы класс `FractalBranch` наследовал от класса `MovieClip`:

```
FractalBranch.prototype.__proto__ = MovieClip.prototype;
```

Наследование с использованием конструкции `__proto__` обсуждается в главе 2.

Затем я прибегаю к обычному приему объявления переменной-ярлыка для прототипа класса:

```
var FBP = FractalBranch.prototype;
```

Теперь можно создавать методы и присоединять их к `FBP`. Начнем с метода `init()`.

Метод FractalBranch.init()

Метод `init()` подготавливает экземпляр класса `FractalBranch`, который является подклассом клипа, к существованию в качестве элемента фрактального дерева. Вот код этого метода:

```
FBP.init = function () {
    this.numBranches = 0;
    if (this.myLevel == undefined) {
        this.myLevel = 0;
        this.fRoot = this._parent;
    } else {
        this.doColor();
        if (this.myBranch == 1)
            this.fRoot.xCam.addListener (this);
        else if (this.myBranch == 2)
            this.fRoot.yCam.addListener (this);
    }
};
```

Вначале обнуляется свойство `numBranches`, поскольку к текущей ветви еще не присоединена ни одна ветвь:

```
this.numBranches = 0;
```

Затем оператор `if` проверяет, является ли данная ветка стволом дерева:

```
if (this.myLevel == undefined) {
    this.myLevel = 0;
    this.fRoot = this._parent;
}
```

Дерево сконструировано так, что каждой подветви присваивается уровень, который хранится в ее свойстве `myLevel`. Таким образом, если свойство `myLevel` не определено, то ветвь не является ничьей подветвью, то есть это ствол с уровнем 0.

Кроме того, определяется свойство fRoot, которое будет хранить ссылку на компонент FractalTree (вычисленную с помощью свойства this._parent). Ссылка fRoot впоследствии будет передана каждой ветке дерева, что облегчит доступ к компоненту.

С другой стороны, если ветка НЕ является стволом дерева, то выполняется конструкция else:

```
} else {
    this.doColor();
    if (this.myBranch == 1)
        this.fRoot.xCam.addListener (this);
    else if (this.myBranch == 2)
        this.fRoot.yCam.addListener (this);
}
```

В первой строчке этой конструкции вызывается метод doColor(), изменяющий цвет ветки. Оператор if проверяет номер ветки, хранящийся в свойстве myBranch. В этом фильме правая ветка имеет номер 1, а левая – номер 2.

В конце кода метода текущая ветка добавляется в качестве слушателя к одному из объектов класса MotionCam – либо к xCam, либо к yCam:

```
this.fRoot.xCam.addListener (this);
// или
this.fRoot.yCam.addListener (this);
```

В результате ветка будет получать события от объекта MotionCam, причем правыми ветками будет управлять свойство _xmouse, а левыми – _ymouse. Но для реализации этих замыслов необходимо определить обработчики событий onMotionChanged().

Обработчик событий FractalBranch.onMotionChanged()

Как было только что показано в методе init(), экземпляр класса FractalBranch является слушателем событий, поступающих от экземпляра класса MotionCam (либо xCam, либо yCam). Класс MotionCam рассылает целый ряд различных событий, но нас интересует только одно – onMotionChanged. Поэтому мы и определяем метод onMotionChanged(), реагирующий на это событие:

```
FBP.onMotionChanged = function (source, position) {
    if (position != undefined)
        this._rotation = position + 175;
};
```

Мы хотим, чтобы координаты мыши определяли поворот ветки, то есть чтобы при изменении координат соответственно обновлялось свойство _rotation. Левая ветка принимает вертикальную координату мыши, а правая – горизонтальную.

Координаты нового положения указателя мыши рассылаются экземпляром класса MotionCam посредством аргумента position. В коде обработчика событий выполняется несложная проверка того факта, что новое положение указателя определено. Проверка необходима, поскольку фильм объекта MotionCam изначально, то есть до записи данных, заполнен значениями undefined. Мы же хотим, чтобы ветка двигалась только в случае приема допустимых значений.

В конце устанавливается свойство _rotation, причем к значению position прибавляется 175. Это число произвольно; я просто подстраиваю размер фигур, рисуемых мышью, под размер экранной области.

Метод FractalBranch.doColor()

Первоначально я окрашивал все ветви фрактального дерева в один цвет. Однако через некоторое время мне захотелось придать дереву более интересный внешний вид. Когда я сделал все ветви полупрозрачными, стало лучше, но недоставало цветового разнообразия.

Я испробовал различные схемы динамического окрашивания и, в конце концов, пришел к следующему коду:

```
FBP.doColor = function () {
    if (this.myBranch == 1)
        var tint = {r:255, g:255, b:255};
    else
        var tint = {r:0, g:0, b:255};

    (new Color (this)).setTint (tint.r,
                                tint.g,
                                tint.b,
                                30);
};
```

Дерево окрашивается по двум направлениям. Левая ветка получает 30% белого цвета. Правая ветка окрашивается 30% синим цветом. Для этих целей я применяю свой метод Color.setTint(), описанный в главе 9.

Я вызываю его как метод безымянного объекта Color, пользуясь синтаксисом (new Color (this)). Поскольку изменение цвета представляет собой одноразовую акцию, нет нужды хранить объект класса Color, а отработавший безымянный объект автоматически уничтожается.

Метод FractalBranch.newBranch()

Метод newBranch() присоединяет новую ветку к текущей при помощи следующего кода:

```
FBP.newBranch = function (bNum) {
    if (this.myLevel >= this.fRoot.maxLevels) return;
    var p = this.fRoot.posers[bNum];
    var initObj = {
```

```
            _x: p._x,
            _y: p._y,
            _xscale: p._xscale,
            _yscale: p._yscale,
            _rotation: p._rotation,
            myLevel: this.myLevel + 1,
            myBranch: bNum,
            fRoot: this.fRoot
        }
        this.attachMovie ("FractalBranchSymbol",
                          "b" + bNum,
                          bNum,
                          initObj);
        this.numBranches++;
    };
```

Метод принимает один параметр, `bNum`. В данном фильме он может иметь значение 1 или 2, которое идентифицирует присоединяемую ветку (левая или правая).

Для начала оператор `if` решает, следует ли вообще создавать новую ветвь:

```
if (this.myLevel >= this.fRoot.maxLevels) return;
```

Разбиение на подветви должно происходить только определенное количество раз, установленное параметрами компонента `FractalTree`. Ссылка на компонент хранится в объекте `this.fRoot`, а максимальное количество уровней – в его свойстве `this.fRoot.maxLevels`. По умолчанию оно равно 6.

Далее объявляется локальная переменная `p`, и в ней сохраняется ссылка на соответствующий клип, определяющий «позу» фрактала:

```
var p = this.fRoot.posers[bNum];
```

Такой клип может быть позиционирован на этапе проектирования внутри компонента, и от этого будет зависеть форма дерева. Ссылки на эти клипы хранятся в массиве `this.fRoot.posers`, и аргумент `bNum` используется для доступа к ним в качестве индекса. Если новая подветвь является правой, `bNum` равен 1, и нужный клип хранится в элементе `this.fRoot.posers[1]`.

В следующей секции кода создается инициализирующий объект `initObj`, а затем присоединяется новый библиотечный клип:

```
var initObj = {
    _x: p._x,
    _y: p._y,
    _xscale: p._xscale,
    _yscale: p._yscale,
    _rotation: p._rotation,
    myLevel: this.myLevel + 1,
```

```
        myBranch: bNum,
        fRoot: this.fRoot
    }
    this.attachMovie ("FractalBranchSymbol",
                      "b" + bNum,
                      bNum,
                      initObj);
```

Во Flash MX у метода `MovieClip.attachMovie()` появился четвертый аргумент, `initObj`. Если в качестве четвертого аргумента передать объект, то все свойства этого объекта будут скопированы в новый экземпляр клипа, создаваемый методом `attachMovie()`.

В данном случае я создал объект `initObj` для хранения значений `_x`, `_y`, `_xscale`, `_yscale` и `_rotation`, нужных для новой ветви, а также и ряда других свойств. Визуальные свойства копируются из позиционирующего клипа. Если клип для правой ветки повернут на 35°, каждая правая подветвь будет повернута относительно своей родительской ветки. Аналогичным образом, если клип левой ветки составляет 75% от оригинального размера, то значения его свойств `_xscale` и `_yscale` будут равны 75, и каждая левая подветвь будет иметь такие относительные пропорции.

Метод заканчивается инкрементированием свойства `numBranches`:

```
this.numBranches++;
```

Это свойство нужно методу `nextBranch()`, который обсуждается в следующем разделе.

Метод FractalBranch.nextBranch()

В самом начале своего существования фрактальное дерево состоит из одной ветви – ствола. Затем одна за другой появляются новые ветки, пока не будет сформировано все дерево.

Метод `nextBranch()` переходит к построению очередной ветки в этой последовательности. «Следующая ветка» определяется рекурсивным образом в таком коде:

```
FBP.nextBranch = function () {
    if (this.numBranches < this.fRoot.maxBranches
        && this.myLevel < this.fRoot.maxLevels) {
        this.newBranch (this.numBranches + 1);
    } else {
        this._parent.nextBranch();
    }
};
```

Говоря простым языком, логика этого метода такова: если веток и уровней еще не слишком много, то присоединить к данной ветви новую подветвь. В противном случае велеть родительской ветке «вырастить» новую ветку.

Класс MotionCam

Ранее в этой главе, изучая компонент FractalTree, мы увидели, как два объекта класса MotionCam записывают и воспроизводят координаты мыши. Объекты xCam и yCam являются экземплярами класса, специально созданного для «съемок» меняющейся информации. Класс MotionCam нацеливается на объект-актер, записывает значения одного из его свойств в свой фильм и по требованию воспроизводит эту информацию.

Код класса MotionCam хранится в файле *motioncam_class.as*.

Конструктор класса MotionCam

У конструктора класса MotionCam всего три аргумента: obj, prop и duration. Этот класс является подклассом класса Motion, обсуждаемого в главе 7. Подобно своему суперклассу, класс MotionCam предназначен для управления конкретным свойством объекта, которое доступно при помощи конструкции obj[prop]. Аргумент duration задает максимальную продолжительность снимаемого движения. В классе Motion можно было указывать продолжительность либо в кадрах, либо в секундах. Однако в классе MotionCam ее нужно указывать в кадрах. Это необходимо для того, чтобы фильм был записан и воспроизведен правильно.

Конструктор имеет следующий код:

```
_global.MotionCam = function (obj, prop, duration) {
    this.superCon (obj, prop, obj[prop], duration);
    this.setActor (obj, prop);
    this.film = [];
};
```

В первой строчке кода вызывается конструктор суперкласса с соответствующими параметрами:

```
this.superCon (obj, prop, obj[prop], duration);
```

Параметрами конструктора Motion() являются obj, prop, begin и duration. Поэтому параметры obj, prop и duration отображаются без изменений, а в качестве третьего параметра, begin, передается obj[prop]. Это означает, что исходное значение (begin) контролируемого свойства (prop) определяется не пользователем, а текущим значением самого свойства.

Затем определяется «актер», которого будет «снимать» камера:

```
this.setActor (obj, prop);
```

Экземпляр класса MotionCam наблюдает за свойством prop объекта obj (то есть за obj[prop]), записывая изменяющиеся данные. Аналогия из обычной жизни: мы можем так установить видеокамеру, чтобы она записывала высоту полета птицы. Птица будет объектом, а высота полета – ее свойством.

Напоследок в камеру заряжается пустая пленка:

```
this.film = [];
```

Данные фильма хранятся в свойстве-массиве по имени `this.film`. По ходу записи данные индексируются значениями времени. Например, данные для момента номер 24 хранятся в элементе `this.film[24]`.

Открытые методы

Чтобы класс `MotionCam` стал подклассом класса `Motion`, я вызываю свой метод `Function.extend()`, устанавливающий цепочку наследования:

```
MotionCam.extend (Motion);
```

В результате класс `MotionCam` наследует все методы класса `Motion`.

Теперь требуется определить дополнительные методы класса `MotionCam`. Для начала создадим переменную-ярлык, указывающую на `MotionCam.prototype`:

```
var MCP = MotionCam.prototype;
```

Методы класса `MotionCam` будут присоединяться к `MCP`.

Методы MotionCam.startRecord() и stopRecord()

Методы `startRecord()` и `stopRecord()` включают/выключают режим записи объекта `MotionCam`. Их код очень прост и сводится к переключению свойства `isRecording` между значениями `true` и `false`:

```
MCP.startRecord = function () {
    this.isRecording = true;
};

MCP.stopRecord = function () {
    this.isRecording = false;
};
```

Метод MotionCam.cutFilm()

Когда происходит съемка движения экземпляром класса `MotionCam`, часто не представляется возможным заранее определить количество кадров будущего фильма. Поэтому приходится начинать съемку с неустановленным свойством `duration`, записывать столько, сколько нужно, и получить в результате какое-то количество записанных кадров в массиве `film`. В такой ситуации у «оператора» может возникнуть желание «подрезать» фильм до определенной длины. При последующем воспроизведении фильм автоматически остановится, когда достигнет конца (или, в зависимости от значения свойства `isLooping`, перемотается в начало и возобновит свое воспроизведение).

Метод `cutFilm()` имеет следующий код:

```
MCP.cutFilm = function (t) {
```

```
    if (t == undefined) var t = this.$time;
    this.film.length = t + 1;
    this.setDuration (t);
};
```

Обратите внимание, что метод позволяет указывать конкретный момент обрезки фильма. Если же параметр не определен, фильм будет обрезан в текущий момент времени.

Метод MotionCam.eraseFilm()

Метод `eraseFilm()` стирает фильм в экземпляре класса `MotionCam`:

```
MCP.eraseFilm = function () {
    this.film = [];
    this.setDuration (0);
};
```

Данные стираются, когда свойству `film` присваивается новый массив, а длина обнуляется.

Метод MotionCam.toString()

Метод `toString()` посылает в окно вывода резюме объекта:

```
MCP.toString = function () {
    return "[MotionCam prop=" + this.$prop + " t=" + this.$time +
          " pos=" + this.$position + " mode: " + this.mode + "]";
};
```

Метод `MotionCam.toString()` переопределяет метод своего базового класса `Motion.toString()`.

Метод MotionCam.print()

Метод `print()` предоставляет простой способ отправки фильма объекта `MotionCam` в окно вывода в виде строки значений, разделенных запятыми, например, «83,9,41,22,53,10». Его код:

```
MCP.print = function () {
    trace (this.getFilmString());
};
```

Метод `MotionCam.getFilmString()` будет рассмотрен чуть позже. Он преобразует массив-фильм в строку, в которой значения разделены запятыми.

Методы чтения и установки

В дополнение к открытым методам в классе `MotionCam` определен ряд методов чтения/установки, предоставляющих доступ к таким характеристикам, как положение, объект съемки и фильм.

Метод MotionCam.getPosition()

Метод `getPosition()` возвращает положение в указанный момент времени. Его код:

```
MCP.getPosition = function (t) {
    if (t == undefined) t = this.$time;
    return this.film[t];
};
```

Положение считывается из массива `film`, причем время используется в качестве индекса. Если параметр `t` опущен, принимается текущее время.

Примечание

Метод MotionCam.getPosition() переопределяет метод своего базового класса Motion.getPosition() (который является абстрактным; подробности см. в главе 7).

Методы MotionCam.setActorObj() и getActorObj()

Как говорилось выше, объект `MotionCam` нацеливается на объект-актер и записывает информацию о нем. Методы `setActorObj()` и `getActorObj()` позволяют задать объект-актер или обратиться к нему:

```
MCP.setActorObj = function (ao) {
    this.actorObj = ao;
};

MCP.getActorObj = function () {
    return this.actorObj;
};
```

Методы MotionCam.setActorProp() и getActorProp()

Помимо нацеливания экземпляра класса `MotionCam` на объект-актер, необходимо уточнить, какое свойство актера подлежит записи. Объект `MotionCam` может записывать значения только одного свойства за сеанс, поскольку его фильм одномерный. Имя записываемого свойства хранится в виде строки. Методы `setActorProp()` и `getActorProp()` предоставляют доступ к этой строке, как показано в следующем коде:

```
MCP.setActorProp = function (ap) {
    this.$actorProp = ap;
};

MCP.getActorProp = function () {
    return this.$actorProp;
};
```

Метод MotionCam.setActor()

Метод `setActor()` является удобным сплавом методов `setActorObj()` и `setActorProp()`:

```
MCP.setActor = function (ao, ap) {
    this.setActorObj (ao);
    this.setActorProp (ap);
};
```

Напомню, что при обсуждении метода `init()` компонента `FractalTree` было продемонстрировано, как с помощью метода `setActor()` объекты `xCam` и `yCam` нацеливаются на координаты указателя мыши. Код был такой:

```
this.xCam.setActor (_level0, "_xmouse");
// ...
this.yCam.setActor (_level0, "_ymouse");
```

Методы MotionCam.setFilm() и getFilm()

Метод `setFilm()` позволяет «заправить» в камеру новую «катушку с пленкой», то есть массив. И наоборот, метод `getFilm()` возвращает массив-фильм:

```
MCP.setFilm = function (film_arr) {
    this.film = film_arr;
};

MCP.getFilm = function () {
    return this.film;
};
```

Методы MotionCam.setFilmString() и getFilmString()

Описанные выше методы `setFilm()` и `getFilm()` обращаются с фильмом как с объектом-массивом. Существует альтернативная возможность загрузки и сохранения фильма в виде строки, содержащей последовательность значений, разделенных запятыми. Например, строка записанных положений может выглядеть так: «24,53,62,59,86,23,123». Имея такую строку, можно загружать ее в объект `MotionCam` с помощью метода `setFilmString()`:

```
MCP.setFilmString = function (str) {
    this.film = str.split (",");
};
```

Для строки, переданной в качестве параметра, вызывается метод `String.split()`, разбивающий ее на элементы массива, считая запятые разделителями.

И наоборот, если потребуется сохранить записанный фильм в виде строки, можно вызвать метод `getFilmString()`. Его код:

```
MCP.getFilmString = function () {
    return this.film.toString();
};
```

Я вызываю метод `Array.toString()` для массива `this.film`, и строка с запятыми-разделителями генерируется автоматически.

Закрытые методы

У данного класса есть только один закрытый метод, `MotionCam.update()`, который переопределяет метод своего базового класса `Motion.update()` и имеет следующий код:

```
MCP.update = function () {
    if (this.isRecording)
        with (this) film[$time] = $actorObj[$actorProp];
    super.update();
};
```

Метод проверяет, находится ли камера в режиме записи, и если это так, записывает в фильм текущее положение объекта-актера. Затем он вызывает метод `update()` своего базового класса, чтобы обновить экран в соответствии с фильмом.

Данные объекта-актера можно получить с помощью свойства `this.actorObj[this.actorProp]`. Текущий кадр фильма расположен в элементе массива `this.film[this.$time]`. Сохранение положения объекта сводится к копированию значения из одного места в другое:

```
this.film[this.$time] = this.actorObj[this.actorProp];
```

Для сокращения кода и ускорения его выполнения я применил оператор `with (this)`:

```
with (this) film[$time] = actorObj[actorProp];
```

В завершение вызывается метод базового класса `Motion.update()` (обсуждаемый в главе 7). Приведу здесь его код для справки:

```
Motion.prototype.update = function () {
    this.setPosition (this.getPosition (this.$time));
};
```

Заключение

В этой главе обсуждался возможный подход к построению интерактивной фрактальной структуры, который заключался в применении двух классов, `FractalBranch` и `MotionCam`. Первый из них является подклассом класса `MovieClip`, а второй – класса `Motion`, описанного в главе 7. Я надеюсь, класс `MotionCam` окажется полезным читателю и в других ситуациях, когда потребуется записать и воспроизвести данные. В следующей главе будет рассмотрена имитация еще одного природного явления – вихря.